FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" – UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIAS DA COMPUTAÇÃO

CRISTIANE DA SILVA MORONY

IMPLEMENTAÇÃO E DESEMPENHO EM HARDWARE DOS ALGORITMOS GLOBAL E LOCAL DE SEQUENCIAMENTO DE GENS

MARÍLIA
2006

CRISTIANE DA SILVA MORONY

IMPLEMENTAÇÃO E DESEMPENHO EM HARDWARE DOS ALGORITMOS GLOBAL E LOCAL DE SEQUENCIAMENTO DE GENS

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciências da Computação. (Área de Concentração: Arquitetura de Computadores).

Orientador
Prof. Dr. Edward David Moreno Ordoñez

MARÍLIA
2006

CRISTIANE DA SILVA MORONY

IMPLEMENTAÇÃO E DESEMPENHO EM HARDWARE DOS ALGORITMOS GLOBAL E LOCAL DE SEQUENCIAMENTO DE GENS

Banca examinadora da dissertação apresentada ao Programa de Mestrado da UNIVEM,/F.E.E.S.R., para obtenção do Título de Mestre em Ciências da Computação. Área de Concentração: Arquitetura de Computadores.

Orientador: Prof. Dr. Edward David Moreno Ordonez

1º Examinador: Prof. Dra. Kalinka Regina J. L. Castelo Branco

2º Examinador: Prof. Dr. Norian Marranghello

Marília, fevereiro de 2006.

AGRADECIMENTOS

A Deus, pelo dom da vida, da paciência e pela sua energia divina sempre presente em minha existência.

Ao Prof. Dr. Edward David Moreno Ordonez, pela amizade, confiança, por acreditar no meu potencial, por estar sempre presente, conduzindo de forma coerente todo o processo de construção e extrema competência na orientação deste trabalho.

A meus pais, Ivalter e Sueli e ao meu noivo, Fausto, pelas vossas presenças constante e efetiva, que com seus estímulos e suas ajudas me fizeram enfrentar e ultrapassar todos os desafios.

A todos os professores do programa de pós-graduação da Fundação de Ensino Eurípides Soares da Rocha de Marília e ao amigo e professor César Giacomini Penteado pelos conhecimentos e experiências que me foram passados.

A todos que de alguma forma contribuíram para essa minha experiência, direta ou indiretamente.

Obrigada.

"... Viver é ser capaz de extrair de cada fato uma lição, de cada ato um ensinamento, de cada acontecimento um aprendizado. "

Hemus.

MORONY, Cristiane da Silva. **Implementação e desempenho em hardware dos algoritmos global e local de sequenciamento de gens**. 2006. 164 f. Dissertação (Mestrado em Ciências da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, fevereiro de 2006.

RESUMO

A comparação de seqüências genéticas é uma das operações fundamentais para a computação biológica, por isso, biólogos moleculares comparam frequentemente seqüências de DNA, RNA e proteínas, com um conjunto de outras seqüências desconhecidas na tentativa de detectar altas propriedades entre elas que implicam em similaridades estruturais e funcionais. O projeto de pesquisa desta dissertação de mestrado tem como objetivo implementar em *hardware*, dispondo da tecnologia FPGA, dos algoritmos que são considerados ou adotados como padrão para a comparação e o alinhamento global e local das seqüências genéticas de DNA utilizando a técnica de programação dinâmica, cujo método computacional calcula o melhor alinhamento possível entre as seqüências. Foram implementados e analisados algoritmos de alinhamento global (*Needleman-Wunsch*) e local (*Smith-Waterman*), pois, são considerados de extrema importância pela comunidade acadêmica na bioinformática. Os algoritmos primeiramente foram implementados em *software* (linguagem C) e posteriormente a implementação em *hardware*, onde os algoritmos foram descritos na linguagem VHDL e prototipados usando a tecnologia FPGA, visando conseguir o melhor desempenho possível. Finalizando, comparou-se os tempos de execução dos resultados alcançados tanto em *software*, *hardware* e com outros sistemas voltados para o seqüenciamento genético e dentre todas as análises, nossa implementação mostrou-se mais rápida do que algumas plataformas específicas para problemas de bioinformática.

MORONY, Cristiane da Silva. **Implementação e desempenho em hardware dos algoritmos global e local de sequenciamento de gens**. 2006. 164 f. Dissertação (Mestrado em Ciências da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, fevereiro de 2006.

ABSTRACT

The comparison of genetic sequences is one of the fundamental operations to the biological computing because of this, molecular biologists frequently compare sequences of DNA, RNA and proteins, with a group of other unknown sequences trying to detect high priorities among them that imply in, structural and functional similarities. The research project of this dissertation has as main objective to implement, in hardware - making use of FPGA technology, the algorithms that are considered or adopted as standard for the comparison and the global and local alignment of the DNA genetic sequences. These algorithms use the dynamic programming technique, whose computational method calculates the best possible alignment among the sequences. Thus, we have implemented and analyzed algorithms of global alignment (Needleman-Wunsch) and local (Smith-Waterman), since they are considered of extreme importance for the academical community in the bioinformatics. Firstly, these algorithms were implemented in software (C language) and, later, the hardware implementation for using VHDL and FPGAs.Finally, we compare our obtained software and hardware results, by the execution time of the algorithms, and it was possible to verify that our hardware implementation is better than our software version and others bioinformatics-specific platforms.

LISTA DE ILUSTRAÇÕES

FIGURA 1.1 – Modelo estrutural do DNA 19

FIGURA 1.2 – Exemplo de alinhamento entre duas seqüências de DNA 20

FIGURA 2.1 – Exemplo de duas seqüências genéticas 34

FIGURA 2.2 – Alinhamento de duas seqüências de proteínas 36

FIGURA 2.3 – Comparação de seqüências genéticas e suas formas de alinhamento 36

FIGURA 2.4 – Matriz de pontos utilizando duas seqüências de DNA 39

FIGURA 2.5 – Matriz de pontos compara duas seqüências de DNA 39

FIGURA 2.6 – Preenchimento de uma célula de uma dada matriz...................................44

FIGURA 2.7 – Representação da menor distância 44

FIGURA 2.8 – Comparação global de duas seqüências de DNA 45

FIGURA 2.9 – Exemplo de duas seqüências de DNA..46

FIGURA 2.10 – Inicialização da matriz de programação dinâmica para o alinhamento global 46

FIGURA 2.11 – Inicialização da primeira linha e coluna da matriz A 47

FIGURA 2.12 – Preenchimento da posição $A_{1,1}$ da matriz A 48

FIGURA 2.13 – Preenchimento da posição $A_{1,2}$ da matriz A 48

FIGURA 2.14 – Representação da segunda fase – Matriz A preenchida..49

FIGURA 2.15 – Representação da ordem preferencial da verificação

do *traceback* 50

FIGURA 2.16 – Representação do passo inicial do processo de *traceback* 50

FIGURA 2.17 – Matriz A após ter encontrado o primeiro alinhamento............................51

FIGURA 2.18 Continuidade do processo de *traceback* 51

FIGURA 2.19 – Prosseguimento do processo de *traceback* 52

FIGURA 2.20 – *Traceback* no alinhamento global..52

FIGURA 2.21 – Possível alinhamento da matriz A 52

FIGURA 2.22 – Sistema de *score* da matriz A 52

FIGURA 2.23 – Diversas soluções alternativas da matriz A para o algoritmo de alinhamento global 53

FIGURA 3.1 – Alinhamento de diversas seqüências 62

FIGURA 3.2 – Performances dos sistemas VLSI e FPGA 65

FIGURA 3.3 – Representação das Performances em CPS...70

FIGURA 3.4 – Representação das Performances em CPOS...70

FIGURA 3.5 – Sistema completo do *hardware* SAMBA 72

FIGURA 3.6 – Comparação de seqüência sob um *array* sistólico linear 74

FIGURA 3.7 – Arquitetura de cada elemento de processamento do *Splash* 75

FIGURA 3.8 – Diagrama de Fluxo da Programação do *Splash*-2 76

FIGURA 3.9 – Princípio da arquitetura do filtro HScan..79

FIGURA 3.10 – *Speed-up* entre o HScan e os *softwares* BLASTN e FASTA 81

FIGURA 3.11 – *Array* linear SSR com fluxo de dados 82

FIGURA 3.12 – Visão da placa de FPGAs agentes e FPGA de controle 85

FIGURA 3.13 – Esquema simplificado de uma placa do POLYP.....................................87

FIGURA 3.14 – Cálculo do valor da diagonal da estrutura sistólica para comparação de seqüências genéticas..91

FIGURA 4.1 – Cálculo da matriz de programação dinâmica no alinhamento Morony-Global implementada na linguagem de programação C...97

FIGURA 4.2 – Resultado obtido do alinhamento ótimo no alinhamento global em C..97

FIGURA 4.3 – Cálculo da matriz de programação dinâmica no alinhamento local, Morony-Local implementada na linguagem de programação C...100

FIGURA 4.4 – Resultado obtido do alinhamento ótimo no alinhamento local em C..100

FIGURA 4.5 – Performances das implementações em C dos algoritmos Global e Local 102

FIGURA 4.6 – Performances do tempo consumido em segundos versus tamanho das seqüências 104

FIGURA 4.7 – Performances dos alinhamentos com tamanhos de

seqüências até 300 caracteres 105

FIGURA 5.1 – Representação das operações dos algoritmos Morony-Globla e Morony-Local 108

FIGURA 5.2 – Fluxograma das Operações no AGM e ALM..110

FIGURA 5.3 – Representação Estrutural para o Alinhamento Global...............................111

FIGURA 5.4 – Representação do diagrama de estados do algoritmo Global utilizando o recurso de máquina de estado....................................113

FIGURA 5.5 – Representação do diagrama de estados do algoritmo Global otimizado 114

FIGURA 5.6 – Representação do diagrama de estados do algoritmo Local 115

FIGURA 5.7 – Representação do diagrama de estados do algoritmo Local otimizado 116

FIGURA 5.8 – Representação do estado 1 na ferramenta de simulação *Xilinx*...119

FIGURA 5.9 – Representação do estado 2 na ferramenta de simulação *Xilinx*...120

FIGURA 5.10 – Representação do estado três na ferramenta de simulação *Xilinx* 122

FIGURA 5.11 – Representação da matriz do algoritmo Global preenchida utilizando a linguagem VDHL..122

FIGURA 5.12 – Representação do estado 4 na ferramenta de simulação em VHDL...125

FIGURA 5.13 – Procedimento *traceback* na simulação em VHDL no algoritmo Global otimizado..125

FIGURA 5.14 – Cálculo do alinhamento ótimo obtido no algoritmo Global otimizado em VHDL 125

FIGURA 5.15 – Recursos utilizados da família VIRTEX V800FG680 nos algoritmos Global, Global Otimizado, Local e Local Otimizado 128

FIGURA 5.16 – Recursos utilizados da família VIRTEXE V812EFG900 nos algoritmos Global, Global Otimizado, Local e Local Otimizado 129

FIGURA 5.17 – Tempo de execução em nanosegundos das implementações em *software* e *hardware* 132

LISTA DE TABELAS

Tabela 1.1 – Sistemas de *hardware* destinados à solução da bioinformática......................27

Tabela 3.1 – Medida de Performance em *Characters Processed per Second*.....................69

Tabela 3.2 – Medida de Performance em *Comparison Operations per Second* 70

Tabela 3.3 – Desempenho obtido através da ferramenta de síntese 92

Tabela 4.1 – Performances dos algoritmos Global e Local em C 101

Tabela 4.2 – Performances do tempo consumido em segundos...103

Tabela 5.1 – Descrição dos estados do algoritmo Global...114

Tabela 5.2 – Descrição dos estados do algoritmo Global otimizado..................................114

Tabela 5.3 – Descrição dos estados do algoritmo Local 115

Tabela 5.4 – Descrição dos estados do algoritmo Local otimizado 116

Tabela 5.5 – Tempo em minutos consumido dos algoritmos Global e Local otimizados em VHDL...126

Tabela 5.6 – Quantidade de *clocks* dos estados nos algoritmos Global e Local em VHDL..126

Tabela 5.7 – Quantidade de *clocks* dos estados nos algoritmos Global e Local otimizados em VHDL 127

Tabela 5.8 – Recursos utilizados no FPGA VIRTEX V800FG680 127

Tabela 5.9 – Recursos utilizados no FPGA VIRTEXE V812EFG900 129

Tabela 5.10 – Temporização dos algoritmos Global e Local das versões otimizadas em VHDL com a família VIRTEX 130

Tabela 5.11 – Temporização dos algoritmos Global e Local e as otimizações em VHDL com a família VIRTEXE..130

Tabela 5.12 – Performance da implementação em *software* (em C)..................................131

Tabela 5.13 – Performance da implementação em *hardware* (em FPGA) 131

Tabela 5.14 – Desempenho de tempo em nanosegundos 133

LISTA DE ABREVIAÇÕES

ASIC - *Application Specific Integrated Circuit*

BioSCAN - *Biological Sequence Comparative Analysis Node*

BISP - *Biological Information Signal Processor*

BLAST – *Basic Local Alignment Search Tool*

BLOSUM - ***BLO**cks **SU**bstitution **M**atrices*

B-SYS - *Brown SYStolic Array*

CLBs - *Configurable Logic Blocks*

COPS - *Comparison Operations per Second*

CPS - *Characters Processed per Second*

CPUs - *Central Processing Unit*s

DNA – *Dexoxyribonucleic Acid*

FPGA – *Field Programmable Gate Array*

GPP – *General Purpose Processor*

I/O - *Input/Output*

IOBs – *In/Out Block*

IRISA - *Institut de Recherque en Informatique et Systèmes Aléatories*

LUTs - *Look Up Table*

MGAP – *Micro Grain Arithmetic Processor*

MIMD - *Multiple Instruction Multiple Data*

mRNA – RNA (*Ribonucleic Acid*) mensageiro

MTA – *Multithreaded Architecture*

NCBI – *National Center for Biotechnology Information*

NGEN - *eNGiNE*

PAM - *Point Accepted Mutation*

PCs - Computadores Pessoais

PCI – *Peripherical Component Interconnect*

PLDs – *Programmable Logic Devices*

POLYP - *Parallel OnLine PolYmer Processing*

RAM - *Random Access Memory*

RISC - *Reduced Indtruction Set Computer*

SAMBA - *Systolic Accelerator for Molecular Biological Application*

SIMD - *Single Instruction Multiple Data*

SSR - *Systolic Shared Register*

VHDL – ***V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage*

VLSI - *Very Large Scale Integration*

VME - *Versa Module Europ*

WU – Universidade de *Washington*

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO 17

1.1 Conceitos Básicos do Projeto 18

1.1.1 Bioinformática 18

1.1.2 Comparação e Alinhamento de Seqüências Genéticas 18

1.1.3 Métodos de Alinhamento de Seqüências 21

1.1.4 Programação Dinâmica para o Alinhamento Global e Local 22

1.1.5 A Importância do Desenvolvimento dos Algoritmos de Alinhamento de Seqüências em Bioinformática 23

1.1.6 FPGA e VHDL 24

1.2 Trabalhos Correlatos...25

1.3 Objetivos da Dissertação 28

1.4 Justificativa...28

1.5 Organização da Dissertação...29

CAPÍTULO 2 – ALINHAMENTO DE SEQÜÊNCIAS 31

2.1 Considerações Iniciais 31

2.1.1 Comparação de Seqüências Genéticas 32

2.1.2 Alinhamento de Seqüências Genéticas...34

2.2 Métodos de Alinhamento de Seqüências Genéticas 35

2.2.1 Alinhamento por Matriz de Pontos (*Dot Plot* / *Dot Matrix*) 38

2.2.2 Alinhamento por Programação Dinâmica 40

2.2.2.1 Definição de uma Distância de Edição 41

2.2.2.2 Cálculo de uma Distância de Edição: o Algoritmo de Base 42

2.3 Alinhamento Global utilizando a Técnica de Programação Dinâmica 45

2.4 Alinhamento Local utilizando o Método de Programação Dinâmica 53

2.5 Alinhamento Semi-Global Utilizando o Método de Programação Dinâmica 54

2.6 Alinhamento Múltiplo de Seqüências Genéticas 56

CAPÍTULO 3 – *SOFTWARES* E *HARDWARES* PARA A COMPARAÇÃO DE SEQÜÊNCIAS 58

3.1 Considerações Iniciais 58

3.2 *Softwares* Utilizados para a Comparação de Seqüências Biológicas 59

3.2.1 FASTA 60

3.2.2 BLAST 61

3.2.3 *Clustal* W 63

3.3 *Hardware* para Comparação de Seqüências Biológicas 64

3.3.1 *Arrays* Dedicados VLSI - BioSCAN 66

3.3.2 Arquitetura RISC de Propósito Geral – SUN 690 68

3.3.3 Arquitetura Vetorizada – CONVEX 240 68

3.3.4 Arquitetura SIMD – MP1 69

3.4 *Arrays* Dedicados VLSI - SAMBA 71

3.4.1 Algoritmo SAMBA 73

3.4.2 Paralelização 73

3.5 *Arrays* FPGAs - *Splash*-2 74

3.6 *Arrays* FPGAs - HScan...76

3.6.1 Algoritmo Filtro...77

3.6.2 Arquitetura e Implementação do Filtro HScan...79

3.6.3 Performances do HScan...80

3.7 *Arrays* Programáveis VLSI - B_SYS 82

3.8 Arquiteturas Reconfiguráveis 82

3.8.1 NGEN (*eNGiNE*)..84

3.8.1.1 O *Hardware* do NGEN 84

3.8.1.2 Implementando Multiprocessadores com FPGAs 85

3.8.2 POLYP (*Parallel OnLine PolYmer Processing*)...86

3.8.3 Alinhamento de Seqüências sobre o *Cray* MTA-2...88

3.8.4 Estruturas Sistólicas para Algoritmos de Comparação de Seqüências.............89

3.8.4.1 Descrição da Implementação da Estrutura Sistólica para Algoritmos de Comparação de Seqüências 90

3.8.4.2 Performances da Estrutura Sistólica para Algoritmos de Comparação de Seqüências 91

CAPÍTULO 4 – ALGORITMOS DE ALINHAMENTO GLOBAL E LOCAL EM C 94

4.1 Considerações Iniciais 94

4.2 AGM - Algoritmo Global – Implementação Morony 95

4.3 ALM - Algoritmo Local – Implementação Morony....98

4.4 Performances dos Algoritmos Global e Local....101

4.5 Performances dos Algoritmos AGM, ALM e SAMBA 102

CAPÍTULO 5 – IMPLEMENTAÇÃO EM HARDWARE DOS ALGORITMOS DE SEQUENCIAMENTO 106

5.1 Considerações Iniciais....106

5.2 Operações e Funções dos Algoritmos Global e Local 108

5.3 Implementação dos Algoritmos Global e Local em Hardware 111

5.4 Descrição da Implementação em FPGA 112

5.4.1 AGM – Algoritmo Global em Hardware 113

5.4.2 ALM – Algoritmo Local em Hardware 115

5.4.3 Simulação do Algoritmo AGM Otimizado em Hardware....117

5.4.4 Recursos FPGAs dos Algoritmos 127

5.5 Comparação *Software* e *Hardware* 131

CAPÍTULO 6 – CONCLUSÃO 134

REFERÊNCIAS 138

APÊNDICE A 146

APÊNDICE B...153

APÊNDICE C...159

Capítulo

1

INTRODUÇÃO

Nesta sessão introdutória é feita uma síntese que engloba temas como bioinformática e a programação dinâmica voltada para os algoritmos de alinhamentos de seqüências genéticas, enfatizando os que são estudados neste projeto.

Destaca a tecnologia FPGA (*Field Programmable Gate Array*) [19] e a linguagem padrão para a descrição de *hardware* VHDL (***V****ery High Speed Integrated Circuit* ***H****ardware* ***D****escription* ***L****anguage*) [43][86], além de descrever a importância do desenvolvimento e aplicação dos algoritmos em bioinformática tais como os algoritmos de alinhamento global e local de seqüências genéticas a fim de obter o melhor alinhamento entre duas seqüências de DNA (*Dexoxyribonucleic Acid*) [54], que visa determinar o grau de similaridade entre as seqüências, ou a similaridade entre fragmentos destas seqüências.

1.1 Conceitos Básicos do Projeto

1.1.1 Bioinformática

Os recentes avanços nas áreas computacionais, tecnologia da informação e biologia molecular, aliados às poderosas técnicas estatísticas deram origem à bioinformática, uma nova disciplina que está em rápido crescimento e sendo desenvolvida para atender as necessidades de manipular e processar grandes quantidades de dados genéticos e bioquímicos e do gerenciamento e análise de informações biológicas. A bioinformática usa o poder computacional para catalogar, organizar e estruturar estas informações em uma entidade compreensiva e extremamente importante para a biologia [25].

Geralmente ela é referida como tendo a tarefa de organizar e analisar dados complexos resultantes de modernas técnicas de biologia molecular e bioquímica [74]. A bioinformática se baseia na premissa de que existe um relacionamento hierárquico entre as estruturas dos genes, seu arranjo em relação ao genoma, a função das proteínas e suas interações em um organismo resultando em energia, metabolismo, reprodução e forma [63].

A partir da análise genética é possível expressar as informações contidas nas seqüências de bases de um ou mais genes e compará-las às propriedades esperadas com aquelas apresentadas pelas proteínas em relação à forma e função. Dessa maneira, surge a necessidade de comparar seqüências genéticas para poder encontrar altas similaridades estruturais e consequentemente funcionais entre duas ou mais seqüências de DNA, RNA (*Ribonucleic Acid*) e de proteínas [54]. A similaridade é dada pela melhor pontuação do alinhamento dentre todos os possíveis entre as seqüências a serem comparadas e posteriormente alinhadas.

1.1.2 Comparação e Alinhamento de Seqüências Genéticas

Antes de adentrar no assunto sobre comparação e alinhamento de seqüências genéticas, é conveniente salientar como são compostas as bases do DNA.

O DNA é constituído por duas cadeias polinucleotídicas (seqüências repetitivas de nucleótidos ou bases azotadas) que adotam uma estrutura em **dupla hélice**. Estas duas cadeias associam-se através da formação de pontes de hidrogênio entre as bases azotadas constituintes: a **Guanina (G)** associa-se especificamente com a **Citosina (C)**; a **Adenina (A)** associa-se especificamente com a **Timina (T)**. Este emparelhamento de

bases fundamenta a **complementaridade** do DNA [57]. Este modelo estrutural do DNA implica que as duas cadeias estejam orientadas em sentidos opostos - **anti-paralelas,** conforme pode ser observado na figura 1.1.

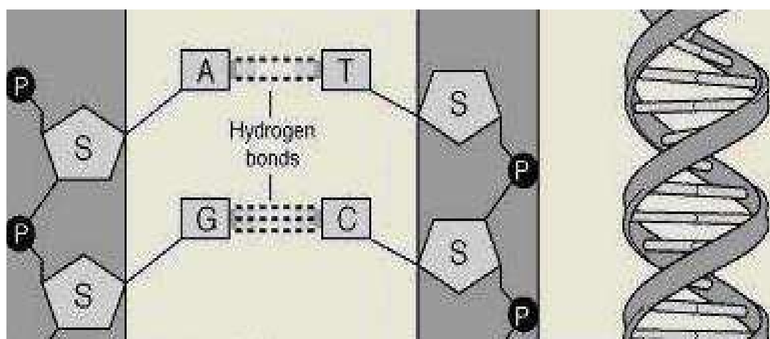


Figura 1.1 Modelo estrutural do DNA [90]

Uma seqüência ou *string* "**s**"é uma sucessão ordenada de caracteres de um alfabeto. Em bioinformática interessam seqüências compostas de caracteres que representam bases de DNA: **A**, **T**, **G**, **C**. Um exemplo de seqüência pode ser: **S = ACTGG**. Neste exemplo, o tamanho da seqüência **S** equivale ao número de caracteres, isto é, **| S | = 5**. Neste exemplo, caracteres aleatórios foram utilizados, mas poderiam ser quaisquer caracteres, com diferentes tamanhos. A seqüência é denominada vazia se **S =** **' '** ou **S = ε** ou então **| S | = 0,** usado neste caso, para exemplificar a seqüência **S**.

Atualmente, a comparação de seqüências é um importante problema de toda a biologia computacional, dado o número e diversidade de seqüências existentes e a freqüência com que precisa ser resolvido rotineiramente pelos biologistas moleculares, ou seja, é o *sorting* (classificação) da biologia computacional.

O objetivo da comparação de seqüências é estabelecer parâmetros de semelhança entre duas ou mais seqüências respeitando critérios específicos. Sua aplicação na biologia computacional é essencial.

As seqüências a serem comparadas não precisam ser exatamente iguais, ou terem subcadeias exatamente iguais; aceita-se certo número de "erros" de vários tipos no resultado. Segundo Setúbal [75], a comparação exata é menos relevante na biologia. Já na computação esse processo de comparação é mais conhecido como busca de padrões.

A dimensão das seqüências e a complexidade dos algoritmos que as processam são em determinadas aplicações fatores limitantes. Um exemplo desta limitação é a aplicação destas estratégias para buscas em grandes bases de dados. Algumas variantes

de métodos de comparação de seqüências tentam trabalhar com heurísticas[1] para minimizar o tempo de busca [57].

O alinhamento de seqüências genéticas pode ser definido como a maneira de comparar duas ou mais seqüências por meio de buscas de uma série de caracteres ou padrões de caracteres que não precisam estar na mesma ordem. O alinhamento é muito útil na predição de função, estrutura e inferência filogenética [57].

A figura 1.2 descreve um exemplo bastante simples de um alinhamento de seqüência de DNA, no qual a seqüência 1 é definida por **A T C G A A G C T** e a seqüência 2 por **T C C A _ C G C _**.

Posição	1 2 3 4 5 6 7 8 9
Seqüência 1 ⟶	**A T C G A A G C T**
	| | |
Seqüência 2 ⟶	**T C C A _ C G C _**

Figura 1.2 – Exemplo de alinhamento entre duas seqüências de DNA

Neste exemplo, o pareamento de bases de DNA acontece apenas em três lugares, local ilustrado com um traço na vertical (da esquerda para a direita), ou seja, o pareamento do terceiro caracter C que está na posição 3, o mesmo acontece para o pareamento do caracter G que se encontra na posição 7 e finalmente o pareamento do caracter C estando na posição 8. Na seqüência 2, pode-se observar que existem dois *gaps* (estando nas posições 5 e 9) que são representados pelos *underlines* de modo a proibir explicitamente uma correspondência ou simplesmente para completar posições faltantes para conseguir obter o melhor alinhamento entre as seqüências.

Em síntese, o alinhamento é uma maneira de colocar a seqüência 1 sobre a seqüência 2 de modo a estabelecer uma correspondência entre as bases da seqüência 1 e as bases da seqüência 2, ou seja, o pareamento do alinhamento entre as duas seqüências de DNA.

Em geral, segundo Setúbal [75], num alinhamento entre duas seqüências, o pareamento de caracteres distintos é chamado de substituição; o pareamento de um caracter com um *gap* é chamado de inserção ou remoção, dependendo da seqüência de referência.

---

[1] Heurística: Conjunto de regras e métodos que conduzem à descoberta, à invenção e à resolução de problemas.

1.1.3 Métodos de Alinhamento de Seqüências

O alinhamento de seqüências genéticas é similar a outros tipos de análise comparativa, onde este envolve a atribuição de valores de semelhança e diferenças entre um grupo de entidades relacionadas. Isto é, inferir relações estrutural/funcional/evolucionária [91].

Segundo Cristino [15], os principais métodos de alinhamento de seqüências genéticas dividem-se em duas categorias:

1) Alinhamento de pares de seqüências; e
2) Alinhamento de múltiplas seqüências.

A primeira categoria, alinhamento de pares de seqüências é composta dos seguintes métodos:

- **Matriz de pontos:** utilizada para comparar duas seqüências genéticas buscando possíveis alinhamentos de caracteres entre as seqüências, através da comparação de seqüências genômicas pareadas e seqüências repetitivas e inversões;
- **Programação Dinâmica:** é uma técnica de projeto de algoritmos, cujo método computacional calcula o melhor alinhamento possível entre as seqüências genéticas. É um caso especial da técnica de indução: resolve-se o problema utilizando a solução de problemas menores, que tem como objetivo guardar todas as subsoluções numa tabela (matriz) e determinar quais subsoluções são necessárias para a solução global, tendo que existir essa tabela (matriz) para que não seja preciso re-calcular as subsoluções [70]; e
- **Dicionário de palavras ou k-tuplas (BLAST)**: O *software* BLAST (*Basic Local Alignment Search Tool*) [1] é atualmente o método mais utilizado para realizar buscas de seqüências similares em bancos de dados de seqüências. Este *software* não faz uso da programação dinâmica e sim de heurísticas e sua comparação é local. Suas implementações mais conhecidas são a do NCBI (*National Center for Biotechnology Information*) [55] e o da Universidade de *Washington*, conhecido como WU-BLAST [55].

Entretanto, a segunda categoria, chamada de alinhamento de múltiplas seqüências possui o grande problema de alinhar simultaneamente mais de duas

seqüências e é também importante porque existe uma necessidade muito ampla de definir famílias de seqüências (principalmente de proteínas), sendo que essa definição permite caracterizar a família, procurar novos membros da família e entender a estrutura, a evolução e a função da família. A comparação de duas seqüências não é suficiente para caracterizar uma família, pois as partes comuns a todos os membros da família podem estar espalhadas no meio de outras partes que são comuns a apenas dois membros [95].

Portanto, o método de alinhamento de seqüências que será utilizada neste projeto é a programação dinâmica, porque permite calcular o melhor alinhamento possível entre as seqüências genéticas de DNA.

1.1.4 Programação Dinâmica para o Alinhamento Global e Local

A técnica de programação dinâmica aplicada nos algoritmos de alinhamento global, conhecido como algoritmo de *Needleman-Wunsch* [56] tem como finalidade a similaridade global entre duas seqüências de DNA, onde todas as bases são alinhadas umas com as outras ou com *gaps*. Isto é, os valores globais requerem que o alinhamento comece no início e se estenda até o final de todo o comprimento da seqüência [91].

O algoritmo de alinhamento local, como o algoritmo de *Smith-Waterman* [80], possui similaridade local entre duas seqüências, não necessita alinhar todas as bases em todas as seqüências, ou seja, os valores locais requerem identificação na região mais similar entre duas seqüências e também a penalidade por falta atribuída ou não [91].

Com uma pequena mudança da técnica de programação dinâmica aplicada no alinhamento global torna-se local. Assim, é visível a importância dos algoritmos de alinhamento de seqüências na bioinformática, pois ao encontrar o alinhamento máximo será possível detectar altas similaridades que existem entre as seqüências de maneira a permitir encontrar similaridades funcionais e consequentemente estruturais, contribuindo desta forma para os avanços nas pesquisas dessas áreas.

Conhecendo que os algoritmos de alinhamento global e local de seqüências genéticas são de extrema importância para a comunidade acadêmica na área da bioinformática, logo, pretende-se neste projeto implementar em *hardware*, dispondo da tecnologia FPGA, os algoritmos que são considerados ou adotados como padrão para a comparação e o alinhamento global e local das seqüências genéticas de DNA utilizando a técnica de programação dinâmica, uma vez que estes estão sendo amplamente

utilizados para novas pesquisas, através das implementações de novas ferramentas tanto em nível de *software* quanto de *hardware* como são mencionados em vários artigos [26][42][76][95].

1.1.5 A Importância do Desenvolvimento dos Algoritmos de Alinhamento de Seqüências em Bioinformática

Com o desenvolvimento de novos algoritmos e estatísticas na bioinformática para avaliar as relações em grande conjunto de dados, tais como, métodos de localização de genes numa sucessão de seqüências, predição de estrutura/função de proteínas e agrupamentos de seqüências de proteínas e famílias; a área da bioinformática necessita de desenvolvimento e implementação de ferramentas eficientes que possibilitam usar e administrar vários tipos de informações, ou seja, a utilização de instrumentos computacionais como *software* e *hardware,* para realizar de maneira mais eficiente e em menor tempo as análises dos algoritmos de alinhamento de seqüências são justificáveis, visto que, em grande parte destas aplicações o tempo e o resultado são variáveis fundamentais [57].

Por isso, a grande importância no desenvolvimento de algoritmos para realizar o processo de alinhamento de seqüências genéticas deve-se a três grandes fatores, sendo eles [91]:

1. Encontrar seqüências similares ajuda a determinar propriedades e inferir funções da nova seqüência, devendo ser verificada experimentalmente;
2. Posições conservadas (a similaridade entre elas é muito grande) em seqüências homólogas (são aqueles que descendem de um mesmo gene ancestral) sugerem sítios funcionalmente importantes (sítios ativos ou catalíticos, domínios de ligação de DNA, entre outros), ou seja, nesse caso, os alinhamentos locais são mais adequados para determinar quando dois genes são realmente homólogos, pois conseguem detectar as regiões mais conservadas; e
3. Nucleotídios conservados podem indicar elementos regulatórios que podem ser pré ou pós transcricionais [91].

Nos dias atuais, os bancos de dados constituem-se em ferramenta de trabalho essencial para biologistas moleculares, porque permitem a proliferação, o

compartilhamento de grande quantidade de informações contidas em bancos de dados centralizados, a comunicação entre os pesquisadores a redução da repetição de trabalhos em várias áreas da pesquisa científica em todo o mundo.

Baseado na observação de que genes ou proteínas com seqüências similares ou com regiões similares têm grande chance de possuírem funções similares, as primeiras informações para determinação da função de um gene, cuja seqüência foi recentemente obtida, quase sempre são obtidas pela busca de similaridades entre a nova seqüência e seqüências de proteínas ou famílias de proteínas conhecidas [1]. Entretanto, para que essa tarefa de busca de seqüências similares possa ser efetivamente realizada é necessário que os biologistas moleculares tenham à sua disposição uma ferramenta computacional que os auxiliem.

Entretanto, isso levou ao desenvolvimento de métodos heurísticos para esta tarefa, tais como BLAST [1] e FASTA [2]. Por outro lado, o algoritmo acima mencionado apesar de ter custo computacional elevado, é ainda muito utilizado nas publicações de artigos recentes, exemplo disso, o artigo internacional [58].

Portanto, os biologistas moleculares precisam de *hardware* e *software* com alta performance para poder detectar similaridades funcionais. A tecnologia que tem sido empregada para conseguir essa performance é *hardware* reconfigurável ou sistemas de *hardware* dedicados.

1.1.6 FPGA e VHDL

O *Field Programmable Gate Array* (FPGA) [19][93] é um circuito programável composto por um conjunto de células lógicas ou blocos lógicos alocados em forma de uma matriz que podem ser reconfigurados pelos projetistas quando necessário, sendo ideais para o desenvolvimento rápido de protótipos de sistemas digitais. Na realidade, um mesmo *chip* pode assumir arquiteturas e funções completamente diferentes, sem a necessidade de mudanças do mesmo, em função apenas da necessidade do usuário.

São muitas as vantagens da importância da utilização da tecnologia na área biológica, pois, os FPGAs possuem dois grandes campos de utilização: o rápido desenvolvimento de protótipos de circuitos (*Rapid Prototyping*) [10] e a possibilidade de reprogramação no próprio circuito acarretam uma diminuição significativa no tempo

de desenvolvimento devido à agilidade no processo de simulação/teste/depuração/alteração do projeto.

Nesta dissertação tem-se como objetivo implementar em *hardware*, dispondo da tecnologia FPGA, dos algoritmos que são considerados ou adotados como padrão para a comparação e o alinhamento global e local das seqüências genéticas de DNA utilizando a técnica de programação dinâmica. Esta implementação é descrita e modelada em alto nível de abstração usando-se a linguagem de descrição de *hardware* VHDL (***V****ery High Speed Integrated Circuit* ***H****ardware* ***D****escription* ***L****anguage*) [86].

1.2 Trabalhos Correlatos

Dentre as arquiteturas que são mencionadas como trabalhos correlatos, considera-se importante mencionar alguns projetos de arquiteturas voltadas para o trabalho de análise de seqüências genéticas, justamente pela importância que os sistemas de *hardware* dedicados estão tendo nas pesquisas atuais.

Recentemente, soluções usando FPGAs têm sido desenvolvidas para contribuir no avanço da bioinformática, com especial atenção às pesquisas genômicas que fazem uso maciçamente dos algoritmos de comparação e análise de seqüências de DNA para tentar encontrar maiores similaridades funcionais entre as seqüências e algumas soluções através da tecnologia de *hardware* reconfigurável, especificamente os FPGAs.

Alguns desses trabalhos são: PAM [42] e *Splash* [30], *Bioccelerator* [42] [12] e o *Decypher* II [42], MGAP (*Micro Grain Arithmetic Processor*) [9].

Esses sistemas têm sido muito usados pela comunidade acadêmica da área de bioinformática, uma vez que os biólogos necessitam de algoritmos de comparação e alinhamento que não são fixos e soluções paralelas programáveis são exigidas para agilizar essas tarefas.

Dessa maneira, os FPGAs parecem ser uma alternativa para resolver o problema da alta complexidade dos algoritmos em bioinformática, como por exemplo dos algoritmos usados para fazer a comparação e o alinhamento de seqüências genéticas, pois os mesmos trabalham com grandes seqüências exigindo altas tecnologias de ponta, tanto em nível de *software* quanto de *hardware*, para obter desempenhos satisfatórios para processar essas enormes quantidades de informações.

Não obstante, tem-se desenvolvido estratégias de solução tanto em nível de *software* quanto de *hardware* para que a bioinformática possa aumentar sua capacidade

de análise fina, de modificação de fenômenos biológicos e também para que sejam desenvolvidas tecnologias aplicáveis à evolução desta área que têm grande importância para a vida humana. Algumas soluções em nível de *hardware* podem ser tais como: *Cray* MTA (*Multithreaded Architecture*) – 2 [8], B-SYS [39], *Splash* – 2 [3][37], *Kestrel* [35] e posteriormente em nível de *software* podem ser citados: BLAST [1], FASTA [18], *Cross_Match* [24], *Clustal* W [75].

Apresenta-se a seguir alguns projetos que tiveram implementação dos algoritmos de análise de seqüências de DNA, RNA e/ou proteínas, cujos resultados de performances foram comparados a outras arquiteturas propostas por outros autores.

Além dos trabalhos que focalizam arquiteturas específicas para resolver os problemas da complexidade dos algoritmos de alinhamento de seqüências genéticas, existem também grupos de pesquisas que estão realizando propostas direcionadas ao desenvolvimento de *softwares* e plataformas. Alguns exemplos representativos nesse último foco são descritos a seguir:

- Cross_Match [24]: é o *software* mais utilizado para a retirada de regiões de vetores das seqüências de DNA. Foi desenvolvido para realizar comparações entre seqüências de DNA, mas tem uma opção que permite mascarar, em uma seqüência, o que for encontrado de similar com uma outra. O *cross_match* é fornecido gratuitamente para uso acadêmico e está disponível em: www.phrap.org/consed/academic_agreement.txt ;

- Cray MTA (Multithreaded Architecture) – 2 [8]: algumas das variações do algoritmo de programação dinâmica foram implementadas na arquitetura *Cray* MTA – 2, onde concentrou-se em analisar as implementações referentes às atualizações sob a forma anti-diagonal com a utilização de um *bit* de sincronização; e

- Kestrel [35]: é um co-processador programável projetado para análise de seqüências. Este co-processador inclui palavras de 8 *bits*, uma instrução de ciclo único "adiciona e minimiza" e uma comunicação eficiente usando registradores compartilhados sistólicos. *Kestrel* é um projeto para desenvolver um *array* sistólico linear programável para análise de seqüências [36]. O projeto *Kestrel* visa desenvolver uma plataforma bem adequada para análise de seqüências biológicas e também fornecer uma arquitetura programável, que está relacionada com a meta acima mencionada.

A tabela 1.1 ilustra brevemente alguns sistemas de *hardware* que estão direcionados às análises de seqüências genéticas, os quais estão sendo usados pela comunidade acadêmica da área de bioinformática.

Tabela 1.1 Sistemas de *hardware* destinados à solução da bioinformática

Sistemas de *Hardware*	**Funcionalidade/Descrição**
Bioccelerator [42]	É uma máquina comercial que usa tecnologia FPGA da *Xilinx* para análise de seqüências. URL: (http://sgbcd.weizmann.ac.il/).
BioScan [78]	É uma nova ferramenta computacional que emprega múltiplos níveis de paralelismo para buscar rapidamente, sensivelmente e rigorosamente todos os alinhamentos possíveis sem *gaps* em bancos de dados.
BISP [14]	É um sistema que faz a implementação em *hardware* da programação dinâmica total com *gaps*. Pode-se implementar a programação dinâmica com *gap penalties*.
B-SYS [39]	É um *array* sistólico de propósito geral que, não obstante otimizado para uma variedade de algoritmo de comparação de seqüências pode implementar a programação dinâmica com *gap penalties*.
Splash e PAM [42]	São arquiteturas FPGAs que fornecem uma implementação dos algoritmos de comparação de seqüências em *hardware* sem a necessidade de fabricar novos *chips* para cada algoritmo.
MGAP [9]	Tem sua própria arquitetura reconfigurável com um *mesh* de 64x64 células da comparação de seqüência no *chip* original.
Kestrel [35]	É um co-processador programável projetado para análise de seqüências.
PIM [22]	Co-processador programável formado como um *array* linear de *bit*-serial de elementos de processamento e memória.
Decypher II [42]	É uma máquina comercial que usa tecnologia FPGA da *Xilinx* para análise de seqüências. URL: (http://timelogic.com/).
SAMBA [28]	É um *array* sistólico dedicado para agilizar o processo de comparação de seqüências. Composto de 128 processadores de 12 *bits*.
Fast Data Finder [60]	Destinado primeiramente a busca de textos, mas tem sido adaptado para análise de seqüências, e um servidor está disponível para o perfil de busca.
Uma estrutura sistólica para algoritmos de comparação de seqüências baseados em programação dinâmica [11].	Dissertação de Mestrado em Ciência da Computação – Departamento de Ciência da Computação – Universidade de Brasília. Esta dissertação teve como objetivo propor, implementar e validar uma solução paralela baseada em *hardware* comparar seqüências utilizando o algoritmo de *Smith-Waterman* [80]; realizar comparações de desempenho com outras implementações do algoritmo, seqüenciais e paralelas. Para o desenvolvimento do projeto foi utilizada a placa APEX PCI (*Peripherical Component Interconnect*) *Development Board* do fabricante *Altera*, contendo o FPGA APEX EP20K400EFC672. Esta placa foi instalada em um computador da *Dell* com um processador *Pentium* IV e a ferramenta de síntese adotada foi o *Quartus* II da própria *Altera*.

Portanto, pode-se dizer que alguns sistemas de *hardware* que utilizam a tecnologia FPGA são: PAM e SPLASH. As mais recentes são o *Bioaccelerator* e

DeCypher II que são máquinas baseadas na tecnologia FGPA da *Xilinx* específicas para análise de seqüências, uma vez que os FPGAs oferecem poder computacional que os transformam em dispositivos mais adequados para aplicações que envolvem algoritmos onde uma rápida adaptação de entrada é requerida.

1.3 Objetivos da Dissertação

O principal objetivo da dissertação é implementar em *hardware*, dispondo da tecnologia FPGA, os algoritmos que são considerados ou adotados como padrão para a comparação e o alinhamento global e local das seqüências genéticas de DNA utilizando a técnica de programação dinâmica.

Em particular, são implementados os algoritmos de alinhamento global (*Needleman-Wunsch*) e local (*Smith-Waterman*), os quais são considerados de extrema importância pela comunidade acadêmica na área de bioinformática.

Considerações sobre o projeto e o desempenho da respectiva implementação em FPGAs são realizadas, assim como otimizações em nível arquitetural e de programação em VHDL, visando obter bons resultados. Será realizada uma comparação e respectiva análise com outras soluções já existentes em *hardware*, em especial, aquelas baseadas em FPGAs.

1.4 Justificativa

Os algoritmos a serem estudados e atualmente utilizados na biologia computacional são de extrema importância porque nesses últimos anos, grupos de pesquisadores em universidades e instituições de pesquisa bem como diversas outras empresas privadas, grandes e pequenas, estão envolvidos em diversos projetos do seqüenciamento genético, como o projeto genoma humano e de outros organismos.

Em projetos como o genoma humano, os cientistas precisam analisar grandes bancos de dados contendo bilhões de caracteres de DNA, de proteínas, onde muitos dos algoritmos usados nessas análises exigem uma varredura de grandes segmentos nos bancos de dados biológicos. Por isso a necessidade de se desenvolver algoritmos e técnicas com o melhor desempenho possível utilizando uma tecnologia compatível que permita conseguir ótimos resultados nas análises pesquisadas.

Por esse motivo, escolheu-se esse assunto, pois mesmo existindo algoritmos de programação dinâmica para o problema da comparação de seqüências genéticas, esses

algoritmos são amplamente usados pela comunidade acadêmica da área biológica na tentativa de encontrar maiores soluções para os desafios que estão sempre surgindo nesta área. Não obstante o tempo para fazer a varredura nos bancos de dados na tentativa de encontrar rápidas soluções ainda é muito alto, uma vez que os mesmos estão crescendo rapidamente.

Então, pode-se pensar na possibilidade de implementação dos algoritmos de alinhamento global e local usando a tecnologia FPGA com a finalidade de amenizar os problemas de desempenho dos mesmos, com intuito de analisar mais detalhadamente as partes mais críticas. Visto que muitos programas hoje se encontram disponíveis para fazer a comparação de seqüências biológicas, não possuem alto custo computacional, como é o caso do BLAST [1], que não utiliza a programação dinâmica e sim heurísticas e que fazem somente o alinhamento local das seqüências genéticas, porém, não têm garantia de que vai encontrar o melhor alinhamento [1].

Como a maior parte dos algoritmos de seqüências biológicas está relacionada com a análise de seqüências genéticas, a comparação de seqüências é uma tarefa de extrema importância na bioinformática. Por isso, a grande importância de dispor de tecnologias alternativas tanto em nível de *hardware* quanto de *software* para tornar os algoritmos para a comparação e alinhamento de seqüências biológicas mais eficientes, faz com que o grande desafio desta área seja desenvolver análises rápidas com tempo de resposta cada vez menor.

Este projeto não tem como objetivo desenvolver uma plataforma paralela para conseguir mais desempenho dos algoritmos, mas sim em saber quais resultados serão obtidos com a implementação dos algoritmos global e local, utilizando a tecnologia FPGA na resolução dos problemas destes no alinhamento de seqüências genéticas.

Portanto, sabe-se que as pesquisas e os estudos biológicos caminham numa rápida evolução, logo, esta área precisa de tecnologia que realmente consiga acompanhar seu enorme desenvolvimento. Diante desse fato, possivelmente, pesquisadores, biólogos e projetistas de *hardware* foram motivados a estudar e averiguar as funcionalidades das arquiteturas reconfiguráveis utilizando a tecnologia FPGA.

1.5 Organização da Dissertação do Mestrado

A dissertação é composta de 6 capítulos, descritos a seguir:

Capítulo 1: Introdução, onde se descreve a importância do assunto, os objetivos da dissertação e a respectiva justificativa.

No capítulo 2, métodos e algoritmos de alinhamento de seqüências, os principais métodos e algoritmos de alinhamento de seqüências genéticas são apresentados de maneira a explicar sua metodologia e seus funcionamentos.

O capítulo 3, ilustra alguns modelos de arquiteturas reconfiguráveis usados na bioinformática.

Já o capítulo 4, apresenta a implementação dos algoritmos de alinhamento global e local na linguagem de programação C e suas respectivas performances.

O capítulo 5, apresenta a implementação, testes e respectivas considerações de projeto e análise de desempenho, assim como proposta e realização de otimizações.

Finalmente, o capítulo 6, apresenta as conclusões, dificuldades encontradas e sugestões para trabalhos futuros. Por último, aparecem os apêndices e as referências bibliográficas.

Capítulo

2

ALINHAMENTO DE SEQÜÊNCIAS

2.1 Considerações Iniciais

Com os avanços da engenharia genética, na tentativa de estudar e mapear o genoma dos seres vivos e, em especial, o do homem, dados de seqüências genéticas são gerados em taxas cada vez maiores. Profissionais como biólogos deparam-se com uma enorme vazão de dados que gostariam de classificar e comparar com aqueles armazenados nos diversos bancos de dados.

A análise de uma seqüência envolve a procura nos bancos de dados, necessitando de algoritmos rápidos de comparação. A disponibilidade de sequências genômicas oferece uma oportunidade sem precedentes de explorar a variabilidade genética quer no mesmo organismo, quer entre organismos diferentes.

As seqüências genômicas podem ser de grande feito tecnológico, mas não são particularmente interessantes se não vierem acompanhadas de tentativas de usar esse conhecimento na prática.

Sabe-se que as centenas ou milhares de genes e seus produtos como RNA (ácido ribonucléico) e proteínas, funcionam de forma ordenada num organismo vivo de modo a manter as suas células em funcionamento. A compilação ordenada das seqüências genéticas correspondentes a toda a informação genética do genoma de uma espécie.

Este capítulo apresenta uma explanação sobre a comparação, o alinhamento, os principais métodos de alinhar as seqüências genéticas e seus respectivos algoritmos.

2.1.1 Comparação de Seqüências Genéticas

Antes de adentrar no assunto de comparação de seqüências genéticas é útil lembrar algumas definições que estão relacionadas ao processo de comparação [66].

1) **Seqüência ou *String***: sucessão ordenada de caracteres de um alfabeto, por exemplo, **s** = ACTGG, **p** = VALIDEFAC.

2) **Tamanho de uma seqüência**: número de caracteres que a seqüência possui, por exemplo, $|s| = 5$, $|p| = 9$, indicam seqüências de tamanho 5 e 9.

3) **Seqüência Vazia**: seqüência que não possui caracter, por exemplo, u = ' ' ou $u = \varepsilon$, $|u| = 0$.

4) ***Substring* de uma seqüência**: caracteres consecutivos e na mesma ordem da seqüência **t** = CGC é *substring* de **s** = AATCGCA.

- A seqüência **s** é uma **superstring** da seqüência **t**.
- A *string* vazia é *substring* de toda seqüência.

5) **Subseqüência de uma seqüência**: caracteres na mesma ordem da seqüência **t** = TA é subseqüência de **s** = AATCGCA,

- A seqüência **s** é uma **superseqüência** da seqüência **t**.

6) **Intervalo de uma *substring***: índices [i..j] indicam a posição na *string*.

7) **Concatenação de duas *strings*** (seqüências s e t):

s = AATTC e **t** = CGA,

st = AATTCCGA, **ts** = CGAAATTC e **| st | = | ts |=| s | + | t |**

8) **Prefixo de Seqüência**: *substring* s[1..j], $0 \leq j \leq |s|$

- s[1..0] denota *substring* vazia.

9) **Sufixo de Seqüência**: *substring* s[i.. | s |], $1 \leq i \leq |s| + 1$

- s[| s | +1 .. | s |] denota uma *substring* vazia.

Essas definições são usadas no decorrer de todo este capítulo a fim de fornecer uma melhor compreensão dos métodos e algoritmos de comparação e alinhamento de seqüências genéticas.

A comparação de seqüências genéticas tem como objetivo estabelecer parâmetros de semelhança entre duas ou mais seqüências respeitando critérios específicos. Sua aplicação na biologia computacional é essencial. A dimensão das seqüências e a complexidade dos algoritmos são em determinadas aplicações fatores limitantes. Um exemplo desta limitação é a aplicação destas estratégias para buscas em bases de dados [74].

Algumas variantes de métodos de comparação de seqüências tentam trabalhar com heurísticas para minimizar o tempo de busca. A utilização de instrumentos computacionais (*software* e *hardware*) para realizar estes processos é justificável visto que em grande parte destas aplicações, o tempo e o resultado são variáveis fundamentais.

No significado biológico do alinhamento de seqüências genéticas é importante definir três termos fundamentais [15]:

- **Identidade**: refere-se à presença do mesmo ácido nucléico (nt) ou aminoácidos (aa) na mesma posição em duas seqüências alinhadas;
- **Similaridade**: porcentagem de ácido nucléico ou de aminoácidos com propriedades químicas semelhantes; e
- **Homologia**: refere-se à relação evolutiva entre as seqüências. Duas seqüências homólogas derivam da mesma seqüência.

A comparação de seqüências é atualmente o problema mais importante de toda a biologia computacional, dado o número e a diversidade de seqüências existentes e dada a freqüência com que ele precisa ser resolvido diariamente pela comunidade acadêmica da área da bioinformática [75].

A importância deste problema vem do fato fundamental em seqüências de DNA, RNA e de proteínas, pois, grande similaridade implica em grande similaridade estrutural e consequentemente funcional, de modo a encontrar a similaridade entre duas ou mais seqüências ou *substrings* [66].

2.1.2 Alinhamento de Seqüências Genéticas

O alinhamento de seqüências genéticas se dá pela comparação de duas ou mais seqüências por meio de buscas de uma série de caracteres ou padrões de caracteres que estão na mesma ordem. O alinhamento é muito útil na predição de função, estrutura e inferência filogenética [75].

Por exemplo, pode-se dizer que a *string* **s** é "muito parecida" com a *string* **t**, e se sabe a função de **t**, então com grande probabilidade **s** terá a mesma função. Mas, é possível ter duas seqüências com a mesma função, porém nada similares [75].

Dadas duas seqüências de DNA **s** e **t**, de tamanhos | **s** | = n e | **t** | = m, e um sistema de pontuação para similaridade, onde pretende-se determinar o alinhamento das seqüências **s** e **t** de maior similaridade. Um alinhamento é uma maneira de colocar a seqüência **s** sobre a seqüência **t** de modo a estabelecer uma correspondência entre as bases de **s** e as bases de **t**.

Pode-se inserir espaços (*gaps*) em **s** e/ou em **t** de modo a explicitamente proibir uma correspondência ou simplesmente para completar posições faltantes entre as seqüências genéticas a serem alinhadas [75], conforme pode ser observada na figura 2.1.

Por exemplo, dadas duas seqüências genéticas **s** e **t**:

Posição	**1 2 3 4 5 6 7**
Seqüência s ⟶	C C G A T C T
	| | | |
Seqüência t ⟶	C A A G T C T

Figura 2.1 Exemplo de duas seqüências genéticas

Um sistema de pontuação para similaridade é uma maneira de atribuir notas a pareamentos de bases e de espaços. Na sua forma mais geral é uma matriz de elementos **P** (por exemplo, poderia ser outra letra) tal que **i, j Є {A, C, T, G -},** onde A (adenina), T (timina), C (citosina), G (guanina), **-** (representa o espaço - *gap*) e **p(i, j)** é um valor numérico, visto que a letra **i** e **j** representam as bases de DNA. Por exemplo, **p (A,A)** = +3 diz que A alinhado com A ganha 3 pontos.

Para o DNA é costumeiro usar um sistema bem mais simples. Por exemplo, **p(i,i)** = +1 e **p(i,j)** = -1 para **i**, **j** bases, e **p(i, -)** = **p(-, i)** = -2 para alinhamentos com espaço [75].

Em geral, num alinhamento entre duas seqüências, o pareamento de caracteres distintos é chamado de substituição. O pareamento de um caracter com espaço é

chamado de inserção ou remoção, dependendo da seqüência de referência. Na literatura, o conjunto de inserções e remoções recebe às vezes o nome de *indels* [75].

A equação 2.1 formaliza o conceito de similaridade que é dada pela pontuação do melhor alinhamento dentre todos os possíveis.

$$\mathbf{sim(s,t) = \max_{alin}(\sum_{k} p(s_k,t_k))} \qquad \text{Equação 2.1}$$

Onde k é um índice que varre todas as colunas do alinhamento (e por isso supõe-se que as seqüências **s** e **t** na verdade são suas versões "expandidas" com espaços, de modo a ter exatamente o mesmo tamanho).

Nas próximas seções são apresentados os principais métodos e algoritmos considerados ou adotados como padrão para a comparação e o alinhamento de seqüências genéticas que são utilizados pela comunidade acadêmica na bioinformática.

2.2 Métodos de Alinhamento de Seqüências Genéticas

O alinhamento de seqüências possui uma diversidade de aplicações na bioinformática, sendo considerada uma das operações mais importantes desta área. Este método de comparação procura determinar o grau de similaridade entre duas ou mais seqüências, ou a similaridade entre fragmentos destas seqüências. No caso de mais de duas seqüências o processo é denominado alinhamento múltiplo.

É importante ressaltar e lembrar que similaridade e homologia são conceitos diferentes. O alinhamento indica o grau de similaridade entre seqüências genéticas, já a homologia é uma hipótese de cunho evolutivo, e não possui gradação. Duas seqüências são homólogas caso derivem de um ancestral comum ou, caso esta hipótese não se comprove, simplesmente não são homólogas.

Existem vários *softwares* que realizam esta tarefa e a grande maioria deles pode ser utilizado *on-line*, sem a necessidade de instalação. Como exemplo tem-se os programas: *ClustalW* [66], *Multialin* [75], FASTA [2], BLAST 2 *sequences* [75], entre outros.

O processo (exemplo da figura 2.2) consiste em introduzir espaços (*gaps*) entre os monômeros de uma ou mais seqüências a fim de obter o melhor alinhamento possível. A qualidade de um alinhamento é determinada pela soma dos pontos obtidos

por cada unidade pareada (*match*) menos as penalidades pela introdução de *gaps* e posições não pareadas (*mismatch*) [69].

No caso exemplo da figura 2.1 é possível ter os seguintes resultados:

Match = posições: 1,2,3,6,7,8,10,11,14,16,18
Mismatch = posições: 4,15
Gap = posições: 5,9,12,13,17

Posição	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Seqüência s	K	K	A	K	-	T	T	H	H	G	A	-	-	V	W	N	-	G
Seqüência t	K	K	A	D	D	T	T	H	-	G	A	I	K	V	G	N	T	G

Figura 2.2 Alinhamento de duas seqüências de proteínas [67].

O alinhamento de seqüências genéticas é similar a outros tipos de análise comparativa, onde este envolve a atribuição de valores de semelhança e diferenças entre um grupo de entidades relacionadas. Isto é, inferir relações estrutural/funcional/evolucionária [91].

A figura 2.3 mostra uma abordagem da comparação de seqüências genéticas, onde se percebe que há três tipos de alinhamento, sendo estes detalhados posteriormente:

i) **Global**: o alinhamento se estende por toda a seqüência a ser comparada;

ii) **Local**: o alinhamento localiza fragmentos de seqüências que são mais similares, utilizando na comparação as *substrings* de DNA; e

iii) **Semi-local**: o alinhamento encontra a similaridade entre duas seqüências sem penalizar espaços no início e/ou no final do alinhamento, o qual faz a utilização de prefixos e/ou sufixos.

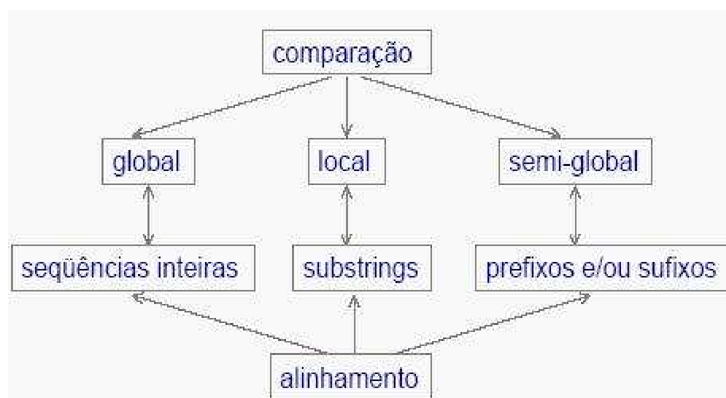


Figura 2.3 Comparação de seqüências genéticas e suas formas de alinhamento [66].

Os principais métodos de alinhamento de seqüências genéticas dividem-se em duas categorias [15]: Alinhamento de pares e de múltiplas seqüências.

- O alinhamento de pares de seqüências é composto pelos seguintes três métodos [15]:
    - **Matriz de Pontos**: Utilizada para comparar duas seqüências buscando possíveis alinhamentos de caracteres entre as seqüências, através da comparação de seqüências genômicas pareadas e seqüências repetitivas e inversões;
    - **Programação Dinâmica**: é um nome dado a uma técnica de projeto de algoritmos, cujo método computacional calcula o melhor alinhamento possível entre as seqüências. É um caso especial da técnica de indução, isto é, resolve-se o problema utilizando a solução de problemas menores, que tem como objetivo guardar todas as subsoluções numa tabela (matriz), e determinar quais sub-soluções são necessárias para a solução global tendo que existir essa tabela (matriz) para que não seja preciso re-calcular a sub-soluções.

      Dentro da programação dinâmica tem-se a vantagem de fazer uso dos dois algoritmos de alinhamento, ou seja, (i) o **global**, conhecido como algoritmo de *Needleman-Wunsch* [1]. Este alinhamento se estende por toda seqüência e (ii) **local** conhecido como o algoritmo de *Smith-Waterman* [80] que localiza fragmentos de seqüências que são mais similares; e
    - **Dicionário de Palavras ou K-Tuplas (BLAST)**: O *software* BLAST (*Basic Local Alginment Search Tool*) [2] é mais usado para comparar seqüências, ou seja, é usado basicamente para fazer buscas em banco de seqüências.

      A partir de uma seqüência tipo consulta (*query*) do usuário, tenta achar todas as seqüências do banco (*subjects*) que têm alinhamentos estaticamente significativos. O programa BLAST não faz uso da programação dinâmica e sim de heurísticas e a comparação é local. Muitos pesquisadores da área da biologia consideram o *software* BLAST muito mais econômico e viável para as pesquisas relacionadas à bioinformática, pois consideram a programação dinâmica muito custosa.

- Entretanto, a segunda categoria, chamada de alinhamento de múltiplas seqüências possui o grande problema de alinhar simultaneamente mais de duas seqüências e é também muito importante porque existe uma necessidade muito ampla de definir famílias de seqüências (principalmente de proteínas), sendo que essa definição permite caracterizar a família, procurar novos membros da família e entender a estrutura, a evolução e a função da família.

A comparação de duas seqüências não é suficiente para caracterizar uma família, pois as partes comuns a todos os membros da família podem estar espalhadas no meio de outras partes que são comuns a apenas dois membros [95].

2.2.1 Alinhamento por Matriz de Pontos (*Dot Plot* / *Dot Matrix*)

As matrizes de pontos são muito provavelmente as formulações mais antigas para comparar seqüências [50]. Elas quais são representações visuais das similaridades entre duas seqüências e também um grupo de métodos que visualmente comparam duas seqüências e examinam as regiões das próximas similaridades entre elas.

Existem essencialmente duas principais formas de realizar matriz de pontos (*dot plots*), tais como forma exata e blocos de identidade, sendo descritas a seguir:

• **A forma exata**: As seqüências a serem comparadas são arrumadas ao longo da matriz. A cada célula **(i,j)** da matriz associa-se um ponto se **i=j** ou se **i** e **j** se assemelham segundo um qualquer critério (isto é, do ponto de vista da matriz de *scores* escolhida).

Uma seqüência diagonal de pontos indica regiões onde as duas seqüências são semelhantes. Este gráfico é útil quando aplicado às seqüências de proteínas, uma vez que elas se codificam com um alfabeto de 20 letras. Para o DNA, o seu alfabeto de 4 letras (A, T, G, C) implica em ter gráficos muito carregados de pontos, fazendo com que sejam dificilmente perceptíveis.

A figura 2.4 ilustra a representação de uma matriz de pontos utilizando duas seqüências, ou seja, a seqüência 1: **ATGCGTCGTT** (horizontal) e a seqüência 2: **ATCCGCGAT** (vertical). Os pontos são colocados em cada lugar que houver um *match* entre as duas seqüências.

Logo, os trechos diagonais (indicados por linhas) são áreas de alinhamento. Importante ressaltar que pode surgir mais de um alinhamento.

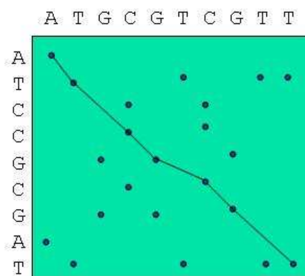


Figura 2.4 Matriz de pontos utilizando duas seqüências de DNA [72].

Existem métodos estatísticos que permitem analisar precisamente os resultados [20]. *Maizel* e *Lenk* popularizaram os *dot-plots* e sugeriram o uso de um filtro para reduzir o ruído provocado por *matches* aleatórios [50]. Muitos filtros são possíveis, mas o mais comum consiste em colocar um ponto na célula (i, j) se uma janela de 10 bases centrada em (i, j) contém mais de 6 *matches* positivos.

Outra forma de filtrar os resultados consiste em dar-lhes um peso de acordo com a sua semelhança química [82]. Independentemente do filtro, este método requer a construção de uma matriz MxN, e portanto cresce com o produto do comprimento das seqüências (O (N^2)). Isto acarreta um peso computacional conseqüente. Para dois genes com um comprimento médio (1.000 ácidos nucléicos), este método implica a construção de uma matriz com 106 células [20].

A figura 2.5 ilustra uma matriz de pontos MxN (M e N representam a quantidade de linhas e colunas da matriz de pontos), com a seqüência 1 = **ACCTGAGC** (horizontal) e a seqüência 2 = **ACCTGAGCT** (vertical). Isto é, uma matriz de pontos com 8x9.

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

Figura 2.5. Matriz de pontos compara duas seqüências de DNA [15].

Note que a matriz é colocada na intersecção da linha e da coluna com as mesmas bases de DNA. Os quadrados escuros representam o alinhamento entre seqüências idênticas na matriz de pontos.

- **Blocos de identidade**. Este método envolve "*hashing*" e em vez de ter em conta a matriz completa e calcular os pontos para cada célula da matriz, pode-se poupar custo computacional considerável se procurar apenas por *matches* exatos de certo comprimento. Este método procura unicamente blocos de identidade (semelhança) perfeita. A complexidade deste algoritmo cresce linearmente com N. O algoritmo simplesmente subdivide as duas seqüências em "palavras" de comprimento pré-especificado. Para cada seqüência, a localização de cada palavra é registrada. Estes vetores de "palavras" são então ordenados em paralelo com as palavras. Então, por comparação do vetor ordenado de uma seqüência com o da outra, obtém-se automaticamente as localizações de todas as "palavras" idênticas. As heurísticas de alinhamento na base de BLAST utilizam este método para selecionar as regiões de alinhamento mais promissoras.

Existem vários programas gratuitos que permitem realizar as matrizes de pontos. Dentre estes, sobressai especialmente o programa *dotter*, pela sua interatividade [81].

2.2.2 Alinhamento por Programação Dinâmica

Programação dinâmica é um nome dado a uma técnica de projeto de algoritmos, cujo método computacional calcula o melhor alinhamento possível entre as seqüências. É um caso especial da técnica de indução: resolve-se o problema utilizando a solução de problemas menores, que tem como objetivo guardar todas as subsoluções numa tabela (matriz), e determinar quais subsoluções são necessárias para a solução global tendo que existir essa tabela (matriz) para que não seja preciso re-calcular as subsoluções.

Dentro da programação dinâmica tem-se a vantagem de fazer uso de alguns algoritmos de alinhamento, tais como, o **global**, conhecido como algoritmo de *Needleman-Wunsch* [1], o **local** conhecido como o algoritmo de *Smith-Waterman* [80], e o **semi-local**.

Para que a técnica de programação dinâmica possa ser melhor compreendida nos algoritmos de alinhamento acima mencionados é importante que se conheçam os conceitos de edição de distância.

2.2.2.1 Definição de uma Distância de Edição

A distância de edição é uma métrica para se estabelecer uma comparação entre duas seqüências **S** e **T** (por exemplo), independente de suas respectivas cardinalidades.

O conceito de distância de edição e sua relação com o alinhamento de seqüências são de suma importância para a área da comunidade acadêmica da bioinformática [57]. Segundo Oliveira [57], a caracterização da distância de edição é a seguinte: Dadas as seqüências S e T, deve-se definir três operações de edição básicas (*Edit Steps*) sobre as seqüências, tais como:

1. **Operação de Inserção**: esta operação consiste em acrescentar ou inserir um caracter X em uma dada posição da seqüência S. Exemplo: **S = AGTC → S = AGTCX**;
2. **Operação de Deleção**: esta operação consiste em excluir ou deletar um caracter X em uma dada posição da seqüência S. Exemplo: **S = AGTC → S = AGT**; e
3. **Operação de Substituição:** esta operação consiste em substituir um caracter X presente em S por outro Y em uma dada posição da seqüência S. Exemplo: **S = AGTC, X = T e Y = A →** tem-se então **S = AGAC.**

Pode-se então associar a cada alinhamento possível um *score* igual ao número de operações de edição elementares efetuadas. O problema que se deve resolver é determinar o alinhamento "ótimo" de duas seqüências. O pesquisador *Levenshtein* introduziu em 1966 o conceito de distância de edição [49].

A distância **d(a,b)** entre duas seqüências **a** e **b** é definida como o número mínimo de operações elementares de edição necessárias para transformar **a** em **b**. O termo de distância é utilizado em matemática para definir uma função que verifica as seguintes propriedades métricas:

$d(a,b) \geq 0$ quaisquer que sejam **a** e **b**

$d(a,b) = 0$ se e apenas se $a = b$

$d(a,b) = d(b,a)$ quaisquer que sejam **a** e **b**

$d(a,b) + d(b,c) \geq d(a,c)$ quaisquer que sejam **a**, **b** e **c**

A procura do melhor alinhamento reduz-se assim o cálculo de uma distância. A definição precedente pode ser ligeiramente modificada, sem perturbar significativamente a sua generalidade, associando a cada operação elementar de edição um peso (isto é, um custo).

Se considerar que as operações elementares de edição tidas em conta (substituição ou inserção/deleção) representam diferenças elementares, o problema consiste em minimizar estas diferenças, ou seja, procurar o alinhamento de menor custo.

Para que esse menor custo seja encontrado é necessário ter um algoritmo que sirva como a base possível aplicar a programação dinâmica nos algoritmos de alinhamento global e local e semi-global.

2.2.2.2 Cálculo de uma Distância de Edição: o Algoritmo de Base

A distância de edição entre duas seqüências é calculada utilizando um algoritmo de programação dinâmica. A programação dinâmica é uma técnica fundamental de programação. É aplicável sempre que um grande espaço de procura pode ser estruturado numa sucessão de passos, de tal forma que [69]:

- O passo inicial contenha as soluções triviais dos subproblemas;
- Cada solução parcial num passo posterior possa ser calculada por recorrência a um número fixo de soluções parciais de passos anteriores; e
- O passo final contenha a solução global.

Portanto, neste caso utiliza-se a recursividade, isto é, supõe-se que o problema esteja resolvido até o passo **(i -1)** para resolvê-lo no passo **i**. Este algoritmo pode ser descrito da seguinte forma recursiva:

- Considerem-se duas seqüências **A** e **B** de comprimento **M** e **N** respectivamente; e **Ai** e **Bj** (**i** e **j** representam respectivamente os índices das linhas e colunas de uma dada matriz) são os símbolos correspondentes às posições **A0 ... Am** e **B0 ... Bn**. Seja **D(i,j)** a distância mínima entre as duas seqüências alinhadas do início **D(0,0)** até aos caracteres **An** e **Bm**; e
- Calcula-se sucessivamente as distâncias **D(i,j)** para os valores crescentes de **i** e **j**, até atingir o valor de **D(m,n)** que será a distância mínima entre as seqüências **A** e **B**. Os valores correspondentes são guardados numa tabela (matriz) com duas dimensões (**A** e **B**). O procedimento de preenchimento na

matriz inicia-se em **$D_{(0,0)}$ = 0**. O valor de uma célula **(i,j)** é definido a partir das três células precedentes que possuem distâncias **D(i -1,j)**, **D(i -1,j - 1)** e **D(i,j - 1)**.

Assim, calcula-se a distância mínima **D(i,j)** a partir da seguinte equação recursiva (ver equação 2.2), que faz uso de algumas expressões com seus respectivos significados [69].

- **w (Ai, 0)** corresponde ao custo associado à deleção do caracter **Ai**;
- **w (Ai, Bj)** corresponde ao custo associado à substituição de **Ai** por **Bj**; e
- **w (0, Bj)** corresponde ao custo associado à inserção do caracter **Bj.**

$$D(i,j) = \min \begin{cases} D(i-1, j) + w(Ai,0) \\ D(i-1, j-1) + w(Ai, Bj) \\ D(i, j-1) + w(0, Bj) \end{cases} \quad \text{Equação2.2 [69]}$$

Desta maneira, o alinhamento ótimo entre **Ai** e **Bj** é obtido seguindo os três itens a seguir, os quais permitem calcular o menor custo sendo realizado pelo cálculo da distância mínima.

1. Considera-se o alinhamento ótimo entre **A(i-1)** e **Bj** e prolonga-se este pela supressão do caracter **Ai**;

2. Considera-se o alinhamento ótimo entre **A(i-1)** e **B(j-1)** e prolonga-se este substituindo o caracter **Ai** pelo caracter **Bj**; e

3. Considera-se o alinhamento ótimo entre **Ai** e **B(j-1)** e prolonga-se este inserindo o caracter **Bj**.

Assim, calcula-se **D(i,j)** que faz uso de algumas expressões com seus respectivos significados, a qual pode ser observada na equação 2.3 [69].

- **ws (Ai,Bj)**: corresponde ao custo associado à substituição do caracter **Ai** por **Bj**;
- **wd (Ai)**: corresponde ao custo associado à deleção do caracter **Ai**; e
- **wi (Bj)**: corresponde ao custo associado à inserção do caracter **Bj.**

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + ws(Ai,Bj) \\ D(i-1,j) + wd(Ai) \\ D(i,j-1) + wi(Bj) \end{cases} \quad \text{Equação 2.3 [69]}$$

A figura 2.6 esquematiza o resultado do preenchimento de uma célula da matriz, que procede das três células precedentes:

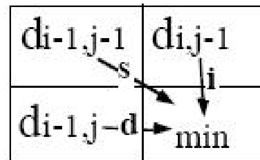


Figura 2.6 Preenchimento de uma célula de uma dada matriz [69].

A figura 2.7 ilustra um exemplo de como é possível obter a distância entre duas seqüências que apresentam caracteres escolhidos aleatoriamente [69], ou seja, a seqüência 1 contém os seguintes caracteres = **ELAGUEUR** (horizontal) e a seqüência 2 = **LARGEUR** (vertical). Assim, a matriz A é composta de 8x7 linhas e colunas respectivamente. Os valores correspondentes à inserção, deleção e substituição são todos iguais a 1.

Portanto, após ter preenchido a matriz A, pode-se observar que as setas (dentro da matriz) indicam duas maneiras diferentes (lado direito da matriz) as quais indicam a menor distância entre as duas seqüências comparadas. As setas que estão embaixo das seqüências alinhadas (direita da matriz) indicam apenas as inserções e deleções que essas duas seqüências sofreram.

Logo, o valor das menores distâncias obtidas com diferentes soluções encontradas na matriz A abaixo são iguais a 3.

A=

	0	E	L	A	G	U	E	U	R
0	0	1	2	3	4	5	6	7	8
L	1	1	1	2	3	4	5	6	7
	2	2	2	1	2	3	4	5	6
	3	3	3	2	2	3	4	5	5
G	4	4	4	3	2	3	4	5	6
E	5	4	5	4	3	3	3	4	5
U	6	5	5	5	4	3	4	3	4
R	7	6	6	6	5	4	4	4	3

distância = 3

```
ELA-GUEUR    { 1 inserção
-LARG-EUR    { 2 deleções
↑  ↑ ↑

ELAGUEUR     { 1 deleção
-LARGEUR     { 2 mismatch
↑  ↑↑
```

Figura 2.7 Representação da menor distância [69].

Needleman e *Wunsch* [56], e posteriormente *Sankoff* e *Sellers* [71][72][73], propuseram a aplicação deste algoritmo ao caso particular das seqüências biológicas. O método exato de *Needleman* e *Wunsch* é um pouco diferente daquele que foi enunciado

anteriormente, na medida em que eles puseram o problema em termos de maximização de semelhanças em vez de minimização de diferenças.

Na sua abordagem, o *score* correspondente ao melhor alinhamento é o maior de todos. Os dois métodos permitem alinhar de forma ótima duas seqüências (os algoritmos são frequentemente chamados de *Needleman-Wunsch* e *Smith-Waterman*). Pode-se mostrar que o *score* calculado a partir das minimizações das diferenças verifica as propriedades de uma distância.

Isto pode parecer menos evidente para o caso da implementação de *Needleman* e *Wunsch*. No entanto, *Smith* e *Waterman* [80] demonstraram que este pode ser definido de tal forma que o procedimento seja igualmente métrico. Em seguida, apresenta-se uma matriz definida de acordo com o algoritmo inicial de *Needleman-Wunsch* e *Smith-Waterman*.

2.3 Alinhamento Global utilizando a Técnica de Programação Dinâmica

Para o alinhamento global, deve ser utilizado o algoritmo de *Needleman-Wunsch* [1] que consiste em encontrar a similaridade global entre duas seqüências e os valores globais requerem que o alinhamento comece no início e se estende até o final de todo o comprimento.

Dadas duas seqüências inteiras, pretende-se encontrar o melhor alinhamento entre elas. Por exemplo, alinhar as seqüências **s = GATTCCG** e **t = GAATTCAG** [66], ver figura 2.8.

```
G A - T T C C G
G A A - T C A G
```

Figura 2.8 Comparação global de duas seqüências de DNA[66].

O alinhamento se dá pela inserção de espaços tais que as seqüências possuam o mesmo tamanho e que haja correspondência entre seus caracteres ou espaços. O espaço (*gap*) não pode ser alinhado com outro espaço.

Logo, a pontuação de um alinhamento ocorre pela soma das pontuações de suas colunas e o melhor alinhamento ou alinhamento ótimo é o alinhamento de maior pontuação.

O exemplo a seguir permite realizar o alinhamento entre duas seqüências de DNA, o qual utiliza a técnica de programação dinâmica. As seqüências 1 e 2 são

compostas de 11 e 7 caracteres de bases de DNA, as quais estão dispostas na matriz de programação dinâmica podendo ser observada na figura 2.9.

Posições	1 2 3 4 5 6 7 8 9 10 11
Seqüência #1 ⟶	**G A A T T C A G T T A**
Seqüência #2 ⟶	**G G A T C G A**

Figura 2.9 Exemplo de duas seqüências de DNA [15].

Assim, as seqüências #1 e #2 são compostas de 11 e 7 caracteres de DNA respectivamente, podendo ser representado por: M =11 e N = 7.

A programação dinâmica voltada para o alinhamento global é composta de três fases, sendo estas:

1. **Inicialização**: O primeiro passo da programação dinâmica no alinhamento global é criar uma matriz com M + 1 linhas e N + 1 colunas correspondentes ao tamanho da seqüência a ser alinhada. Nota-se na figura 2.10 que a primeira linha e primeira coluna da matriz não é preenchida por nenhum caracter de DNA, recebendo apenas o valor zero, (representado na figura por um traço) e por esse motivo a matriz contém uma linha e uma coluna a mais, isto é, M + 1 linhas e N + 1 colunas, onde a letra M (horizontal) representa as linhas e N (vertical) representa as colunas da matriz A.

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-												
1	**G**												
2	**G**												
3	**A**												
4	**T**												
5	**C**												
6	**G**												
7	**A**												

Figura 2.10 Inicialização da matriz de programação dinâmica para o alinhamento global [15].

Um simples sistema de *score* (pontuação) é assumido onde a matriz A é representada por valores que foram escolhidos aleatoriamente referentes aos itens 1, 2, 3 respectivamente; como é possível observar a seguir, cujos valores representados pelas letras **i** e **j** são referentes aos valores das linhas e colunas da matriz A [70].

a) **$A(a_i,b_j) = 5$** se o caracter de DNA (A, G, C, T) da matriz A da posição **a_i** da seqüência #1 for igual à da posição **b_j** da seqüência #2, chamado na bioinformática de *match*;

b) **$A(a_i,b_j) = -3$** se o caracter de DNA (A, G, C, T) da matriz A da posição **a_i** da seqüência #1 for diferente da posição **b_j** da seqüência #2, chamado na bioinformática de *mismatch*; e

c) **$w = -4$** (quando tiver *gap penalty* nas seqüências). Entende-se por *gap penalty* uma proposta de não penalizar um grupo de espaços consecutivos (buracos) e espaços isolados da mesma forma. Na comparação de seqüências biológicas isso não é desejável, pois não captura o fato de que durante a evolução molecular, blocos de bases são removidos ou inseridos num único evento mutacional [75].

Entretanto, é possível aplicar uma simples fórmula para fazer a inicialização da matriz A, para a primeira linha e primeira coluna:

$A_{i,0} = w * i$; (percorrendo a primeira linha do início ao fim); e

$A_{0,j} = w * j$ (percorrendo a primeira coluna do início ao fim).

A representação da matriz A com a primeira linha e coluna preenchida podem ser observadas na figura 2.11. O valor da variável **w** (caso haja *gaps* nas seqüências), já foi especificado no sistema de *score* descrito anteriormente.

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
1	**G**	-4											
2	**G**	-8											
3	**A**	-12											
4	**T**	-16											
5	**C**	-20											
6	**G**	-24											
7	**A**	-28											

Figura 2.11 Inicialização da primeira linha e coluna da matriz A [15]

2. **Preencher a matriz de programação dinâmica para obter o alinhamento global**: para cada posição da matriz **$A_{i,j}$** é definida a função MAX (máximo), representada na equação 2.4, a qual retornará o valor máximo obtido através da técnica de programação dinâmica que gerará o algoritmo de alinhamento global [56]. O valor da posição **$a(a_i,b_j)$** na

função MAX, retornará apenas o valor estipulado se for *match* ou *mismatch*, ou seja, se os caracteres de DNA são idênticos ou não:

$$\mathbf{A_{i,j} = MAX\ [}$$

$$\mathbf{A_{i-1,j-1} + a(a_i,b_j)}\ (\textit{match/mismatch}),$$
$$\mathbf{A_{i,j-1} + w}\ (\textit{gap}\ \text{seqüência}\ \#1),$$
$$\mathbf{A_{i-1,j} + w}\ (\textit{gap}\ \text{seqüência}\ \#2)$$
$$\mathbf{]}$$

Equação 2.4 [15]

Usando esta informação, a figura 2.11 ilustra a matriz A com o sistema de *score* na posição $\mathbf{A_{1,1}}$ já calculado, pois o primeiro valor das seqüências a serem alinhadas é igual a 1 (*match*) uma vez que as duas seqüências representam o caracter de DNA Guanina (G), e assumindo o estado como início, o *gap* será igual a **w = 0**.

Portanto, a figura 2.12 ilustra o preenchimento da matriz na posição da linha e coluna iguais a 1: $\mathbf{A_{1,1} = MAX\ [\ A_{0,0} + 5,\ A_{1,0} - 4,\ A_{0,1} - 4\ ]}$

$$\mathbf{= MAX\ [\ 5, -8, -8\ ]}$$
$$\mathbf{= 5.}$$

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
1	**G**	-4	5										
2	**G**	-8											
3	**A**	-12											
4	**T**	-16											
5	**C**	-20											
6	**G**	-24											
7	**A**	-28											

Figura 2.12 Preenchimento da posição $A_{1,1}$ da matriz A [15].

Observa-se na figura 2.13 o preenchimento da matriz A na posição da linha 1 e coluna 2, $\mathbf{A_{1,2}}$, cuja função MAX resulta em:

$$\mathbf{A_{1,2} = MAX\ [\ A_{0,1} -3,\ A_{1,1} - 4,\ A_{0,2} - 4\ ]}$$
$$\mathbf{= MAX\ [\ -4 - 3,\ 5 - 4,\ -8 - 4\ ]}$$
$$\mathbf{= MAX\ [\ -7,\ 1,\ -12\ ]}$$
$$\mathbf{= 1}$$

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
1	**G**	-4	5	1									
2	**G**	-8											
3	**A**	-12											
4	**T**	-16											
5	**C**	-20											
6	**G**	-24											
7	**A**	-28											

Figura 2.13 Preenchimento da posição $A_{1,2}$ da matriz A [15].

Prossegue-se desta forma até que a matriz esteja totalmente preenchida, finalizando assim a segunda fase do algoritmo. A figura 2.14 mostra o término da segunda fase da técnica de programação dinâmica voltado para o algoritmo de alinhamento global.

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
1	**G**	-4	5	1	-3	-7	-11	-15	-19	-23	-27	-31	-35
2	**G**	-8	1	2	-2	-6	-10	-14	-18	-14	-18	-22	-26
3	**A**	-12	-3	6	7	3	-1	-5	-9	-13	-17	-21	-17
4	**T**	-16	-7	2	3	12	8	4	0	-4	-8	-12	-16
5	**C**	-20	-11	-2	-1	8	9	13	9	5	1	-3	-7
6	**G**	-24	-15	-6	-5	4	5	9	10	14	10	6	2
7	**A**	-28	-19	-10	-1	0	1	5	14	10	11	7	11

Figura 2.14 Representação da Segunda fase – Matriz A preenchida [15].

3. *Traceback*: após ter preenchido a matriz A, dá-se início ao procedimento chamado de *traceback.* Este procedimento identifica os segmentos bem como produz o alinhamento correspondente que determinará qual é o resultado do sistema de pontuação máxima da matriz de programação dinâmica (matriz A), denominado de melhor alinhamento entre as duas seqüências ou alinhamento ótimo. A pontuação desse alinhamento, inicia-se na posição $\mathbf{A_{m,n}}$ da matriz A, a posição que conduz para o sistema de pontuação máximo, isto é, dá-se pelas somas das pontuações de suas colunas $\mathbf{A_{m,n}}$ até chegar à posição $\mathbf{A_{0,0}}$.

O *traceback* posiciona na célula atual da posição $A_{m,n}$ ($A_{11,7}$) cujo valor desta célula é igual a 11 e verifica as células vizinhas que poderiam ser suas predecessoras, ou seja, células que podem dar continuidade para obter o melhor alinhamento entre as duas seqüências da matriz A. Isso significa que a célula $A_{11,7}$ deve verificar o vizinho acima (*gap* na seqüência #1), o vizinho da diagonal (*match*/*mismatch*) e por último o seu vizinho da esquerda (*gap* na seqüência #2).

Portanto, verifica-se no máximo três possibilidades segundo uma ordem preferencial, conforme pode ser observado na figura 2.15, visto que, cada seta indica um par de alinhamento. Se houver mais de uma possibilidade, então existe mais de um alinhamento ótimo.

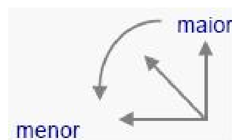


Figura 2.15 Representação da ordem preferencial da verificação do *traceback* [66].

Após o processo de *traceback* estar posicionado na célula $A_{11,7}$ da matriz A, o próximo passo é escolher a célula seguinte que poderá ser sua célula predecessora, tendo como objetivo dar continuidade a fim de obter o alinhamento ótimo.

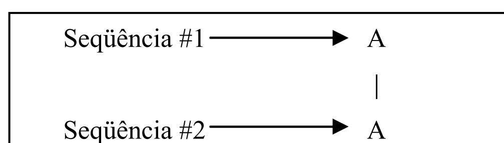
A matriz A representada pela figura 2.16 faz referência ao passo acima mencionado. As setas desta figura tem como o vizinho acima (*gap* na seqüência #1) a posição $A_{11,6}$ (valor 2) o vizinho da diagonal (*match*/*mismatch*) da posição $A_{10,6}$ (valor 6) e o vizinho da esquerda (*gap* na seqüência #2) com a posição $A_{10,7}$ (valor 7).

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
1	**G**	-4	5	1	-3	-7	-11	-15	-19	-23	-27	-31	-35
2	**G**	-8	1	2	-2	-6	-10	-14	-18	-14	-18	-22	-26
3	**A**	-12	-3	6	7	3	-1	-5	-9	-13	-17	-21	-17
4	**T**	-16	-7	2	3	12	8	4	0	-4	-8	-12	-16
5	**C**	-20	-11	-2	-1	8	9	13	9	5	1	-3	-7
6	**G**	-24	-15	-6	-5	4	5	9	10	14	10	6	2
7	**A**	-28	-19	-10	-1	0	1	5	14	10	11	7	11

Figura 2.16 Representação do passo inicial do processo de *traceback* [15].

Entretanto, a célula atual $A_{11,7}$ tem valor igual a 11 e desde que os *scores* definidos são: 5 para *match*, – 3 para *mismatch* e – 4 para *gaps*, existe apenas uma possível célula predecessora que é o vizinho da diagonal (*match/mismatch*), uma vez que a célula atual $A_{11,7}$ que representa a linha 11 e coluna 7 apresentam o caracter de DNA que é a Adenina (A), acontecendo desta forma um *match*. Isso permite o alinhamento atual das seqüências:



Se mais do que uma célula predecessora existe, qualquer uma pode ser escolhida [70]. Após ter encontrado a próxima célula predecessora, $A_{10,6}$, deve ser desconsiderada a linha 11 e a coluna 7 que representavam a célula anterior ao

alinhamento $A_{11,7}$ deixando a matriz A com a seguinte aparência, conforme a figura 2.17.

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	**G**	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	
1	**G**	-4	5	1	-3	-7	-11	-15	-19	-23	-27	-31	
2	**G**	-8	1	2	-2	-6	-10	-14	-18	-14	-18	-22	
3	**A**	-12	-3	6	7	3	-1	-5	-9	-13	-17	-21	
4	**T**	-16	-7	2	3	12	8	4	0	-4	-8	-12	
5	**C**	-20	-11	-2	-1	8	9	13	9	5	1	3	
6	**G**	-24	-15	-6	-5	4	5	9	10	14	10	6	
7	**A**												11

Figura 2.17 Matriz A após ter encontrado o primeiro alinhamento [70].

Nesta etapa, deve-se verificar a célula atual $A_{10,6}$ (com valor 6) e ver quais são suas células vizinhas (acima, diagonal e esquerda) que podem ser consideradas as possíveis células predecessoras para poder dar continuidade ao alinhamento.

O alinhamento produz um *gap* na seqüência #2, logo a célula atual é $A_{9,6}$, conforme o alinhamento seguinte:

```
Seqüência #1 ──────► T A
                     _ |
Seqüência #2 ──────►   A
```

A representação de acordo com o alinhamento anterior pode ser observado na figura 2.18.

A =

		G	**A**	**A**	**T**	**T**	**C**	**A**	**G**	**T**	**T**	**A**
	0	-4	-8	-12	-16	-20	-24	-28	-32	-36		
G	-4	5	1	-3	-7	-11	-15	-19	-23	-27		
G	-8	1	2	-2	-6	-10	-14	-18	-14	-18		
A	-12	-3	6	7	3	-1	-5	-9	-13	-17		
T	-16	-7	2	3	12	8	4	0	-4	-8		
C	-20	-11	-2	-1	8	9	13	9	5	1		
G	-24	-15	-6	-5	4	5	9	10	14	10	6	
A												11

Figura 2.18 Continuidade do processo de *traceback* [70].

Novamente, o predecessor direto produz um *gap* na seqüência #2. Logo, o alinhamento correspondente apresentado e a figura 2.19 ilustram o alinhamento atual.

```
Seqüência #1 ──────► T T A
                         |
Seqüência #2 ──────► _ _ A
```

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	G	A	A	T	T	C	A	G	T	T	A
0	-	0	-4	-8	-12	-16	-20	-24	-28	-32			
1	G	-4	5	1	-3	-7	-11	-15	-19	-23			
2	G	-8	1	2	-2	-6	-10	-14	-18	-14			
3	A	-12	-3	6	7	3	-1	-5	-9	-13			
4	T	-16	-7	2	3	12	8	4	0	-4			
5	C	-20	-11	-2	-1	8	9	13	9	5			
6	G	-24	-15	-6	-5	4	5	9	10	14	10	6	
7	A												11

Figura 2.19 Prosseguimento do processo de *traceback* [70].

Deve-se continuar com o processo de *traceback* até a posição da linha e coluna iguais a zero, $A_{0,0}$. Portanto, as figuras 2.20 e 2.21 ilustram a matriz A apresentando um possível alinhamento entre as duas seqüências de DNA.

A =

Posições		0	1	2	3	4	5	6	7	8	9	10	11
		-	G	A	A	T	T	C	A	G	T	T	A
0	-	0											
1	G		5										
2	G			2									
3	A				7	3							
4	T						8						
5	C							13	9				
6	G									14	10	6	
7	A												11

Figura 2.20 *Traceback* no alinhamento global [70].

```
G A A T T C A G T T A
|   |   | |   |     |
G G A - T C - G - - A
```

Figura 2.21 Possível alinhamento da matriz A [15][70].

O próximo passo é fazer a verificação do sistema de *score* aplicado anteriormente (figura 2.22), ou seja, 5 para *match*, - 3 para *mismatch* e – 4 para *gap penalty*. Para uma melhor representação, os valores dos *scores* são colocados embaixo do possível alinhamento obtido, cuja pontuação do alinhamento ótimo ficou com valor 11.

```
G A A T T C A G T T A
|   |   | |   |     |
G G A - T C - G - - A

+ - + - + + - + - - +
5 3 5 4 5 5 4 5 4 4 5

5 - 3 + 5 - 4 + 5 + 5 - 4 + 5 - 4 - 4 + 5 = 11 ✓
```

Figura 2.22 Sistema de *score* da matriz A [15].

Como foi mencionado anteriormente, o melhor alinhamento ou alinhamento ótimo é o de maior pontuação, que é dada pela soma das pontuações das colunas da

matriz em questão. Existem mais soluções alternativas (ver figura 2.23) resultantes do sistema de *score* do alinhamento máximo, que no exemplo utilizado foi 11 (valor da posição da matriz A).

A =

	-	G	A	A	T	T	C	A	G	T	T	A
-	0	-4	-8	-12	-16	-20	-24	-28	-32	-36	-40	-44
G	-4	5	1	-3	-7	-11	-15	-19	-23	-27	-31	-35
G	-8	1	2	-2	-6	-10	-14	-18	-14	-18	-22	-26
A	-12	3	6	7	3	-1	-5	-9	-13	-17	-21	-17
T	-16	7	2	3	12	8	4	0	-4	-8	-12	-16
C	-20	11	-2	-1	8	9	13	9	5	1	-3	-7
G	-24	-15	6	5	4	5	9	10	14	10	6	2
A	-28	-19	-10	-1	0	1	5	14	10	11	7	11

Figura 2.23 Diversas soluções alternativas da matriz A para o algoritmo de alinhamento global [15].

Logo, a similaridade não se altera se trocar uma dada seqüência **"s"** pela seqüência **"t"** em uma matriz, porém, o alinhamento ótimo pode ser diferente quando trocar a seqüência **"s"** pela seqüência **"t"** na matriz [66].

Esse é um problema exponencial, a maioria dos algoritmos de programação dinâmica imprimirá somente a saída de uma solução [70].

2.4 Alinhamento Local utilizando o Método de Programação Dinâmica

O alinhamento global é útil para comparar duas seqüências homólogas. Mas quando as duas seqüências apenas possuem certos domínios em comum, ou quando é necessário comparar uma seqüência com todas as entradas de uma base de dados, está-se mais interessado nos melhores alinhamentos locais entre duas subseqüências.

A programação dinâmica voltada para o alinhamento local é composta das três fases apresentadas no alinhamento global, ou seja, (1) **inicialização**, (2) **preenchimento da matriz de programação dinâmica para obter o alinhamento global** e (3) ***traceback***.

Porém, o alinhamento local possui algumas diferenças nos itens acima mencionados. Isto é, a matriz A de programação dinâmica para o alinhamento local no item 1, **inicialização**, (ver figura 2.9 utilizou o mesmo exemplo) deve ser inicializada com a primeira linha e a primeira coluna com valores zero para prevenir a similaridade de cálculos negativos (pois qualquer alinhamento dado pela primeira linha ou pela primeira coluna teria valor negativo no caso do alinhamento global) e o sistema de *score* (pontuação) é assumido onde a matriz A é representada pelos mesmos valores que

foram escolhidos aleatoriamente para o alinhamento global, os quais correspondem a *match* = 5, *mismatch* = - 3 e *gap* = - 4.

Na segunda fase, **preenchimento da matriz de programação dinâmica para obter o alinhamento local**, a função MAX que foi usada para calcular o valor de cada célula da matriz A (no alinhamento global) deve ser aplicada sobre os 3 termos habituais mais o zero, que pode ser observada na equação 2.5.

$$A_{i,j} = \mathrm{MAX}\left[\begin{array}{l} A_{i-1,j-1} + a(a_i,b_j)\ (\textit{match/mismatch}), \\ A_{i,j-1} + w\ (\textit{gap}\ \text{seqüência}\ \#1), \\ A_{i-1,j} + w\ (\textit{gap}\ \text{seqüência}\ \#2), \\ 0 \end{array}\right] \qquad \text{Equação 2.5 [15]}$$

Por último, a terceira fase, ***traceback***, inicia-se através do par de segmentos com similaridade máxima é encontrado pela primeira localização do elemento máximo da matriz A. Os outros elementos máximos conduzem para este valor máximo que são determinados sequencialmente com um procedimento de *traceback* encontrando um elemento da matriz A igual a $A_{0,0}$.

2.5 Alinhamento Semi-Global Utilizando o Método de Programação Dinâmica

De acordo com Raupp [66], este alinhamento tem por objetivo não cobrar os pontos dos espaços, no início e/ou no final de alinhamentos, entre seqüências, porém o problema é encontrar a similaridade entre duas seqüências sem penalizar espaços no início e/ou no final do alinhamento das seqüências de DNA.

O alinhamento semi-global permite alinhar seqüências de DNA, mas para isso é preciso adaptar o algoritmo básico da técnica de programação dinâmica prevendo seis casos diferentes [66].

A aplicação é dada pela montagem dos fragmentos de DNA, onde se busca uma seqüência pequena (**T**) dentro de uma grande seqüência (**S**), visto que não tem como objetivo penalizar espaços nas pontas da pequena seqüência **T**.

Um exemplo dessa aplicação é ilustrado a seguir:

A seguir, são mostrados todos os seis casos adaptando o algoritmo básico de programação ao alinhamento semi-global.

1º Caso: o alinhamento semi-global acontece entre uma seqüência **S** e um prefixo da seqüência **T**.

Seqüência S ⟶ S_1 S_2 S_3 … S_m _ _ _ _
Sequencia T ⟶ T_1 T_2 T_3 … T_j … T_{n-1} T_n

2º Caso: o alinhamento ocorre entre um prefixo das seqüências S e T.

Seqüência S ⟶ S_1 S_2 S_3 ... S_i ... S_{m-1} S_m
Seqüência T ⟶ T_1 T_2 T_3 ... T_n _ _ _ _

3º Caso: Apresenta as seqüências S e T, as quais terminam com espaços.

Seqüência S ⟶ S_1 S_2 S_3 ... S_m _ _ _ _
Seqüência T ⟶ T_1 T_2 T_3 ... T_j ... T_n _ _

OU

Seqüência S ⟶ S_1 S_2 S_3 ... S_i .. .S_m _ _
Seqüência T ⟶ T_1 T_2 T_3 ... T_n _ _ _ _

4º Caso: o alinhamento entre a seqüência **S** e um sufixo da seqüência **T**.

Seqüência S ⟶ _ _ _ _ S_1 S_2 ... S_m
Seqüência T ⟶ T_1 T_2 ... T_j T_{j+1} ...T_n

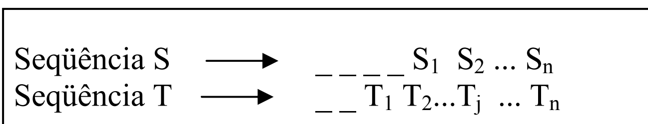
5º Caso: o alinhamento ocorre entre um sufixo da seqüência S e T.

Seqüência S ⟶ S_1 S_2 ... S_i S_{i+1} ...S_m
Seqüência T ⟶ _ _ _ T_1 T_2 ... T_n

6º Caso: o alinhamento ocorre entre sufixos das seqüências S e de T.

Seqüência S ⟶ _ _ S_1 S_2 ... S_i ... S_m
Seqüência T ⟶ _ _ _ _ T_1 T_2 ...T_n

OU



Assim, foram apresentados os três tipos de alinhamentos que fazem uso da técnica de programação dinâmica. O primeiro algoritmo de *Needleman-Wunsch* que visa o alinhamento global e realiza o alinhamento de seqüências inteiras. O segundo, o algoritmo de *Smith-Waterman*, que é uma modificação do algoritmo global e visa obter o alinhamento local, visto que o alinhamento localiza fragmentos de seqüências que são mais similares. O terceiro, o alinhamento semi-global, que através de seis adaptações da técnica de programação dinâmica consegue realizar vários alinhamentos sem penalizar os espaços.

Além desses algoritmos que são considerados importantes na comunidade acadêmica da bioinformática, existem muitos *softwares* que realizam o alinhamento de seqüências como é o caso do BLAST (*Basic Local Alignment Search Tool*) [1].

Os recursos computacionais como *hardware* e *softwares* voltados para o alinhamento de seqüências são descritos detalhadamente no capítulo 3.

2.6 Alinhamento Múltiplo de Seqüências Genéticas

Até o momento, foi dado enfoque à descrição dos algoritmos de alinhamentos com apenas duas seqüências, mas o problema de alinhar simultaneamente $K > 2$, (onde K representa a quantidade de seqüências a serem comparadas e alinhadas) é também muito importante uma vez que existe a necessidade de definir famílias de seqüências (principalmente de proteínas). Tal definição permite [69]:

- Caracterizar a família;
- Procurar novos membros da família;
- Entender a estrutura/evolução/função da família.

A comparação de duas seqüências não é suficiente para caracterizar uma família, pois as partes comuns a todos os membros da família podem estar dispersas no meio de outras partes que são comuns a apenas dois membros (pois vêm de organismos evolutivamente próximos e, portanto compartilham mais história) ou então o alinhamento duas a duas pode não ser significativo (a pontuação é próxima daquela de

duas seqüências aleatórias); mas quando se alinham várias seqüências e todas elas compartilham de certas regiões, esse compartilhamento é muito mais significativo.

O alinhamento múltiplo pode ser visto como a resolução inversa do problema do alinhamento de duas seqüências. Ao encontrar um alinhamento significativo entre duas seqüências (quaisquer) S e T, faz-se a descoberta de que as seqüências S e T são relacionadas biologicamente [69].

Atualmente, a comparação de seqüências é um importante problema de toda a biologia computacional, tendo como objetivo estabelecer parâmetros de semelhanças entre duas ou mais seqüências respeitando critérios específicos.

O alinhamento de seqüências genéticas pode ser definido como a maneira de comparar duas ou mais seqüências por meio de buscas de uma série de caracteres ou padrões de caracteres que não precisam estar na mesma ordem.

A comparação de seqüências tem origem no problema da distância de edição que avalia o quanto duas seqüências são diferentes em sua edição, visto que, cada operação está associada a um custo para determinar as operações necessárias para transformar a seqüência S na seqüência T. Logo, a distância entre duas seqüências visa obter o menor custo resultante de um conjunto de operações possíveis para transformar S em T.

Contudo, a programação dinâmica resolve o problema de distância de edição, que também pode ser aplicada aos algoritmos de alinhamento global, local e semi-global para que seja possível obter o alinhamento máximo ou alinhamento ótimo nas seqüências que estão sendo alinhadas e comparadas.

Capítulo

3

SOFTWARES E *HARDWARES* PARA COMPARAÇÃO DE SEQÜÊNCIAS

3.1 Considerações Iniciais

A comparação de seqüências biológicas, tais como a pesquisa de DNA ou banco de dados de proteínas, é uma tarefa fundamental na biologia molecular. Esta operação consiste principalmente em identificar segmentos similares entre duas seqüências. A complexidade computacional desta operação é proporcional ao produto do tamanho das duas seqüências.

Softwares como o BLAST [1] ou FASTA [61] são extensivelmente usados para desempenhar a visualização dos bancos de dados biológicos. Eles têm sido projetados para executar em computadores padrão, isto é, máquinas *Von Neuman*, e incluem técnicas para acelerar os processos. Essas técnicas são baseadas em heurísticas as quais podem ser compostas de parâmetros externos [47].

A sensitividade da busca depende principalmente desses parâmetros. Em geral, a baixa sensitividade implica um curto tempo de computação (na ordem de minutos),

enquanto uma alta sensitividade envolve um tempo de computação muito maior (na ordem de horas). De início poderia-se pensar que no futuro, o aumento de poder nos micro-processadores poderia diminuir o tempo de computação.

Porém, os bancos de seqüências crescem em tamanho aproximadamente 50% ao ano, então não há razão para esperar esta diminuição de tempo de solução nos próximos anos, pois o desempenho dos bancos de dados biológicos e os micro-processadores crescem aproximadamente na mesma taxa.

Entretanto, biólogos que usam computadores *Von Neuman* continuaram a ter o dilema de obter os resultados incompletos num curto período de tempo ou aguardar um longo período de tempo para obter soluções satisfatórias.

Uma maneira de limitar o tempo de execução é paralelizar a computação. Assim, três abordagens podem ser propostas para suportar tarefas de concorrência: redes de computadores, máquinas maciçamente paralelas ou *hardwares* dedicados.

Este capítulo faz a abordagem dos principais *softwares* voltados para a comparação de seqüências tais como: BLAST, FASTA e *Clustal* W, logo em seguida se apresentam os *hardwares* projetados para acelerar a comparação de seqüências biológicas sendo estas baseadas em estruturas de arranjos (*arrays*) lineares, tais como: VLSI *Dedicated Arrays*, FPGA *Arrays* e VLSI *Programmable Arrays* juntamente com alguns dos principais sistemas que compreendem estes *hardwares* e no final do capítulo se apresentam outros trabalhos relacionados.

3.2 *Softwares* Utilizados para a Comparação de Seqüências Biológicas

Com o aumento dos investimentos na indústria da informática no desenvolvimento de *softwares* para a biologia molecular, há uma tendência da bioinformática ficar mais comercial e menos acadêmica, é o que menciona a avaliação da professora Diana Magalhães de Oliveira, coordenadora do Núcleo de Genômica e Bioinformática da Universidade Estadual do Ceará [68].

Atualmente, a maior parte dos *softwares* usados para a bioinformática foi desenvolvida por pesquisadores de universidades, que distribuem gratuitamente os programas para as instituições de pesquisa e os vendem, eventualmente, para empresas privadas.

A partir dos resultados, ainda incertos, da mudança no contexto de produção desses programas de análise genética, crescem as expectativas de ampliação do

comércio no setor e aumenta a necessidade do debate sobre as vantagens, as desvantagens e a forma que deve assumir a mercantilização da atividade.

As seqüências de bases nitrogenadas que compõem o DNA, obtidas pelos projetos genoma, teriam pouca utilidade se não existissem as ferramentas computacionais necessárias para a compreensão da enorme quantidade de dados resultantes. Os benefícios sociais, ambientais e econômicos que prometem os cientistas a partir das pesquisas genômicas só serão possíveis se existir a capacidade de integrar a análise do DNA com as diferentes funções que ele pode determinar no organismo [68].

Dentro desta perspectiva, o rápido desenvolvimento de *softwares* que consigam responder melhor às necessidades específicas das pesquisas, é um fator fundamental para o desenvolvimento científico e tecnológico no Brasil.

Com o auxílio da bioinformática é possível extrair informações relevantes a partir das seqüências de DNA e de proteínas, obtidas pelo processo de seqüenciamento automático de nucleotídeos e de aminoácidos. A análise computacional pode desvendar detalhes e revelar arranjos na organização de dados genômicos e proteômicos, ajudando a esclarecer a estrutura e a função dos genes e proteínas estudados [68].

Os principais programas utilizados, geralmente, são os *softwares* desenvolvidos exclusivamente para esta função. Eles são elaborados na plataforma *Unix* ou *Linux*, que têm uma interface menos "amigável", ou seja, são mais difíceis de serem operados em relação aos programas convencionais, mas possibilitam uma maior capacidade de intervenção do pesquisador para atender a suas exigências.

Nesta seção são explanados alguns dos principais *softwares* que contribuem para a descoberta e a evolução da bioinformática.

3.2.1 FASTA

FASTA é a família de *softwares* para fazer busca em bancos de seqüências de DNA e de proteínas [66]. O algoritmo FASTA é um método heurístico para comparação de *string*. Foi desenvolvido por *Lipman* e *Pearson* em 1985 [61] e melhorado em 1988 [79]. FASTA compara uma *string query* junto à única *string* de texto. Quando se busca todo o banco de dados para *matches* para uma dada seqüência *query*, compara-se a seqüência *query* usando o algoritmo FASTA para toda *string* no banco de dados.

O algoritmo FASTA usa um método de busca de similaridades locais por meio do emprego de matriz de substituição. Concentra-se na busca de casamentos de sub-

seqüências curtas de tamanho k (k-uplas) que podem contribuir para o casamento total, usando implementações computacionalmente eficientes e que não se baseiam em *hardwares* específicos.

Com vistas à eficiência de processamento, este *software* utiliza inicialmente o padrão observado de casamento de resíduos. Ao invés de analisá-los individualmente, o *software* procura segmentos com alta incidência destes casamentos. Por meio de um procedimento heurístico, atribui-se um *score* a cada um destes segmentos de alta incidência, que serão compostos ao final da busca para produzir um *score* global.

Somente um alinhamento é fornecido para cada uma das seqüências do banco de dados utilizado que possui um alto *score* de alinhamento com a seqüência de teste.

Assim, como outros alinhamentos importantes podem não ser observados por este enfoque, estes casamentos devem ser re-analisados por outros *softwares* tais como o programa LALIGN [61]. Além dos *scores* citados, o *software* FASTA apresenta ainda uma estimativa da significância estatística de cada alinhamento encontrado.

Antes de submeter a seqüência, alguns dígitos numéricos na seqüência *query* podem ser removidos ou substituídos pelos códigos de letra apropriados (por exemplo: N para um resíduo de ácido nucléico não conhecido ou X para resíduo de aminoácido não conhecido).

Os códigos de ácidos nucléicos aceitos são:

```
A --> adenosina           M --> A C (amino)
C --> citosina            S --> G C (forte)
G --> guanina             W --> A T (fraca)
T --> timina              B --> G T C
U --> uridina             D --> G A T
R --> G A (purina)        H --> A C T
Y --> T C (pirimidina)    V --> G C A
K --> G T (keto)          N --> A G C T

(qualquer uma das bases)

- gap de comprimento indeterminado
```

3.2.2 BLAST

O *software* BLAST (*Basic Local Alignment Search Tool*) [1] é o método mais utilizado para realizar buscas de seqüências similares em bancos de dados de seqüências, servindo como instrumento de análise de identificação de genes, sendo que suas implementações mais conhecidas são a do NCBI (*National Center for*

Biotechnology Information) [55] e a da Universidade de *Washington*, conhecido como WU-BLAST.

BLAST é um *software* que faz alinhamento local, mas ele não usa programação dinâmica e sim métodos de heurísticas. Este *software* é usado para buscas em bancos de seqüências. A partir de uma dada seqüência consulta (*query*) do usuário, tenta achar todas as seqüências do banco (*subjects*) que têm alinhamentos estatisticamente significativos [75].

O algoritmo utilizado por BLAST pode ser resumido nos seguintes passos e baseia-se na idéia de que bons alinhamentos locais provavelmente contêm pequenos segmentos de identidades [17]:

- ✓ Compilar uma lista de segmentos de alto *score* (*word* no jargão de BLAST). Para proteínas, essa lista é formada por todas as palavras com w caracteres (w-mer) com *score* no mínimo T com algum w-mer da seqüência *query*;
- ✓ Procurar por *hits* na base de dados (cada *hit* corresponde a uma semente). Um *hit* é um pequeno segmento alinhado onde cada posição do alinhamento corresponde a uma identidade (as duas seqüências possuem o mesmo aminoácido na posição correspondente); e
- ✓ Estenda os *hits*. Essa extensão é realizada nos dois sentidos; inicialmente era realizada sem considerar *gaps* [1], mas atualmente, as extensões são feitas com *gaps* e o processo para estender uma semente só é disparado se o seu *score* for maior que um limiar T, ela possui outra semente a certa distância máxima entre elas e se o *score* da extensão com *gaps* que elas geram excede a um dado limiar segmento [2].

Observa-se que, para o cálculo dos *scores*, é utilizada uma matriz de substituição tal como BLOSUM. A matriz BLOSUM [30] é uma matriz de substituição de aminoácido, calculada primeiramente por *Henikoff* [29]. Para este cálculo somente blocos de seqüências de aminoácido com pequenas mudanças entre elas são consideradas. Esses blocos são chamados de blocos conservados (figura 3.1).

```
AABCDA...BBCDA
DABCDA.A.BBCBB
BBBCDABA.BCCAA
AAACDAC.DCBCDB
CCBADAB.DBBDCC
AAACAA...BBCCC
```

Figura 3.1 Alinhamento de diversas seqüências [44].

Dependendo do tipo de busca que se queira realizar, pode-se utilizar o BLAST através de uma das seguintes formas [75]:

- ✓ **BLASTP:** para comparação de seqüências de aminoácidos em bancos de dados de proteínas;
- ✓ **BLASTN:** para comparação de seqüências de nucleotídeos em bancos de dados de DNA;
- ✓ **BLASTX:** para comparação de uma seqüência de nucleotídeos transladada em todos os ORFs *(Open Reading Frames*) com bancos de dados de proteínas;
- ✓ **TBLASTN:** para comparação de seqüência de proteína com um banco de dados de seqüências de nucleotídeos dinamicamente transladados em todos os seus ORFs; e
- ✓ **TBLASTX:** para comparar os ORFs de uma seqüência de nucleotídeos com os ORFs de todos os nucleotídeos em um banco de dados de nucleotídeos.

Além disso, ao formular uma seqüência *query* para busca, pode-se delimitar o espaço de busca de várias formas:

- o Banco de dados a ser utilizado na busca; e
- o Organismo específico.

Portanto, o *software* BLAST constitui-se, hoje, numa ferramenta de fundamental importância para biólogos moleculares, pois permite que no estudo de uma seqüência, seqüências potencialmente homólogas sejam encontradas, fornecendo ainda medidas estatísticas para a avaliação da significância da similaridade detectada.

Entretanto, a decisão sobre se uma similaridade detectada representa uma homologia passa obrigatoriamente pela interpretação biológica do alinhamento obtido por BLAST. Assim, BLAST por si só não é suficiente para revelar uma homologia, mas constitui-se num primeiro passo fundamental neste sentido. Por isso a importância da correta interpretação de seus parâmetros e resultados.

3.2.3 *Clustal* W

O *software Clustal* W [33] é um dos pacotes de *softwares* mais conhecido para alinhamento múltiplo de seqüências de DNA ou de proteínas. Este produz um alinhamento múltiplo biologicamente significativo, isto é, identidades, similaridades e diferenças podem ser visualizadas.

Faz alinhamento progressivo, começando por duas seqüências com a menor distância. Depois avalia se uma nova seqüência é alinhada ao par ou se outras duas são alinhadas isoladamente.

O algoritmo do *Clustal* W tem se tornado mais popular para o alinhamento de múltiplas seqüências [31][32][33][34][84][85]. Este *software* implementa um método progressivo para o alinhamento de múltiplas seqüências. O algoritmo seguinte apresenta uma descrição em alto nível das 3 fases básicas do algoritmo de alinhamento do *software Clustal* W.

Entrada: um conjunto **S** de **N** seqüências.

Saída: um alinhamento múltiplo do conjunto **S**.

1. **Alinhamento**: calcula os alinhamentos para todas as seqüências junto a todas as outras seqüências e armazena o resultado em uma matriz de similaridade. Converte os valores na matriz de seqüência de similaridade para medidas de distância, a qual reflete a distância evolucionária entre cada par de seqüências;
2. **Árvore-Guia**: técnica usada para estruturar as distâncias das seqüências duas a duas, isto é, constrói uma árvore guia que define a ordem nos quais os pares de seqüências são alinhados e combinados com alinhamentos prévios usando a matriz de similaridade e o algoritmo de *neighbour-joining*;
3. **Alinhamento Múltiplo**: alinhamento progressivamente seguindo a árvore-guia. Inicia-se pelo alinhamento dos pares mais próximo relacionados e cada passo alinha duas seqüências ou um para um sub-alinhamento existente; e
4. Final do algoritmo.

3.3 *Hardware* para Comparação de Seqüências Biológicas

A bioinformática utiliza como ferramentas a matemática aplicada e a computação. A Biologia molecular de hoje seria impossível sem os recursos de bioinformática, tais como o armazenamento, distribuição e atualização das informações, as análises estatísticas, a modelagem de dados e a simulação de fenômenos biológicos em computador.

A versatilidade, a velocidade e o aumento do poder computacional dos PCs (Computadores Pessoais) em redes locais, têm guiado os cientistas no trabalho com modelagem molecular e alinhamento de múltiplas seqüências de DNA (evolução molecular) independentemente da existência de supercomputadores [47].

Os laboratórios podem ser equipados com várias opções de interfaces com PCs que ajudam nas pesquisas em aplicações que atendam às necessidades específicas dos experimentos.

A maioria das máquinas (*hardwares*) que tem sido projetadas para acelerar a comparação de seqüências biológicas são baseadas em estrutura de arranjos (*arrays*) lineares.

Entretanto, essas estruturas diferem na flexibilidade de programação. Dentre essas podem ser destacadas três categorias [47]:

- ***Arrays* Dedicados VLSI (*Very Large Scale Integration*)**: essas máquinas podem conseguir a mais alta performance sob um único algoritmo, esse algoritmo é adaptado dentro do silício e não pode ser modificado;
- ***Arrays* FPGAs (*Field Programmable Gate Arrays*)**: eles incluem sistema com *hardware* reconfigurável (FPGA). Eles tendem a ser mais lentos e têm densidade muito mais baixa do que os *arrays* VLSI. A criação e modificação de algoritmos para esses sistemas são possíveis; e
- ***Arrays* Programáveis VLSI**: essa última categoria de máquinas empenha-se para a flexibilidade algorítmica de sistemas reconfiguráveis e a velocidade e a densidade de máquinas VLSI de propósito único. Logo, a principal vantagem é a facilidade de programação.

A figura 3.2 ilustra as diferenças referentes aos itens mencionados. Os *arrays* dedicados VLSI são vinte vezes mais rápidos do que os FPGAs ou os *arrays* programáveis VLSI. Já, os FPGAs e os *arrays* programáveis VLSI têm performances comparáveis.

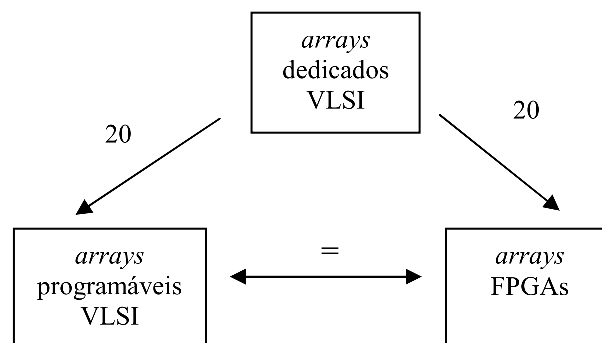


Figura 3.2 Performances dos sistemas VLSI e FPGA [47]

A seguir são apresentados os principais sistemas em *hardware* que se enquadram nas três categorias acima mencionadas.

3.3.1 *Array*s Dedicados VLSI - BioSCAN

O sistema BioSCAN (*Biological Sequence Comparative Analysis Node*) acelera a identificação de seqüências similares de DNA ou de proteínas sem permissão de *gaps*. Esse sistema foi projetado na Universidade da Carolina do Norte, um processador de *array* sistólico maciçamente paralelo construído junto a um *chip* VLSI do tipo ASIC (*Application Specific Integrated Circuit*). Cada *chip* ASIC contém 812 elementos de processamento individuais. O processador pode acomodar um máximo de 16 *chips* para um total de 12,992 elementos de processamento, o que permite implementar e pesquisar em banco de dados até 12,992 caracteres. Esse é um sistema poderoso para detectar seqüências similares de tamanho idêntico, entretanto, nenhuma versão comercial está disponível [47].

O projeto BioSCAN foi descrito primeiro em um artigo por *White* et al. em 1991 [92] e depois expandido em *Dettloff* et al. em 1991 [16] e *Singh* et al em 1993. [77]. A base do *hardware* BioSCAN é um algoritmo baseado sob o trabalho feito por S. *Karlin* e S.F. *Altschul* [45].

Em geral, pares de segmentos (um segmento do banco de dados de seqüências e o outro da seqüência *query*) podem ser considerados similares embora ainda que apenas alguns, se quaisquer, caracteres dentro dos segmentos combinem identicamente. Essa característica distingue análises de seqüências biológicas de outros padrões de problemas de comparação onde tipicamente olha-se para pares de seqüências curtas idênticas.

O programa seguinte descreve um fragmento do algoritmo de BioSCAN representado na linguagem de programação C. O *hardware* BioSCAN desempenha o *loop* interno em paralelo. Um teste para a soma negativa permite os mais altos *scores* aos pares de seqüências para ser computado em um único passo.

Programa 1

```
#include <stdio.h>

#define M 28
#define N 28

/*BioSCAN declaration*/
```

```
/* Inputs */

      unsigned int La, Lb;        /* length A, B */
      char A[La],                       /* sequence A-padded */
           B[Lb];                       /* sequence B */
      signed int T[M][N],         /* similarity table */
           Th;                          /* threshold */

/* Outputs */
      signed int S[Lb+1]; /* scores 1..Lb */

/* Temporaries */
      int i, j;          /* index A and B */

extern report();

/* Algorithm */
Main ()
{
      for (i=0; i<Lb; ++i) S[i]=0;          /* initialize */
      for (i=0; i<La; ++i) {                /* elements of A */
         for (j=Lb-1; j>=0; --j)            /* elements of B */
           if (S[j] < 0) S[j+1] = T[A[i]][B[j]];
           else if (S[j] >= Th) S[j+1] = S[j];
           else S[j+1] = S[j] + T[A[i]][B[j]];/*accumulate*/
         if (S[Lb] >= Th) report(i);
  }
}
```

Programa 1. Algoritmo BioSCAN [38].

O sistema BioSCAN compara uma dada seqüência *query* com cada seqüência em um banco de dados especificado. O sistema identifica localizações no banco de dados que são suficientemente similares para a seqüência *query* para satisfazer uma medida pré-programada *goodness-of-fit* especificada pela tabela T[1] e a entrada do Programa 1 [38]. Pares de segmentos representados sob a mesma diagonal correspondem ao mesmo alinhamento de seqüências A e B.

Em qualquer alinhamento entre as seqüências A e B, existe uma diferença constante nos índices de cada par de letras alinhadas. Isso fornece uma maneira conveniente para numerar os alinhamentos (e diagonais correspondentes). Consequentemente, a diagonal (i-j) contém as letras alinhadas, ou seja, A[i] e B[j].

O BioSCAN não suporta algoritmos de programação dinâmica. Como o BLAST, ele tem sido projetado para detectar segmentos similares de tamanhos idênticos. Consequentemente, o algoritmo é mais simples e habilita uma alta densidade dos processadores (812 por *chips*) para adequar-se no silício [47].

Se todos os elementos de processamento estão em uso, o processador é capaz de comparar (teoricamente) o máximo de 25.984 bilhões de caracteres por segundo. Cada elemento de processamento é um processador de *bit* serial que desempenha o *loop* interno fundamental do algoritmo do BioSCAN nos 16 principais ciclos de *clocks* [38].

Quando o *hardware* é executado com uma taxa de *clock* de 32 MHz, 2 milhões de caracteres são processados por segundo. De acordo com as condições de processamento é possível obter taxas de dados tão altas como 1.978 milhões de caracteres por segundo [38].

3.3.2 Arquitetura RISC de Propósito Geral – SUN 690

A arquitetura RISC (*Programmable Reduced Instruction Set Computer*) [5] é constituída por um pequeno conjunto de instruções simples que são executadas diretamente pelo *hardware*, sem a intervenção de um interpretador (microcódigo), ou seja, as instruções são executadas em apenas uma microinstrução.

A máquina de propósito geral, *Sun* 690 com 64 MBytes de memória principal foi utilizada para implementar o algoritmo (programa 1) tendo como objetivo obter dados de performance a serem comparados com a arquitetura vetorizada e a SIMD (*Single Instruction Multiple Data*).

Dado o tamanho de memória, todo o teste do banco de dados foi testado facilmente o qual conteve espaço do *buffer* de I/O (*Input/Output*) [51]. O algoritmo (programa 1) mostrado anteriormente é executado nessa máquina, tendo esse algoritmo sofrido alterações para ser possível executar nessa arquitetura RISC, tais como a inclusão de uma chamada **"get ()"** para ler os elementos no banco de dados.

3.3.3 Arquitetura Vetorizada – CONVEX 240

Aplicações científicas envolvendo processamento de grandes quantidades de dados, como problemas de física nuclear, termo e hidrodinâmica, assim como previsões de clima, sempre contribuíram para o desenvolvimento de computadores de alto desempenho. Para suprir esta necessidade, surgiram os supercomputadores, que introduziram inúmeros avanços no desenvolvimento dos computadores. Entre as inovações que surgiram, estão o processamento em *pipeline* e o processamento vetorial [5].

A arquitetura de vetor utilizada para realizar os testes foi uma arquitetura chamada *Convex* 240 [38]. Novamente, o algoritmo (programa 1) mostrado anteriormente é executado nessa arquitetura, *Convex* 240, cujas alterações encontram-se na divisão do *loop* interno em três *loops* separados, os quais testam as somas menores que zero ou maiores do que o limite dos valores da tabela de similaridade.

Esta arquitetura mostrou ser mais rápida do que a *Sun* 690, comporta-se da mesma maneira quando aumenta o tamanho da seqüência *query*.

3.3.4 Arquitetura SIMD – MP1

A arquitetura SIMD (*Single Instruction Multiple Data*) corresponde ao processamento de vários dados sob o comando de apenas uma instrução. Em uma arquitetura SIMD, o programa ainda segue uma organização seqüencial. Para possibilitar o acesso a múltiplos dados é preciso uma organização de memória em diversos módulos.

A arquitetura SIMD utilizada foi a MasPar MP-1 [38], a qual foi configurada como uma rede de processadores de 128 por 64 (total de 8192 elementos de processamento) com 64 *KBytes* de memória em cada processador. Essa arquitetura também executou o algoritmo (programa 1), sendo que o código foi executado sobre uma unidade paralela do MasPar que se comunicou com um programa *front-end* executando sob *workstation* DEC (*Digital Equipament Corporation*) [38].

As tabelas 3.1 e 3.2 apresentam os dados das performances do tamanho da seqüência *query* usando as métricas CPS (*Characters Processed per Second*) e o COPS (*Comparison Operations per Second*) para os sistemas BioSCAN, Sun 690, a Convex 240 e a MasPar-MP1.

Tabela 3.1 Medida de Performance em Caracteres Processados por Segundo [38].

Tamanho/ Seqüência	**Performance em CPS**					
	10	**50**	**100**	**500**	**1000**	**5000**
BioSCAN	1.979.583	1.979.583	1.979.583	1.979.583	1.979.583	1.979.583
Sun 690	5.776.729	1.235.597	625.385	125.359	62.629	12.489
Convex 240	6.423.184	2.953.318	1.733.355	371.772	189.344	39.535
MasPar-1	1.162.529	1.160.915	1.160.850	1.161.847	1.161.107	1.154.186

Tabela 3.2 Medida de Performance em Comparações de Operações por Segundo [38].

Tamanho/ Seqüência	Performance em COPS					
	10	**50**	**100**	**500**	**1000**	**5000**
BioSCAN	19.80e6	98.98e6	197.96e6	989.79e6	1979.58e6	9897.91e6
Sun 690	577.673	617.799	625.385	626.794	626.287	624.447
Convex 240	642.318	1.476.659	1.733.355	1.858.858	1.893.439	1.976.755
MasPar-1	116.253	580.458	1.160.850	5.809.237	11.611.066	57.709.315

E com intuito de melhor visualizar os dados que constam nas tabelas acima as figuras 3.3 e 3.4 ilustram essas performances.

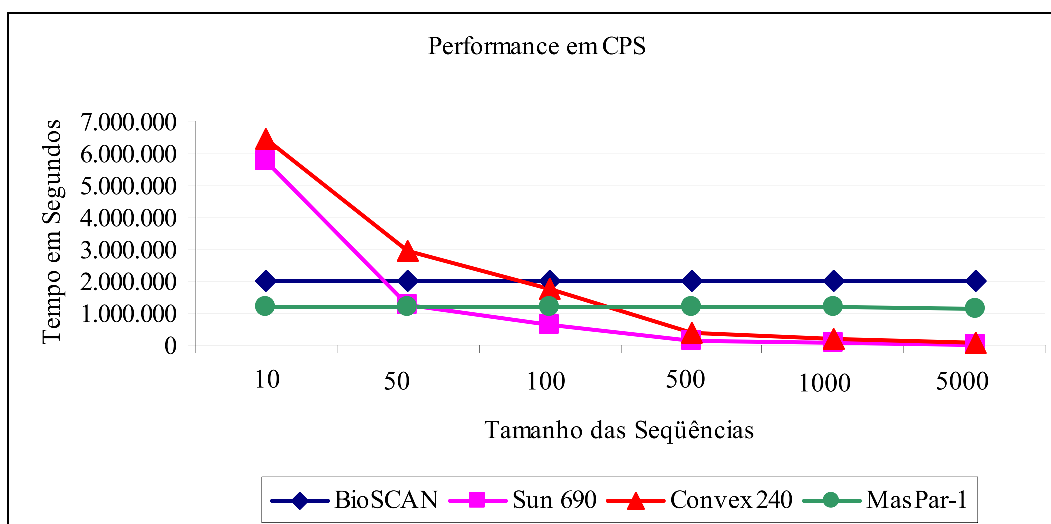


Figura 3.3 Representação das Performances em CPS

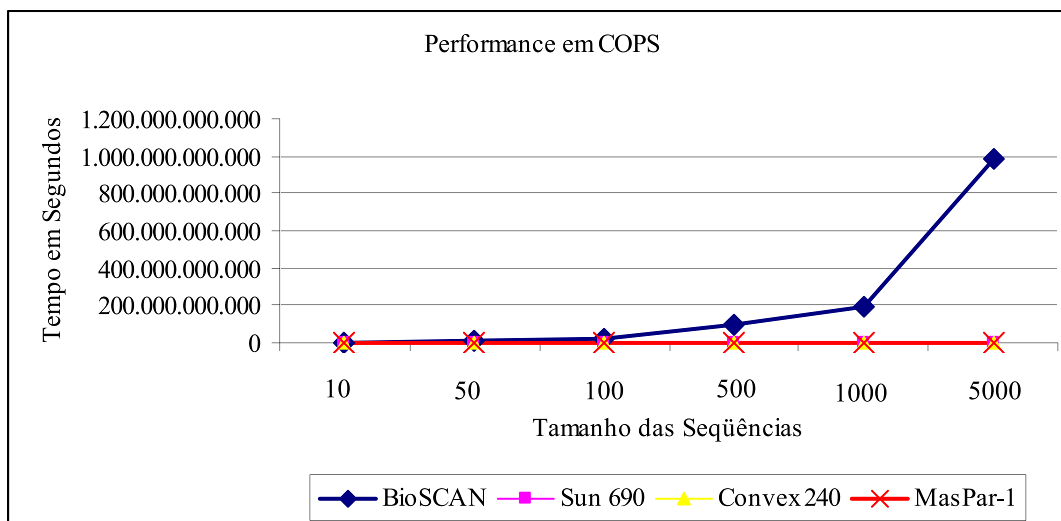


Figura 3.4 Representação das Performances em CPOS

Nessas figuras é possível destacar o seguinte:

BioSCAN: de acordo com as condições de processamento é possível obter taxas de dados tão altas como 1.978 milhões de caracteres por segundo. Isso significa que a taxa de 1.978 caracteres processados por segundo - CPS se manterá constante enquanto a taxa de operações comparadas por segundo – COPS aumentará linearmente com o tamanho da seqüência *query*;

Sun 690: é aguardado que a taxa de CPS caia com o aumento no tamanho da seqüência. Além disso, também é aguardado um melhoramento na taxa de COPS quando aumentar o tamanho das seqüências;

Convex 240: nos tamanhos maiores do que 50 a taxa de melhoramento adequada para a vetorização cai rapidamente. Uma seqüência com 100 caracteres exige 0.577 milissegundos para comparar, havendo uma redução de 14% no tempo de operação, enquanto uma seqüência de 500 caracteres exige 0.506 milissegundos, uma melhora de 25% sobre uma seqüência de 50 caracteres. Numa seqüência de 5.000 caracteres o trabalho eficiente foi ainda melhorado com o aumento do tamanho da seqüência, porém a taxa de melhora foi igualada; e

MasPar – 1: observa-se que a taxa de CPS do MasPar permaneceu essencialmente constante para todas os tamanhos de seqüências testadas. As taxas COPS aumentaram linearmente com o aumento no tamanho das seqüências. Para esta configuração em particular, com 8192 processadores, a taxa de comparação máxima excedeu 95 milhões de COPS.

3.4 *Arrays* Dedicados VLSI - SAMBA

SAMBA (*Systolic Accelerator for Molecular Biological Application*) [47] é um *array* sistólico dedicado para analisar os algoritmos envolvidos na comparação de seqüências biológicas. Esse *hardware* acelera uma versão parametrizada do algoritmo de *Smith-Waterman* [80] permitindo a computação de alinhamentos globais e locais com ou sem *gap penalty*. As computações que exigem diversas horas em *workstations* são executadas em menos de dez segundos no SAMBA [47].

Esse *hardware* está disponível desde o final de 1995 e é amplamente usado pelos biologistas para tarefas de intensas comparações ou para buscas em bancos de dados por meio do servidor de WEB SAMBA, cuja URL: é (http://www.irisa.fr/SAMBA).

SAMBA implementa um algoritmo parametrizado de *Smith* e *Waterman* [23][80]. Pela composição de alguns parâmetros diferentes, as comparações globais e locais podem ser executadas, com ou sem *gap penalty*. Desta maneira, a variedade de *software*, tais como BLAST [1] e FASTA [61] ou SSEARCH [62], podem ser implementados nesse acelerador.

A arquitetura SAMBA contém uma *workstation*, um *array* sistólico composto de 128 processadores de 12 *bits*, e uma interface baseada em FGPA (figura 3.5). A interface FPGA é a placa PerLe-1 desenvolvida por *Bertin* [7] em 1992 que funciona como um *driver* programável para o *array* sistólico.

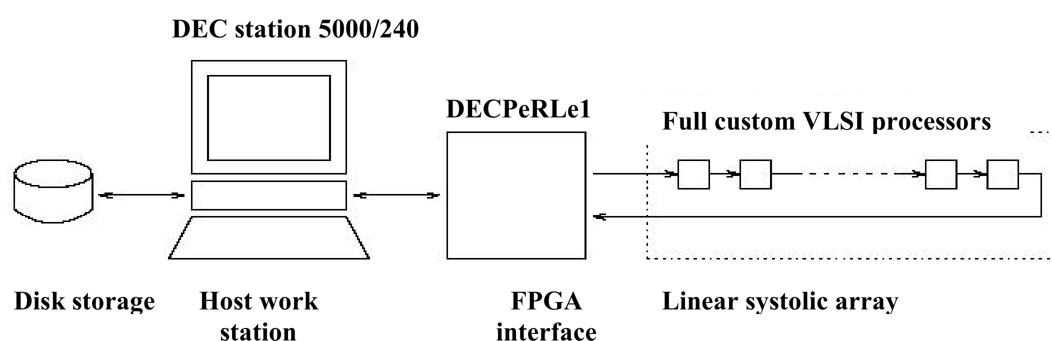


Figura 3.5 Sistema completo do *hardware* SAMBA [48].

SAMBA pode ser comparado com dois outros sistemas de *hardware*: BISP [14] e BioSCAN [92]. O BISP (*Biological Information Signal Processor*) fornece uma alta velocidade e um sistema linearmente extensível que pode localizar estaticamente subseqüências similares de duas seqüências de DNA ou de proteínas. Este implementa uma versão modificada do algoritmo do *Smith Waterman* e permite que vários parâmetros sejam estabelecidos.

O *hardware* foi projetado no Instituto de Tecnologia da Califórnia. Um *chip* contém 16 processadores e um protótipo da máquina de 16 *chips* foi validado, tornando possível a computação de tamanho de seqüências ilimitadas. Essa máquina não tem uma versão comercial.

Os protótipos do BISP e BioSCAN contém respectivamente 256 processadores de 16 *bits* e 12.000 processadores de 1 *bit*. Porém, a arquitetura BISP e SAMBA são similares. Elas diferem da seguinte maneira: o *array* BISP é acionado com um processador programável (Motorola 68020) enquanto o SAMBA usa tecnologia FPGA.

Esta última abordagem é mais simples e assegura que o alto *throughput* de dados exigidos pelo *array* sistólico seja mantido.

3.4.1 Algoritmo SAMBA

O algoritmo implementado pelo SAMBA pertence à classe de programação dinâmica. Esse modelo permite aos biologistas comparar seqüências biológicas com duas operações básicas de edição [48]:

1. A substituição de caracteres; e
2. A inserção ou omissão de caracteres; essa operação é chamada de *gap*.

Pelo uso de uma série de operações de edição, qualquer seqüência pode ser transformada em qualquer outra seqüência. O menor número de operações de edição exigida para mudar uma seqüência em outra é desta maneira uma medida da distância entre elas. A computação da distância de edição é realizada pela programação dinâmica.

A computação da equação do SAMBA vem do algoritmo bem conhecido de *Smith* e *Waterman*. A matriz de similaridade H é calculada recursivamente usando a seguinte equação [48]:

$$H(i,j) = Max \begin{cases} delta \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + Sbt(S1_i, S2_j) \end{cases}$$

com:

$$E(i,j) = Max \begin{cases} H(i, j-1) - alpha \\ E(i, j-1) - beta \end{cases} \quad F(i,j) = Max \begin{cases} H(i-1, j) - alpha \\ F(i-1, j) - beta \end{cases}$$

e as inicializações:

$$H(i,0) = E(i,0) = hi(i) \qquad H(0,j) = F(0,j) = vi(j)$$

Onde: **- Sbt** é um caracter de substituição da tabela de custo.

- *alpha, beta, delta, vi* e *hi* são parâmetros usados para selecionar diferentes variações do algoritmo de *Smith* e *Waterman* [48].

3.4.2 Paralelização

A paralelização dos algoritmos de comparação de *string* sob os *arrays* sistólicos lineares tem sido descrito na literatura [48]. SAMBA usa a arquitetura

mostrada na figura 3.6. Ela é composta de processadores idênticos, organizados linearmente.

A seqüência *query* é armazenada dentro do *array* (um caracter por processador) e o fluxo da outra seqüência da esquerda para a direita por meio do *array*

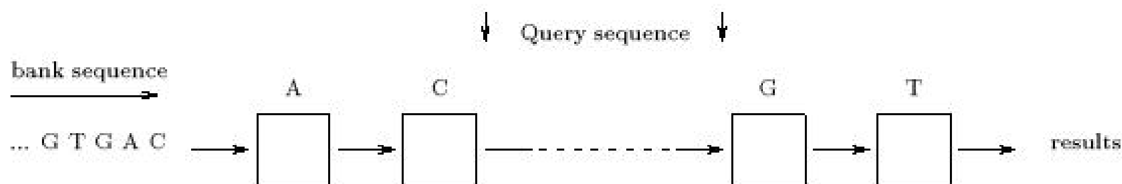


Figura 3.6 Comparação de seqüência sob um *array* sistólico linear [48].

Os dados movem-se em uma direção unidirecional da esquerda para a direita. Tipicamente, a comparação paralela de duas seqüências é desempenhada da seguinte maneira:

1. O *array* é inicializado com uma seqüência (geralmente chamado de seqüência *query*) numa taxa de um caracter por processador;
2. A outra seqüência é arrastada para a esquerda do *array* e prossegue todo ciclo do *array*; e
3. O resultado é coletado sob o processador mais à direita (*rightmost*) quando o último caracter da segunda seqüência está saindo.

Portanto, SAMBA é adequado para análises de seqüências que exigem uma grande quantidade de computação, tais como comparação banco a banco ou consulta a banco com longas seqüências. O crescimento exponencial dos bancos biológicos associados com o seqüenciamento completo do genoma torna o SAMBA uma solução interessante para o futuro.

3.5 *Arrays* FPGAs - *Splash*-2

A arquitetura *Splash*-2 foi desenvolvida pela *Supercomputing Research Center*, em 1992. O objetivo era a construção de uma máquina para aumentar o desempenho e automatizar o reconhecimento de padrões, ou seja, visava apenas as aplicações da biologia molecular [67].

A plataforma de desenvolvimento do *Splash* consiste de um processador *Sun* SPARC-2 e um conjunto de placas de processamento, cada uma contendo 17 FPGAs *Xilinx* XC4010 (X0-X16), podendo ser observado na figura 3.7.

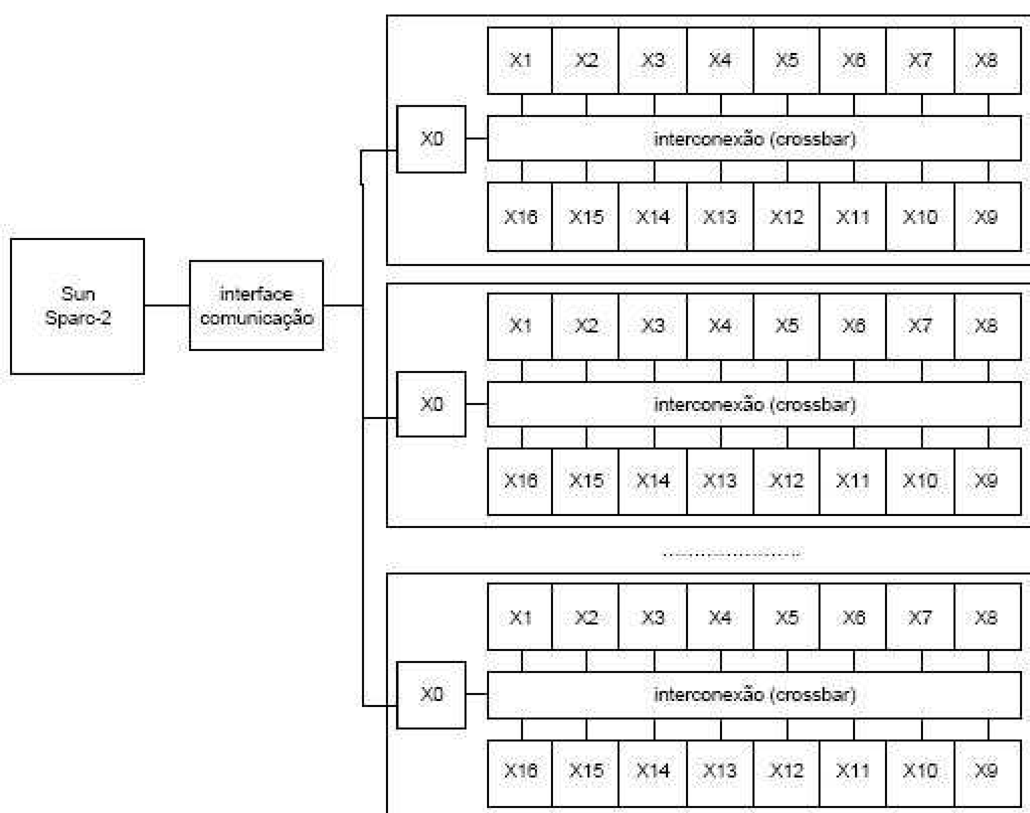


Figura 3.7 Arquitetura de cada elemento de processamento do *Splash* [37].

A arquitetura do *Splash*-2 pode facilmente suportar aplicações paralelas, tais como sistólica ou computações de dados paralelos. O *Splash* -2 é um melhoramento da primeira versão do *Splash* -1 [21], a qual consistiu de um tamanho fixo de *array* linear de *chips Xilinx* 3090 FPGA.

O *Splash* -2 tem diversos melhoramentos sob o SPLASH -1, isto é, escalabilidade, I/O de *bandwidth* e programabilidade. O *Splash* -2 foi baseado na tecnologia de *hardware*, o *Xilinx* XC4010 FPGA. Um *crossbar* é adicionado para conectar os elementos de processamento [21].

Os FPGAs são configurados através da ferramenta da *Xilinx*, via *software*. *Splash* não é uma arquitetura que faz uso intrínseco de sua reconfiguração, foram apenas acoplados diversos FPGAs ao sistema que, devidamente programados, aumentaram o desempenho global.

O ambiente de programação do *Splash* -2 é baseado em VDHL (*Very High Speed Integrated Circuit Hardware Description Language*) [27][86]. A descrição do procedimento é analisada (figura 3.8), simulado e sintetizado para dentro do FPGA *Xilinx*.

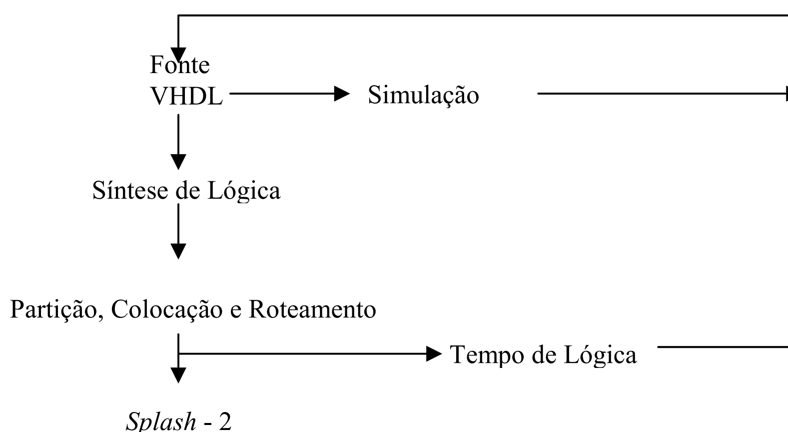


Figura 3.8 Diagrama de Fluxo da Programação do *Splash*-2 [13].

Uma aplicação para o *Splash* -2 é desenvolvida pela escrita de seus procedimentos em VHDL e a descrição é inteiramente refinada e debugada com o simulador do sistema *Splash* -2.

Uma análise de tempo estatística é aplicada ao módulo objeto para determinar a operação de velocidade máxima.

3.6 *Arrays* FPGAs - HScan

HScan é um filtro processador dedicado para pesquisa em bancos de dados de DNA. Foi desenvolvido pela IRISA (*Institut de Recherque en Informatique et Systèmes Aléatories*) [27][47] e validado sob uma plataforma FPGA, o protótipo PeRLe-1 [7], firmemente conectado a uma *workstation*. Este projeto mostra que o filtro pode auxiliar à performance das máquinas por um fator de alcance de 50 a 400 *workstations*. Este filtro encontra segmentos similares de tamanhos idênticos como o *software* BioScan faz.

A principal diferença entre os sistemas BioScan e HScan é que estes não fazem cálculos exatos, apenas detecta áreas potencialmente interessantes onde as similaridades podem aparecer [47]. O filtro age como um co-processador que desempenha as computações mais intensivas que ocorrem durante o processo.

O trabalho inclui a implementação das complexas células permitindo parametrizar custos de *match* e *mismatch*; os valores correntes +1 e -1 podem não ser adequados para a realidade biológica. Por exemplo, os parâmetros *default* do *software* BLAST são +5 e -4 para *match* e *mismatch,* respectivamente, e os parâmetros *default* do *software* FASTA são +4 e-3, respectivamente, para *match* e *mismatch*.

Além desse processo de detectar áreas onde similaridades de DNA podem ocorrer, foi proposta uma metodologia de projeto para automatizar a implementação de arranjos sistólicos: o algoritmo é paralelizado primeiramente usando a linguagem C-*sistolic* [27], então pode ser executado em máquinas paralelas para simulações intensivas, e finalmente é sintetizado para um FPGA. Para melhor compreender o funcionamento do filtro, na próxima seção é descrito o algoritmo.

3.6.1 Algoritmo Filtro

O objetivo do problema é localizar segmentos similares entre duas seqüências de DNA. Dados dois segmentos, Sg0 e Sg1, com N caracteres, a similaridade medida é apresentada pela equação 3.1 [27]:

$$S = \sum_{i=1}^{i=n} \mathbf{SUB}[Sg0(i)][Sg1(i)]$$

Equação 3.1

onde Sg(i) representa o primeiro caracter do segmento Sg e SUB uma tabela de substituição que identifica o custo de substituir um caracter em outro. No caso de seqüências de DNA, o alfabeto consiste de quatro letras, isto é, A,C, G e T (Adenina, Citosina, Guanina e Timina) e a tabela de substituição pode não ser usada.

Apenas os custos para *matches* e *mismatches* são considerados. A similaridade é calculada através da equação 3.2:

$$S = \sum_{i=1}^{i=n} \quad \mathbf{if}\ (Sg0(i) = Sg1(i))\ \mathbf{then}\ M\ \mathbf{else}\ N$$

Equação 3.2

onde M é o custo de um *match* e N o custo do *mismatch*. M é um número positivo enquanto N é um número negativo.

A descoberta de tais segmentos similares entre duas seqüências S0 e S1 implicam na computação da similaridade de todas as subseqüências possíveis de S0 com todas as subseqüências de S1. Isso é alcançado pelo cálculo de uma matriz Tx n0 x n1.

$$Tx[i][j] = MAX \begin{cases} Tx[i-1][j-1] + (\mathbf{if}\ (Sg0(i) = Sg1(j))\ \mathbf{then}\ M\ \mathbf{else}\ N) \\ 0 \end{cases}$$

com $0 < i \leq n0$ e $0 < j \leq n1$

A operação máxima mantém o valor positivo ou nulo de Tx[i][j]; o efeito desta operação é promover *scores* positivos "vis a vis" *scores* negativos: um *score* negativo indica nenhuma similaridade e não é uma informação interessante. Os *scores* que são memorizados no Tx[i][j] e que são maiores do que o valor de entrada representa as áreas de S0(i) e S1(j) onde as similaridades ocorrem [27].

Os dois segmentos similares de tamanho k são detectados entre um *score* máximo local S(i,j) localizado em Tx[i][j] e o primeiro *score* zero localizado no Tx[i-k][j-k]. Geralmente, o primeiro passo consiste em armazenar apenas os números das diagonais e o *score* máximo que foi calculado nessas diagonais. O algoritmo usado foi o seguinte:

```
for (i=1; i<=n0; i++)
   for (j=1; j<=n1; j++)
      {
       D[n0-i+j]=D[n0-i+j] + (if (S0[i]==S1[j])? M:N);
       if (D[n0-i+j]<0) then D[n0-i+j] = 0;
       else Smax[n0-i+j]=max(Smax[n0-i+j],D[n0-i+j]);
      }
```

A tabela **Smax** memoriza o *score* máximo das diagonais n0+n1-1. Somente a diagonal que tem um *score* maior do que o valor do limite então considerará precisamente segmentos similares localizados [27].

Isso constitui a base do algoritmo usado pelos *softwares* tais como: BLASTN ou FASTA para pesquisar no banco de dados e detectar segmentos similares. Para acelerar o processo, eles não desempenham a computação sob caracteres simples (que correspondem aos nucleotídios), mas sob palavras de diversos caracteres. A complexidade do algoritmo é determinada principalmente pelo tamanho das duas seqüências.

O propósito do filtro é comparar caracter por caracter. A idéia é incorporar a versão mais simples do algoritmo prévio em vez da computação do *score* máximo de uma diagonal.

Logo, o algoritmo recebeu algumas transformações, tais como [27]:

```
for (i=1; i<=n0; i++)
  for (j=1; j<=n1; j++)
  {D[n0-i+j]=D[n0-i+j] + (if (S0[i]==S1[j])? M:N);
     if (D[n0-i+j]<0) then
         D[n0-i+j] = 0;
     else R[n0-i+j] = R[n0-i+j] || D[n0-i+j]); }
```

Através da tabela R podem facilmente ser detectadas as diagonais interessantes, calcula-se o *score* exato e localiza-se os segmentos similares precisamente.

3.6.2 Arquitetura e Implementação do Filtro HScan

Uma simples análise de dependência sobre o *loop* aninhado mostra que a computação de cada diagonal pode ser desempenhada em paralelo e suportada sob uma rede linear sistólica. Cada processador é responsável pela computação do *score* da diagonal principal.

A arquitetura do processador é baseada principalmente sob um somador, o qual adiciona um valor positivo ou negativo de acordo com a igualdade ou diferença dos dados de entrada (descrita na figura 3.9).

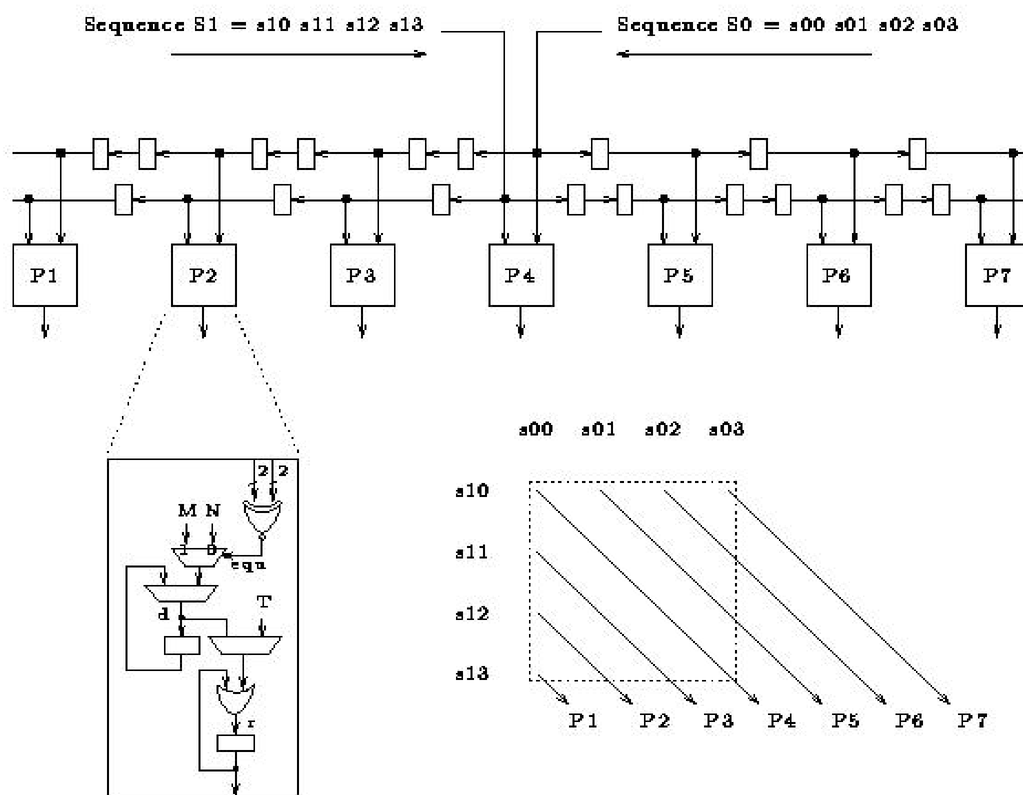


Figura 3.9 Princípio da arquitetura do filtro HScan [27].

Quando a saída torna-se negativa, o somador é ressetado para zero. A computação é feita por um processador que corresponde ao corpo do *loop* aninhado através do seguinte trecho de algoritmo [27].

```
Initialization:
   int d = 0;
   bool r = false;

systolic cycle:
   d = (c0==c1) ? (d + M):(d + N);
   d =  max(d,0);
   r = r || (d>=T)
```

A plataforma de *hardware* utilizada para validar o filtro foi uma placa PRL-DEC Perle-1 [6] desenvolvida pela DEC *Paris Research Laboratory*. Este também foi baseado no conceito PAM (*Programmable Active Memory*) [7] como um módulo de memória RAM (*Random Access Memory*), ou seja, uma PAM é anexada ao sistema de barramento de um computador *host*.

O processador pode escrever e ler da PAM. O processamento específico é determinado pelo conteúdo da configuração de sua memória. A placa Perle-1 foi construída junto a um grande *array* de células lógicas configuráveis de nível de *bit* envolvidas pelos bancos de RAM locais. O cálculo central do *array* consiste de uma matriz de 4x4 da *Xilinx* XC3090 *Programmable Gate Arrays* [27].

A interface do barramento é denominada *TurboChannel* liberando um *bandwidth* de 100 MBytes por segundo.

3.6.3 Performances do HScan

Para medir as performances, foi desenvolvido um protótipo "*software*", segundo *Guerdoux-Jamet* e *Lavenier* [27], chamado HScan, no qual o filtro foi integrado e age como um co-processador. Este *software* produz todos os pares de segmentos, com um *score* maior do que um valor limite (*threshold*), entre uma seqüência *query* de DNA e uma seqüência do banco de dados de DNA.

O algoritmo a seguir descreve como o HScan trabalha [27].

```
read (QS);                  /* lê a seqüência query */
NbSeqDB = open (DB);  /* abre o banco de dados */
for (I = 1; I <= NbSeqDB; i++)
 {
   read (DBS)   /* lê a primeira seq. do banco de dados*/
   NbDiag = filter(QS,DBS,M,N,T,NumDiag);
   for (j = 0; j < NbDiag; j++)
    {
      ComputeSSP(QS,DBS,NumDiag[j],M,N);
    }
 }
```

O tempo de execução tem sido comparado com o tempo de execução do *software* BLASTN (versão 1.4) e FASTA (atualização 17) em uma *workstation* SPARC 10 *workstation*. Os parâmetros desses *softwares* foram estabelecidos para produzir o mesmo resultado, isto é, M =1 e N = -1 (onde M é o custo de um *match* e N o custo do *mismatch*).

Desta maneira, FASTA não pode selecionar apenas pares de segmentos similares que tem um *score* maior do que o valor do limite (*threshold*). O diagrama da figura 3.10 mostra o *speed-up* comparado com os *softwares* BLASTN e FASTA para diferentes tamanhos de seqüências *query*. O *speed-up* entre o HScan que integra o filtro sistólico e os *softwares* BLASTN e FASTA numa *workstation* SPARC-10.

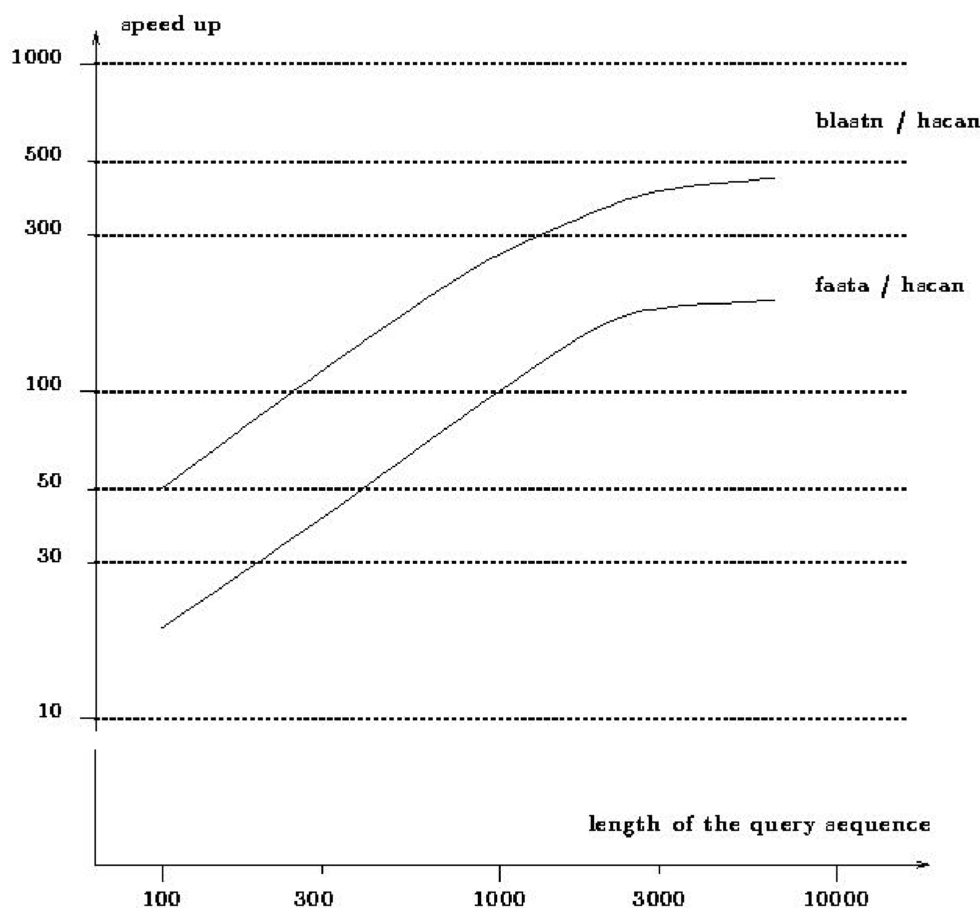


Figura 3.10 *Speed-up* entre o HScan e os *softwares* BLASTN e FASTA [27].

Portanto, pode-se perceber que as maiores seqüências obtém os melhores *speed-ups*. Isso é particularmente interessante desde que o tamanho das seqüências de DNA se torne maior, sendo adequada para técnicas avançadas na clonagem molecular e seqüenciamento de DNA.

3.7 *Arrays* Programáveis VLSI - B_SYS

O B-SYS (*Brown SYStolic Array*) [40] foi projetado principalmente para o propósito de comparação de seqüências, ainda que a flexibilidade de programação possibilite muitas outras aplicações. Esta máquina foi fabricada pela *Brown University* e testada sobre um protótipo de 10 *chips*, conduzindo a uma quantidade total de 470 processadores (47 processadores por *chip*) [41].

Um sistema do protótipo do B-SYS está funcionando desde o início de 1990. Dez *chips* B-SYS foram *linkados* juntos para formar uma pequena placa principal, 470 processadores de *array* sistólico linear têm sido usados para executar vários algoritmos combinatoriais. Esse protótipo desempenha 108 milhões de operações por segundo de 8 *bits* – MOPS (*Milion Operations per Second*) [41].

Portanto, o B-SYS é um trabalho de implementação de uma arquitetura linear SSR (*Systolic Shared Register*) para a computação sistólica programável, a qual preserva a comunicação simples de *arrays* sistólicos de propósito único.

> Os registradores compartilhados (R na figura 3.11) conduzem a uma comunicação sistólica altamente eficiente entre as unidades funcionais (F na figura 3.11) pois a computação e comunicação podem ser desempenhadas simultaneamente.

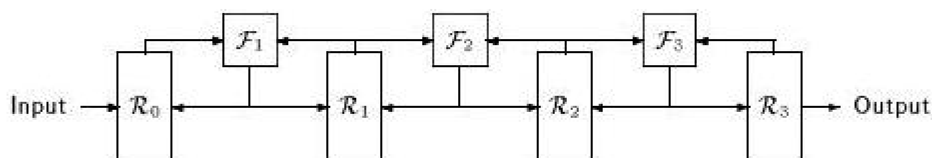


Figura 3.11 *Array* Linear SSR com fluxo de dados [41].

3.8 Arquiteturas Reconfiguráveis

Sabe-se que as pesquisas e os estudos biológicos caminham numa rápida evolução, logo, esta área precisa de tecnologia que realmente consiga acompanhar seu enorme desenvolvimento. Diante desse fato, possivelmente, pesquisadores, cientistas biológicos e projetistas de *hardware* foram impulsionados a estudar e averiguar as funcionalidades da arquitetura reconfigurável utilizando a tecnologia FPGA nesta área.

Essa tecnologia, FPGA, ainda é uma área que tem muito a ser explorada, como conseqüência, pode-se obter muitas vantagens desses recursos que ainda estão sendo testados e comprovados pelos pesquisadores e projetistas de *hardware*.

São muitas as vantagens da importância da utilização da tecnologia na área biológica, pois, os FPGAs possuem dois grandes campos de utilização:

- **Primeiro**: rápido desenvolvimento de protótipos de circuitos (*Rapid Prototyping*) [53]. A possibilidade de reprogramação no próprio circuito acarreta uma diminuição significativa no tempo de desenvolvimento devido à agilidade no processo de simulação/teste/depuração/alteração do projeto. O projetista de *hardware* também tem a possibilidade de verificar o funcionamento do circuito através de simulação funcional e temporal, já considerando os tempos de atraso gerados pela lógica resultante do processo de compilação; e
- **Segundo**: é a computação reconfigurável, que representa sistemas em que a arquitetura interna pode ser modificada por *software* em tempo real. Os FPGAs oferecem poder computacional que os transformam em dispositivos mais adequados para aplicações que envolvem algoritmos onde uma rápida adaptação de entrada é requerida.

Pode-se citar algumas das características que provavelmente tenha motivado a área biológica a optar pelo uso dessas tecnologias. A tolerância de falhas é um fator marcante, pois esta área precisa obter maior número de acertos e precisão em suas pesquisas. Como exemplo, cita-se os campos de estudos relacionados ao Genoma Humano. Desta maneira, espera-se ter menor probabilidade de erros nos testes realizados.

Outro fator é a reconfiguração parcial/dinâmica, que permite aos projetistas de *hardware* conseguir a programação e reconfiguração nos circuitos FPGAs mais modernos, praticamente em tempo real. Com isso, tem-se ganho de tempo para a reconfiguração das aplicações biológicas, assim, os cientistas dessa área realizam suas pesquisas com maior rapidez.

Por outro lado, a programação realizada nos circuitos convencionais não permite a reconfiguração em tempo real e também não podem ser reprogramados, sendo únicos e exclusivos para uma determinada aplicação.

A característica fundamental dos circuitos FPGAs é a sua reutilização. Ainda que circuitos FPGAs tenham um preço elevado considerado por alguns pesquisadores da área [52], tem-se a vantagem da sua reutilização, o que favorece para a economia desses recursos utilizados.

A presente seção retrata a abordagem de dois computadores maciçamente paralelos que utilizam arquiteturas reconfiguráveis: NGEN e POLYP. Destaca-se o alinhamento de seqüências sobre a arquitetura *Cray* MTA-2 e uma abordagem em *hardware* para algoritmos de comparação de seqüências.

3.8.1 NGEN (*eNGiNE*)

NGEN é uma plataforma de *hardware* para computação maciçamente paralela, permitindo sua configuração, em baixo nível, pelo usuário. Baseado na tecnologia FPGA, é reconfigurável para cada problema nos níveis elementares de processamento digital e de comunicação. Mais de 10.000 processadores podem ser configurados pelo usuário, com topologias de conexão variando de simples *arrays* lineares a hipercubos de maiores dimensões.

O grande fator da motivação do projeto NGEN foi a necessidade de realização de demoradas simulações de grandes populações de moléculas em interação, para o estudo dos princípios básicos do processamento criativo de informação, tipicamente encontrado na evolução natural.

A arquitetura NGEN sobressai-se pelo uso de acessos maciçamente paralelos à memória, em conjunto com variadas conexões entre FPGAs e a capacidade de configuração minuciosa da topologia global de interconexão.

Os principais objetivos do projeto da arquitetura NGEN foram [52]:

- Um grande número de processadores configuráveis (10.000 ou mais);
- Interface configurável com a *workstation* hospedeira;
- Interconexão entre processadores transparente aos limites de *chip* e placa;
- Grande quantidade de memória distribuída, ou seja, permite a extensão do tamanho dos problemas sem acarretar "gargalos"; e
- Comunicação global flexível entre processadores para possibilitar arquiteturas de maiores dimensões.

3.8.1.1 O *Hardware* do NGEN

Basicamente, o *hardware* consiste em 18 cópias de uma placa, para o barramento VME (*Versa Module Europe*) [52], especialmente projetada, juntamente com uma placa da *workstation* RISC (*Reduced Indtruction Set Computer*) (HP743i) [94], todas conectadas aos *slots* de um *backplane* VME, ou seja, é uma posição na qual

uma placa pode ser ligada ao *backplane,* que é uma placa de circuito impresso que possui as trilhas de intercomunicação entre diversas outras placas.

Uma característica importante da arquitetura NGEN é a colocação de memórias estáticas SRAM de alta velocidade em quase todas as conexões entre *chips*, conforme pode ser observado na figura 3.12.

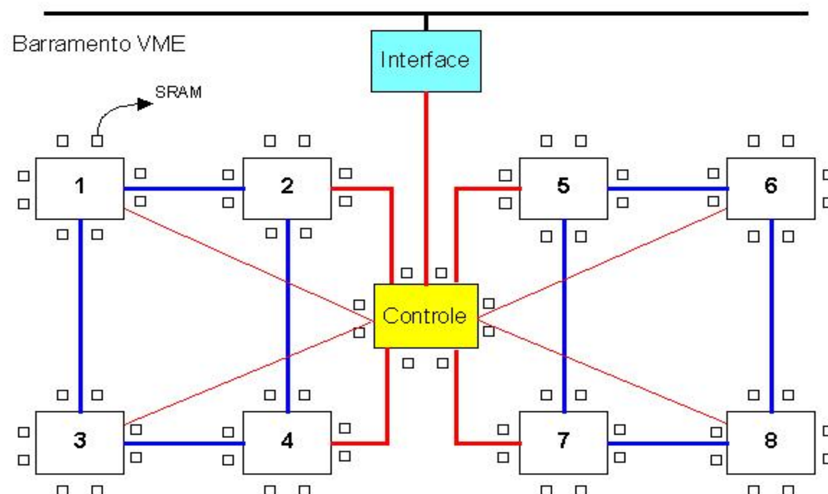


Figura 3.12 Visão da placa de FPGAs agentes e FPGA de controle [52].

As 18 placas configuráveis perfazem um total de 162 FPGAs e podem ser conectadas por cabo, a fim de constituírem a arquitetura global desejada, havendo 288 linhas de dados externas em cada uma delas.

3.8.1.2 Implementando Multiprocessadores com FPGAs

Um ambiente interativo para o NGEN foi desenvolvido em C++, sendo baseado em um *driver*, para o barramento VME, integrado ao *kernel* do UNIX da *workstation* hospedeira.

Os FPGAs escolhidos para compor a estrutura do NGEN provêem um arranjo de 18 x 18 blocos lógicos configuráveis - CLBs (*Configurable Logic Blocks*) [52]. A determinação da família de FPGAs que seriam empregadas como processadores no NGEN baseou-se em características desejáveis, tais como a presença de memória no *chip* dos FPGAs e a complexidade conveniente dos CLBs.

Os FPGAs são configuráveis tanto em termos de seus blocos lógicos quanto no que se refere às suas interconexões. Ferramentas de alto nível para projeto lógico

liberam o usuário da necessidade de conhecer os recursos de *hardware* disponíveis, quando da configuração do NGEN.

Para efetivar a configuração, é necessário carregar um arquivo binário na memória de configuração do FPGA em questão, sendo sua operação suspensa. Tal arquivo é gerado pela compilação de um arquivo simbólico de configuração, que é descrito pelo usuário através do *software* disponível.

A reconfiguração parcial não é suportada pelos FPGAs usados no NGEN. Os CLBs podem ser configurados como funções combinatoriais, além de fornecerem suporte à lógica de *carry*, ensejando sua utilização para operações aritméticas de alta velocidade.

Na comunidade houve pesquisas que contribuiram para o desenvolvimento da resolução dos problemas aplicados ao sistema NGEN. O *hardware* do sistema NGEN foi colocado em operação em 1994, e recentemente tem sido usado para executar grandes simulações individuais do conjunto de moléculas. Essas simulações forneceram novas introspecções na interação entre a seleção e a difusão na evolução molecular cooperativa, que podem ser encurtadas de várias semanas para alguns minutos com ajuda do auto-teste desenvolvido para o sistema NGEN.

Esses desenvolvimentos são acompanhados por cálculos que funcionam nas estações de trabalhos, a fim de determinar a variedade do resultado final.

3.8.2 POLYP (*Parallel OnLine PolYmer Processing*)

A sigla POLYP refere-se ao processamento paralelo o*n-line* de polímeros. POLYP é a segunda geração de computadores maciçamente paralelos reconfiguráveis. Sua construção foi baseada em FPGAs microrreconfiguráveis e alta densidade de memória distribuída adicional [83].

Enquanto o NGEN permite o estudo dos efeitos de interações em populações de moléculas através de um *hardware* configurável pelo usuário, o POLYP possibilita o estudo de tais efeitos em indivíduos das populações, por meio da reconfiguração dinâmica do *hardware* local.

O projeto do computador POLYP difere em vários aspectos do projeto do NGEN, permitindo seus FPGAs a microrreconfiguração dinâmica. Em cada placa, há oito FPGAs agentes com seis grandes unidades de SRAM com 4 *Mbits* cada, localmente conectadas. Embora mais de 20 placas possam ser colocadas no mesmo *backplane*

VME, até o presente momento no máximo 12 placas foram acopladas simultaneamente, e os resultados foram positivos.

Para o controle *on-line* dos oito FPGAs agentes, dois FPGAs distribuidores configuráveis pelo usuário foram agregadas à placa. Os distribuidores, por sua vez, têm total autonomia para o controle das agentes. Quatro *transceivers* para 20 fibras óticas estão também em cada placa, de forma que a mesma é conectada por dois *crossbars* a 80 fibras óticas unidirecionais, possibilitando flexíveis esquemas de roteamento. A figura 3.13 ilustra a estrutura das placas que compõem o POLYP.

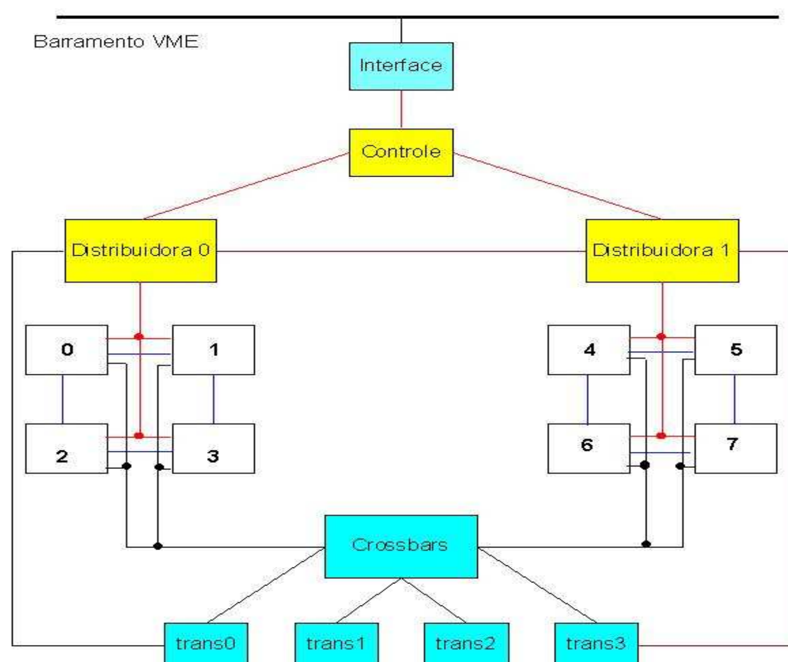


Figura 3.13 - Esquema simplificado de uma placa do POLYP [83].

Diferentemente do NGEN, a interface com o barramento VME foi implementada também com um FPGA que, além de aprimorar a interface propriamente dita, efetua o controle da temperatura dos FPGAs agentes.

As principais funções de controle são executadas por um segundo FPGA, denominado FPGA de controle. Este FPGA provê a interface com o sistema operacional da *workstation* hospedeira, assim como no NGEN. O segundo nível na hierarquia de controle é composto por dois FPGAs distribuidores, cada um controlando quatro FPGAs agentes.

Manteve-se a capacidade de implementar as mais variadas topologias de interconexão, como no NGEN, de modo que redes bidimensionais, tridimensionais e de maiores dimensões são factíveis. Os oitos FPGAs agentes em cada placa são agrupados em constelações de quatro, conectadas em forma de quadrado.

Mecanismos avançados de comunicação ótica tornam possível uma largura de banda de 2 *Gbits/s*. Os dois *crossbars* auxiliam na comunicação entre os FPGAs agentes, de forma a prover a cada FPGA o acesso a todas as fibras óticas, desde que as agentes sejam configuradas adequadamente.

Em suas pesquisas, *Tangen* [83] estendeu o tamanho das populações genéticas tendo as configurações armazenadas nos *chip*s externos da SRAM e nos FPGAs. Isto permitiu que um maior espaço virtual fosse simulado no *hardware*. Um *array* de 8 placas, cada um com 8 FPGAs agentes foram usados nas experiências com o sistema POLYP. Segundo *Tangen*, a evolução desse *hardware* foi muito difícil.

Recentemente, *Tangen* tem realizado pesquisas que possui grande variedade de populações de moléculas que podem ser configuradas por dispositivos que são utilizados para funções especificas, com o objetivo de conseguir elevado nível de desempenho nessas pesquisas. Funções essas que podem evoluir em um ambiente em tempo real.

3.8.3 Alinhamento de Seqüências sobre o *Cray* MTA-2

Esta seção apresenta algumas características da arquitetura *Cray* MTA (*Multithreaded Architecture*), segundo Vianna [87] e como elas influenciam no algoritmo padrão de alinhamento de seqüências de DNA.

Essas características foram relacionadas com as características da arquitetura híbrida SIMD (*Single Instruction Single Data*) e MIMD (*Multiple Instruction Multiple Data*) para amenizar a deficiência do limite do tamanho das seqüências de consulta, na arquitetura híbrida, com a utilização das placas *Systola*1024 e *Fuzion*150 para a componente SIMD, enquanto que a componente MIMD é estruturada em um *Cluster* de 16 PCs.

Algumas variações do algoritmo de programação dinâmica foram implementadas no *Cray*, com relação a diferentes ordens de atualização da matriz, devendo, portanto ter concentração na implementação que realiza atualização da matriz na ordem antidiagonal.

Esta especialização foi escolhida devido à facilidade de relacionamento com a técnica empregada pelas arquiteturas *Systola* e *Fuzion* [87], as quais executam a computação das células da matriz da mesma forma. As arquiteturas *Systola* e *Fuzion* são melhor descritas em [46].

3.8.4 Estrutura Sistólica para Algoritmos de Comparação de Seqüências

Esta seção tem por objetivo apresentar brevemente uma visão geral de uma estrutura sistólica para algoritmos de comparação de seqüências genéticas baseados em programação dinâmica.

Segundo Carvalho em [11], os objetivos dessa estrutura sistólica para comparação de seqüências foram:

- Propor, implementar e validar uma solução paralela baseada em *hardware* para o problema de comparar seqüências utilizando o algoritmo de *Smith-Waterman*; e
- Realizar comparações de desempenho com outras implementações do algoritmo, seqüenciais e paralelas.

No desenvolvimento dessa estrutura sistólica para comparação de seqüências foi utilizada a placa APEX PCI *Development Board* do fabricante *Altera*, contendo o FPGA APEX EP20K400EFC672. Essa placa foi instalada em um computador da *Dell* com um processador Pentium IV. A ferramenta de síntese adotada foi o *Quartus* II da própria *Altera* [11].

A implementação dessa estrutura sistólica em *hardware* foi baseada no algoritmo de comparação local de duas seqüências proposto, por *Smith-Waterman*, de recorrência com penalidade constante para os espaços, o qual é representado pela equação 3.3.

$$\mathbf{S_{i,j} = max} \begin{cases} \mathbf{S_{i-1,j} + w_1} \\ \mathbf{S_{i-1,j-1} + sub} \\ \mathbf{S_{i,j-1} + w_1} \end{cases} \qquad \text{Equação 3.3 [11]}$$

onde $\mathbf{w_1}$ e **sub** representam respectivamente, um *gap penalty* e uma substituição de caracteres das seqüências comparadas; $\mathbf{S_{i,0} = 0}$, i = 0, 1,..., | x | e $\mathbf{S_{j,0} = 0}$ j = 1, 2,..., | y |.

Embora não seja a equação mais correta do ponto de vista biológico, essa equação foi escolhida para a implementação feita nesse projeto, tendo como objetivo atacar o problema com uma abordagem mais simples, e uma vez conhecidas suas peculiaridades, estender a solução obtida para situações mais complexas [11].

3.8.4.1 Descrição da Implementação da Estrutura Sistólica para Algoritmos de Comparação de Seqüências

Para a comparação de seqüências baseadas no algoritmo de *Needleman* e *Wunsch*, as soluções computacionais não oferecem um bom desempenho, mesmo com processadores mais velozes, pois este algoritmo tem complexidade de tempo quadrática [11]. Logo, a computação paralela tem sido empregada na tentativa de reduzir o tempo de execução deste algoritmo.

Com *n* processadores paralelos, é possível aumentar a velocidade de processamento por um fator *n*, desde que o algoritmo estritamente seqüencial usado possa ser reescrito por operações paralelas simultâneas. Como a comparação de seqüências é baseada em computações análogas para cada posição de um vetor linear, pode-se conjecturar que algoritmos paralelos mais eficientes possam ser elaborados para problemas de comparação de seqüências.

Existem basicamente duas formas de aplicar paralelismo ao problema de comparação de seqüências [42]:

1. Paralelizando a operação de comparação: neste caso, todos os processadores cooperam para determinar o *score* de cada célula da matriz de similaridades; e
2. Paralelizando o processo de comparação: neste caso, cada processador realiza um número de comparações de forma independente, ou seja, calcula o *score* das células de porções menores das seqüências comparadas.

Desta maneira, o segundo método é o mais utilizado, porém o primeiro método é mais apropriado para computadores SIMD onde todos os processadores executam a mesma instrução ao mesmo tempo, e a velocidade de comunicação é rápida comparada

à de processamento. Sistemas SIMD geralmente possuem centenas de processadores lentos, mas com baixo custo de comunicação. Esse é um tipo de arquitetura nas quais as estruturas sistólicas se encaixam totalmente.

O segundo método é mais adequado para MIMD, onde cada processador é significativamente mais poderoso que um processador SIMD, e os processadores executam suas instruções de forma independente, em vez de cooperarem para comparar cada seqüência do banco de dados [11].

3.8.4.2 Performances da Estrutura Sistólica para Algoritmos de Comparação de Seqüências

A implementação dessa estrutura sistólica em *hardware* para comparação de seqüências se deu apenas em nível de síntese, desta maneira os resultados apresentados foram obtidos a partir de simulações feitas na ferramenta de desenvolvimento do fabricante. Entretanto, essas ferramentas forneceram resultados bastante precisos, não só em termos de funcionalidade do sistema, como também em termos de velocidade final do sistema indicando caminhos críticos e atrasos dos sinais.

A figura 3.14 ilustra o cálculo do valor da diagonal na arquitetura da estrutura sistólica para comparação de seqüências onde a implementação 1 utiliza dois somadores, um comparador e um multiplexador, enquanto a implementação 2 utiliza um somador, um comparador e um multiplexador.

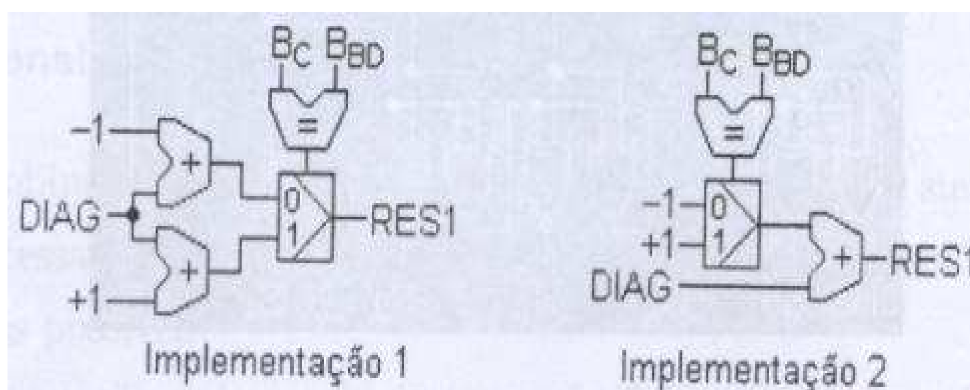


FIGURA 3.14 Cálculo do valor da diagonal da estrutura sistólica para comparação de seqüências genéticas [11]

Independente da quantidade de recursos internos do FPGA requirida pelo somador e comparador, a implementação 2 é mais otimizada que a 1 [11]. Entretanto, a implementação 1 é mais rápida que a 2, pois no circuito 1 as somas e a comparação

estão sendo feitas simultaneamente, enquanto que na implementação 2 o multiplexador tem que aguardar o resultado da comparação antes de definir qual será o valor fornecido ao somador. Como a diferença de performance é baixa e o principal critério é o da economia, o circuito 2 foi o escolhido. A esse resultado parcial foi dado um nome interno de RES1.

Além disso, a comunicação com o ambiente externo não é crítica como em um sistema de tempo real, o que poderia prejudicar os resultados da simulação. Para a validação não bastava apenas conhecer os alinhamentos obtidos na comparação entre duas seqüências, mas também conhecer toda a matriz de similaridade para verificar se as otimizações (paralelizações já mencionadas) foram feitas corretamente.

Assim, os testes foram todos feitos com seqüências fictícias cuja matriz de similaridade era conhecida ou poderia ser construída facilmente. Para o teste de desempenho, vetores de diferentes tamanhos foram sintetizados e suas freqüências máximas de operação foram obtidas a partir de informações fornecidas pela ferramenta de síntese. A tabela 3.3 ilustra os testes realizados com suas respectivas freqüências.

Tabela 3.3.Desempenho obtido através da ferramenta de síntese [11].

Quantidade de células do vetor	Quantidade de elementos lógicos	Freqüência máxima de	Média de elementos utilizados por célula
5	146	122,49 MHz	29,2
10	507	77,54 MHz	50,7
15	900	70,35 MHz	60
20	1292	64,94 MHz	64,6
25	1758	62,63 MHz	70,32
30	2222	56,45 MHz	74,06
40	3204	56,27 MHz	80,1
50	4403	56,1 MHz	88,06

Em cada uma dessas sínteses, a ferramenta de desenvolvimento da ALTERA analisou os caminhos críticos e os diversos atrasos (tempo que a mudança em sinal leva para gerar outros sinais estáveis) e determinou a freqüência máxima de operação do circuito. Além disso, a quantidade de elementos lógicos do FPGA utilizados por vetores de comprimentos diversos também foi obtida.

Pode-se perceber que quanto maior a quantidade de células do vetor, menor sua freqüência máxima de operação.

Portanto, este projeto mostrou como um *hardware* dedicado, baseado em uma estrutura sistólica, é capaz de linearizar a complexidade temporal e reduzir bastante o tempo de processamento desses algoritmos. *Softwares* como o BLAST ou FASTA são extensivelmente usados para desempenhar a visualização dos bancos de dados biológicos. Eles têm sido projetados para rodar em computadores padrão, isto é, máquinas *Von Neuman*, e incluem técnicas para acelerar os processos.

As arquiteturas especiais brevemente analisadas neste trabalho, NGEN, POLYP, *Cray* MTA-2 e estruturas sistólicas para algoritmos de comparação de seqüências baseados em programação dinâmica, mostraram a utilização de FPGAs na computação maciçamente paralela. Ambos projetos permitem antever o potencial de utilização de dispositivos programáveis como importante bloco de construção de sistemas de computação das mais variadas espécies.

Capítulo

4

ALGORITMOS DE ALINHAMENTO GLOBAL E LOCAL EM C

4.1 Considerações Iniciais

O principal objetivo do alinhamento de seqüências de genes ou proteínas consiste em comparar estas mesmas seqüências, para encontrar semelhanças e diferenças entre várias espécies [88].

As seqüências constituem modelos que representam a forma como em um gene ou em uma proteína os aminoácidos que os compõem se encontram combinadas. A partir dos resultados obtidos pela comparação de seqüências poder-se-à eventualmente extrair conclusões importantes que permitam conhecer melhor o ambiente que nos rodeia e melhorar a qualidade de vida do homem [88].

Este capítulo tem por objetivo apresentar o desenvolvimento dos algoritmos de alinhamento global e local, aqui denominados respectivamente de AGM e ALM,

designados a analisar e comparar seqüências de DNA, abordando suas implementações, assim como as análises de desempenho.

4.2 AGM – Algoritmo Global - Implementação Morony

Conforme descrito no capítulo 2, o alinhamento global conhecido como algoritmo de *Needleman-Wunsch* [1] consiste em encontrar a similaridade global entre duas seqüências e os valores globais requerem que o alinhamento comece no início e se estende até o final de todo o comprimento da seqüência. Observa-se a seguir, o principal trecho para o preenchimento total da matriz de programação dinâmica para o alinhamento global implementado na linguagem de programação C, onde se utilizou o compilador Turbo C.

```
void matriz()  /* Calculo e Preenchimento da Matriz de Programação
Dinâmica para o Alinhamento Global – Morony-Global*/

{
      int XLINHA, XCOLUNA;

      long int XLIN, XCOL;

      printf("\n\nInforme o valor para caracteres IGUAIS: ");
      scanf("%d",&IGUAL);

      printf("\n\nInforme o valor para caracteres DIFERENTES: ");
      scanf("%d",&DIFERENTE);

      printf("\n\nInforme o valor referente ao GAP: ");
      scanf("%d",&ESPACO);

      abrearq();
      W=ESPACO;
      XCOLUNA = 0;

      M.valor  = 0;
      M.linha  = 0;
      M.coluna = 0;
      fseek(Mptr,0,2);
      fwrite(&M,sizeof(M),1,Mptr);

      fseek (Lptr,0,0);
      while(fread (&L,sizeof(L),1,Lptr)==1)
      {
            XCOLUNA++;
            M.valor  = W;
            M.linha  = 0;
            M.coluna = XCOLUNA;
            W=W+ESPACO;
            fseek(Mptr,0,2);
            fwrite(&M,sizeof(M),1,Mptr);
      }
```

```
        W=ESPACO;
        XLINHA=1;
        fseek (Cptr,0,0);
        while(fread (&C,sizeof(C),1,Cptr)==1)

        {
              XLIN = ftell(Cptr);
              XCOLUNA=0;
              fseek (Lptr,0,0);

              M.valor = W;
              W = W+ESPACO;
              M.linha  = XLINHA;
              M.coluna = XCOLUNA;
              XCOLUNA++;
              fseek(Mptr,0,2);
              fwrite(&M,sizeof(M),1,Mptr);

              while(fread (&L,sizeof(L),1,Lptr)==1)

              {
                    XCOL = ftell(Lptr);
                    M.valor  = xvalor(XLINHA,XCOLUNA);
                    M.linha  = XLINHA;
                    M.coluna = XCOLUNA;

                    fseek(Lptr,XCOL,0);
                    XCOLUNA++;

                    fseek(Mptr,0,2);
                    fwrite(&M,sizeof(M),1,Mptr);
              }

              fseek(Cptr,XLIN,0);

              XLINHA++;
        }
}
```

No trecho do algoritmo acima, é possível realizar o preenchimento total da matriz, ou seja, inicializa-se o cálculo da primeira linha e primeira coluna seguida do restante da matriz, conforme se ilustra na figura 2.13. Contudo, a figura 4.1 apresenta a execução do cálculo da matriz do algoritmo global (aqui denominado de AGM).

TC.EXE

		G	A	A	T	T	G	C	A
	0	-4	-8	-12	-16	-20	-24	-28	-32
G	-4	5	1	-3	-7	-11	-15	-19	-23
G	-8	1	2	-2	-6	-10	-6	-10	-14
A	-12	-3	6	7	3	-1	-5	-9	-5
T	-16	-7	2	3	12	8	4	0	-4
C	-20	-11	-2	-1	8	9	5	9	5
A	-24	-15	-6	3	4	5	6	5	14
T	-28	-19	-10	-1	8	9	5	3	10
G	-32	-23	-14	-5	4	5	14	10	6

Figura 4.1 Cálculo da matriz de programação dinâmica no alinhamento Morony-Global implementada na linguagem de programação C.

Nesta última figura, utilizavam-se duas seqüências com valores aleatórios, que representam respectivamente **GAATTGCA** (horizontal) e **GGATCATG** (vertical), ou seja, uma matriz de 8x8.

A figura 4.2 ilustra a tela da execução do procedimento *traceback* no algoritmo de alinhamento global onde os valores utilizados para caracteres iguais, diferentes e *gap*, correspondem respectivamente a: 5, -3 e -4. Logo, o alinhamento ótimo ou melhor alinhamento equivale ao valor 7.

```
TC.EXE

A   -
C   G
G   G
T   T
T   A
T   -
T   T
A   A
A   -
G   G
Alinhamento Otimo: 7
```

Figura 4.2 Resultado obtido do alinhamento ótimo no alinhamento global em C.

4.3 ALM – Algoritmo Local - Implementação Morony

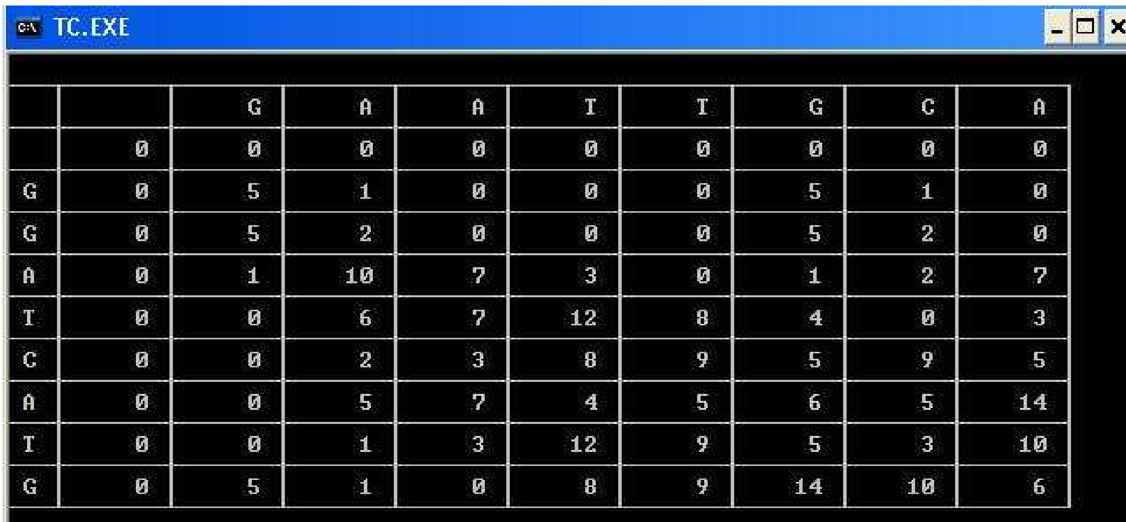
O algoritmo de alinhamento local, conhecido como algoritmo de *Smith-Waterman* não precisa alinhar as seqüências inteiras pois os valores locais requerem identificação da região mais similar entre duas seqüências. Da mesma maneira, apresenta-se o principal trecho para o preenchimento total da matriz de programação dinâmica para o alinhamento local que também foi implementado na linguagem de programação C (aqui denominado de ALM).

```
void matriz()/* Calculo e Preenchimento da Matriz de Programação
Dinâmica para o Alinhamento Local – Morony-Local*/
{
      int XLINHA, XCOLUNA;
      long int XLIN, XCOL;

      printf("\n\n\n\n\n");
      printf("\n\nInforme o valor para caracteres IGUAIS    : ");
      scanf("%d",&IGUAL);
      printf("\n\nInforme o valor para caracteres DIFERENTES: ");
      scanf("%d",&DIFERENTE);
      printf("\n\nInforme o valor referente ao GAP          : ");
      scanf("%d",&ESPACO);

     TSi=0;
      TSf=0;
      TST=0;
      TMi=0;
      TMf=0;
      TMT=0;

      tempo();

      abrearq();
      W=ESPACO;
      XCOLUNA = 0;

      M.valor  = 0;
      M.linha  = 0;
      M.coluna = 0;
      fseek(Mptr,0,2);
      fwrite(&M,sizeof(M),1,Mptr);

      fseek (Lptr,0,0);
      while(fread (&L,sizeof(L),1,Lptr)==1)
      {
            XCOLUNA++;
            M.valor  = 0;
            M.linha  = 0;
            M.coluna = XCOLUNA;
            W=W+ESPACO;
            fseek(Mptr,0,2);
            fwrite(&M,sizeof(M),1,Mptr);
      }

      W=ESPACO;
```

```
        XLINHA=1;
        fseek (Cptr,0,0);
        while(fread (&C,sizeof(C),1,Cptr)==1)
        {
              XLIN = ftell(Cptr);
              XCOLUNA=0;
              fseek (Lptr,0,0);

              M.valor  = 0;
              M.linha  = XLINHA;
              M.coluna = XCOLUNA;
              XCOLUNA++;
              fseek(Mptr,0,2);
              fwrite(&M,sizeof(M),1,Mptr);

              while(fread (&L,sizeof(L),1,Lptr)==1)
              {
                    XCOL = ftell(Lptr);
                    M.valor  = xvalor(XLINHA,XCOLUNA);
                    M.linha  = XLINHA;
                    M.coluna = XCOLUNA;

                    fseek(Lptr,XCOL,0);
                    XCOLUNA++;
                    fseek(Mptr,0,2);
                    fwrite(&M,sizeof(M),1,Mptr);
              }
              fseek(Cptr,XLIN,0);
              XLINHA++;
        }
        tempo();
}
```

Ao comparar o trecho do algoritmo acima descrito com o algoritmo global é possível notar algumas diferenças, ou seja, o preenchimento da primeira linha e da primeira coluna recebe o valor zero e por este motivo, esta matriz de programação dinâmica para o alinhamento local não admite valores negativos, conforme descrito na seção 2.4.

Como exemplo, a figura 4.3 ilustra a tela da execução do algoritmo de alinhamento local, sendo composta pelas seqüências **GAATTGCA** (horizontal) e **GGATCATG** (vertical), isto é, uma matriz de 8x8.



		G	A	A	T	T	G	C	A
	0	0	0	0	0	0	0	0	0
G	0	5	1	0	0	0	5	1	0
G	0	5	2	0	0	0	5	2	0
A	0	1	10	7	3	0	1	2	7
T	0	0	6	7	12	8	4	0	3
C	0	0	2	3	8	9	5	9	5
A	0	0	5	7	4	5	6	5	14
T	0	0	1	3	12	9	5	3	10
G	0	5	1	0	8	9	14	10	6

Figura 4.3 Cálculo da matriz de programação dinâmica no alinhamento MoronyLocal implementada na linguagem de programação C.

Deste modo, o alinhamento ótimo obtido da matriz ilustrada acima recebe valor 14, conforme visualiza-se na figura 4.4.

Figura 4.4 Resultado obtido do alinhamento ótimo no alinhamento local em C.

Neste capítulo, o principal objetivo das implementações dos algoritmos global e local (chamados de AGM e ALM) é assimilar os detalhes de cada um dos mesmos, com intuito de extrair os pontos críticos para uma futura implementação em *hardware*.

4.4 Performances dos Algoritmos Global e Local

O tamanho das seqüências utilizadas na execução dos algoritmos global e local, implementados na linguagem C, varia de 10 a 1000 caracteres, conforme pode ser visualizado na tabela 4.1, sendo que os caracteres das seqüências foram escolhidos aleatoriamente para obter o tempo de execução medido em segundos.

Observa-se que as seqüências de tamanho 10 gastam menos de 1 (um) segundo para realizar a comparação e o alinhamento, ou seja, gastam apenas 0.220 e 0.160 milissegundos no alinhamento global e local, respectivamente. Nota-se que os valores para os tamanhos 30, 40 e 50 consumiram a mesma quantidade de tempo e após as seqüências de tamanho 300, percebe-se que o alinhamento local precisa de um tempo maior do que no alinhamento global.

Tabela 4.1 Performances dos algoritmos Global e Local em C

Tamanho das Seqüências	**Tempo em Segundos.**	
	Morony-Global	**Morony-Local**
10	0.220 (***milissegundos***)	0.160 (***milissegundos***)
20	1	1
30	2	2
40	4	4
50	9	9
100	57	60
150	227	222
300	2664	2979
400	7999	10259
500	19354	21187
600	39704	43281
1000	300184 = **83 horas**	328280 = **91 horas**

A figura 4.5 ilustra as performances obtidas com as implementações na linguagem C dos algoritmos Global e Local, de acordo com a tabela 4.1, tendo o *hardware* utilizado para realizar a execução desses algoritmos foi um computador Pentium IV 1.66 GHz, AMD Athlon (tm) XP 2000+, 240 MB de memória RAM, 256 K de *cache*, com sistema operacional *Microsoft Windows* XP.

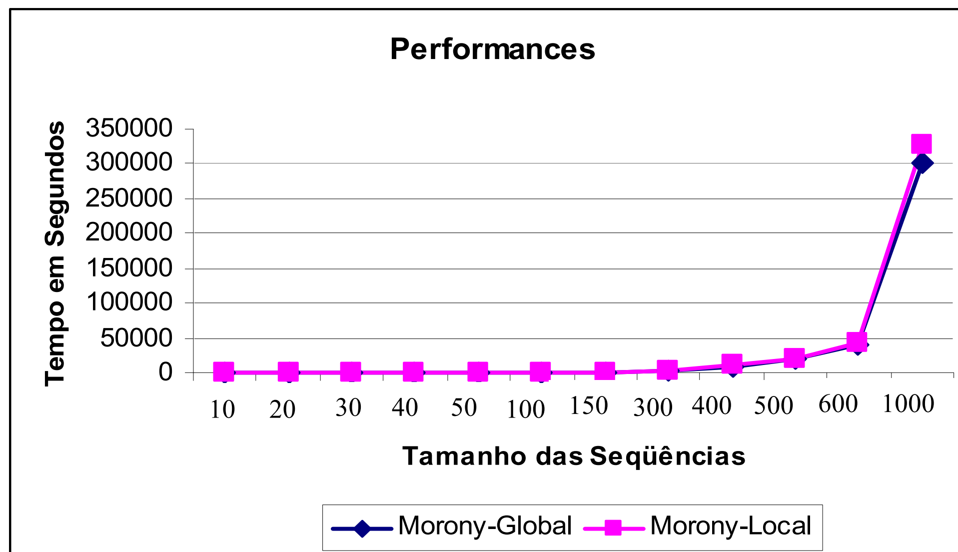


Figura 4.5 Performances das implementações em C dos algoritmos Global e Local.

Na figura 4.5 ilustra-se uma diferença do tempo a partir da seqüência de tamanho de 300 caracteres, visto que o alinhamento global é mais eficiente para o alinhamento e comparação de seqüências de DNA na implementação em C, pois precisa um tempo relativamente menor para executar o alinhamento e comparar as seqüências (que foram escolhidas aleatoriamente) na implementação em relação ao alinhamento local.

Na figura é possível observar também que o tempo para realizar o alinhamento aumenta conforme o tamanho da seqüência aumenta, e se torna crítica seqüências acima de 300 caracteres.

4.5 Performances dos Algoritmos AGM, ALM e SAMBA

Conforme descrito na seção 3.4 a arquitetura SAMBA (*Systolic Accelerator for Molecular Biological Application*) [51] é um *array* sistólico dedicado para a comparação de seqüências biológicas. Esse *hardware* acelera uma versão parametrizada do algoritmo de *Smith-Waterman* [3] permitindo a computação de alinhamentos globais e locais com ou sem *gap penalty*.

Pelo fato da arquitetura SAMBA utilizar o algoritmo de *Smith-Waterman*, é possível realizar a comparação dos dados de performances desta arquitetura com as performances obtidas nas nossas implementações dos alinhamentos Global e Local implementados em C.

Algumas das performances mostradas a seguir têm como objetivo visualizar os resultados obtidos com o SAMBA sob uma intensiva comparação num banco de proteínas, composto por um conjunto de 815 proteínas junto à versão 31 do banco SWISS-PROT[2] [44] usando o rigoroso algoritmo de *Smith* e *Waterman* com diferentes matrizes de substituição[3] e *gap penalties*.

Neste caso considerou-se a execução do *software* SSEARCH (que encontra alinhamentos locais de acordo com o algoritmo de *Smith* e *Waterman*, o qual é fornecido com o pacote de *softwares* FASTA), sob uma *workstation* DEC-Alpha 21064 de 150Mhz, segundo *Lavenier* em [48].

A versão 31 do banco SWISS-PROT contém exatamente 43.470 seqüências distribuídas sob 15.335.248 aminoácidos. As 815 seqüências representam um total de 307.400 aminoácidos [48].

A tabela 4.2 indica o tempo de pesquisa no *software* SSEARCH (em segundos) sob diferentes *workstations* implementadas na arquitetura SAMBA, ou seja, **SAMBA – DEC - Alpha 150MHz**, **SAMBA - Sun SPARC 5 110MHz, SAMBA - DEC 5000/250 40MHz**, utilizando o banco de dados SWISS-PROT cuja versão é a de número 31, com proteínas de diferentes tamanhos e ainda inclui os dados de performances obtidos dos algoritmos Morony-Global e Morony-Local, implementados na linguagem de programação C.

Tabela 4.2 Performances do tempo consumido em segundos.

Tamanho da Seqüência	**Tempo em Segundos**					
	Global	**Local**	**SAMBA**	**SAMBA DEC - Alpha 150MHz**	**SAMBA - Sun SPARC 5 110MHz**	**SAMBA - DEC 5000/250 40MHz**
10	0.220 milissegundos	0.160 milissegundos	25	57	95	182
30	2	2	25	120	239	548
100	57	60	26	350	746	1407
300	2664	2979	30	1041	2215	4054
1000	300184	328280	40	3468	7300	12920

Os dados de performances da arquitetura SAMBA nas *workstations* são aqueles obtidos no artigo "SAMBA: *Systolic Accelerators for Molecular Biological Applications*" [48] em 1996.

[2] SWISS-PROT: *Protein Knowledgebase*. Banco de dados de seqüências de proteínas.
[3] Matrizes de Substituição: desenvolvida na década de 1980 e são as mais usadas no alinhamento de seqüências protéicas [64].

Percebe-se que máquinas paralelas não estão sendo desenvolvidas para a comparação de seqüências biológicas e isso mostra os dados apresentados na implementação dos algoritmos Global e Local em C nesta dissertação são bons se comparados com as implementações das *workstations* da arquitetura SAMBA.

A figura 4.6 apresenta o gráfico obtido com os dados de performances da tabela 4.2.

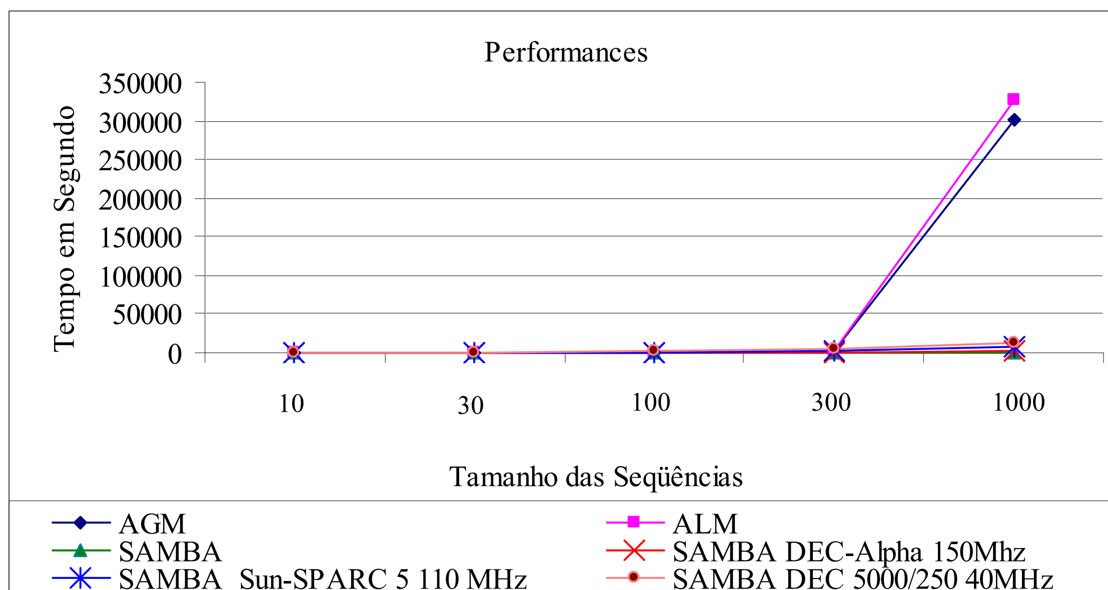


Figura 4.6 Performances do tempo consumido em segundos versus tamanho das seqüências

Nesta tabela as seqüências com tamanho até 300 caracteres quase não é possível distinguir suas performances, tornando-se visível as diferenças a partir das seqüências de tamanho superior a 300 caracteres, visto que, acima deste valor (300 caracteres) as performances dos alinhamentos Global e Local apresentam um consumo de tempo em segundos muito superior a arquitetura SAMBA, juntamente com suas representações nas diferentes *workstations*.

Percebe-se que para uma seqüência de tamanho 1.000, o tempo consumido na operação de alinhamento é maior nos algoritmos AGM e ALM (Global e Local) do que os mesmos resolvidos pela arquitetura SAMBA e suas *workstations*.

Como não é possível ter nitidez na figura 4.6 com tamanhos de seqüências até 300 caracteres, achou-se conveniente explorar melhor a visualização desta figura sendo que o objetivo é encontrar diferentes performances dos alinhamentos Global e Local juntamente com a arquitetura SAMBA sob diferentes *workstations*, conforme ilustra a figura 4.7.

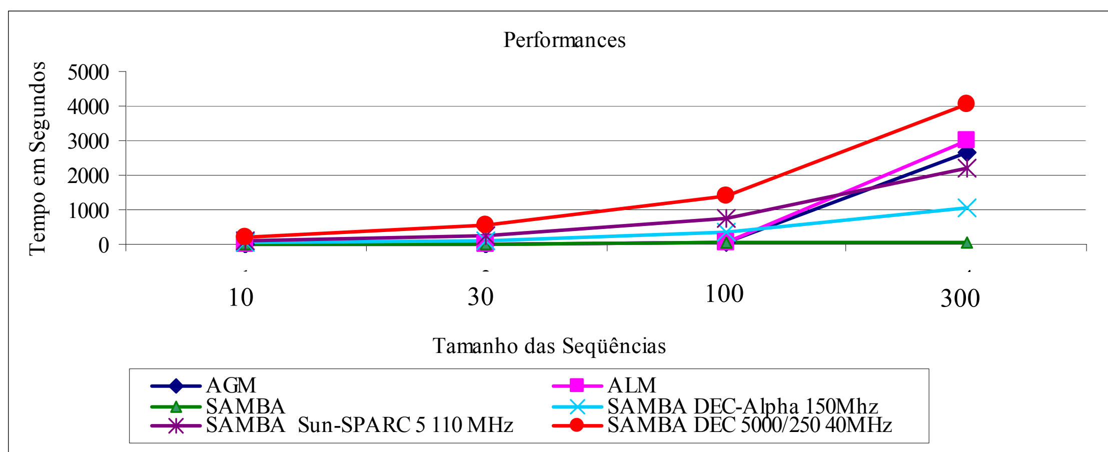


Figura 4.7 Performances dos alinhamentos com tamanhos de seqüências até 300 caracteres

Nota-se que apesar dos algoritmos Global e Local serem implementados na linguagem C e utilizarem apenas um único processador, tem performances boas se comparadas com a arquitetura SAMBA.

Lembrar que a arquitetura SAMBA contém uma *workstation*, um *array* sistólico linear composto de 32 *chips full custom* distribuídos sobre 2 (duas) placas. Um *chip* integra 4 (quatro) processadores, conduzindo a um *array* sistólico de 128 processadores e uma interface baseada em FPGA, conforme mostrou-se na figura 3.3.

Contudo, SAMBA é adequado para análises de seqüências que exigem uma grande quantidade de computação, tais como comparação banco a banco ou consulta a banco com longas seqüências. O crescimento exponencial dos bancos biológicos associados com o completo seqüenciamento do genoma torna o SAMBA uma solução interessante.

E com base nisso, mesmo SAMBA sendo uma solução interessante, decidimos implementar os algoritmos de alinhamento global e local na linguagem de programação C quanto em VHDL para saber quais seriam os dados que obteríamos e compararmos com alguns sistemas de *hardware* principalmente aqueles baseados em FPGA, mesmo sabendo que não estão sendo desenvolvidas máquinas paralelas para a comparação e o alinhamento de seqüências genéticas.

Capítulo

5

IMPLEMENTAÇÃO EM HARDWARE DOS ALGORITMOS DE SEQUENCIAMENTO

5.1 Considerações Iniciais

Com a microeletrônica nos últimos anos, é possível deparar-se com uma diversidade de dispositivos eletrônicos mais velozes, com maior capacidade para armazenamento de informações, menor consumo de energia e custos cada vez mais baixos, colocando os projetistas de computadores frente a um desafio: o balanço entre velocidade de operação dos sistemas versus generalidade de operação desses sistemas [89].

Até alguns anos atrás, a alternativa existente para conseguir alta performance era a construção de circuitos integrados dedicados para aplicações específicas, os chamados ASICs (*Application Specific Integrated Circuits*) [64]. Neste caso, a arquitetura interna dos dispositivos é otimizada e voltada para a aplicação em questão, desta maneira, consegue-se então, uma performance superior à que seria conseguida com a arquitetura de um microprocessador de propósito geral – GPP (G*eneral Purpose Processor*) [93].

A característica marcante do dispositivo de propósito geral é a sua imutabilidade, ou seja, as funções que executam estão "amarradas" ao próprio *hardware* e, uma vez construídas, não podem ser reprogramados.

Com o desenvolvimento dos circuitos integrados, surgiram os dispositivos lógicos programáveis – PLDs (*Programmable Logic Devices*) [94]. Com eles, é possível combinar a característica de alta performance do *hardware* com a generalidade e reprogramabilidade dos microprocessadores. A implementação de circuitos digitais complexos era uma ciência dominada apenas por grandes empresas ou universidades de renome internacional. Com o avanço da tecnologia surgiram os FPGAs, que são dispositivos modernos com características de reprogramabilidade [10][94].

Os FPGAs são circuitos integrados programáveis, que podem ser reconfigurados pelos projetistas quando necessário, sendo ideais para o desenvolvimento rápido de protótipos de sistemas digitais. Na realidade, um mesmo *chip* pode assumir arquiteturas e funções completamente diferentes, sem a necessidade de mudanças do mesmo em função apenas da necessidade do usuário.

Esta tecnologia inovadora está viabilizando a construção e prototipação de circuitos digitais complexos sem a necessidade de muitos recursos computacionais e financeiros. A possibilidade de implementar um circuito digital em um ambiente simplificado e de baixo custo está popularizando cada vez mais esta tecnologia.

Atualmente pode-se descrever um circuito digital para FPGA utilizando a linguagem VHDL, que tem por objetivo a descrição de *hardware* de alta performance e flexível utilizada tanto na indústria como para fins acadêmicos, isto é, a linguagem VHDL é padronizada para descrever componentes digitais. A seguir, algumas das vantagens de usar VHDL [59]:

- Reduz tempo/custo de desenvolvimento;
- Maior nível de abstração;
- Projetos independentes da tecnologia;
- Facilidade de atualização dos projetos; e
- Grande número de usuários (internacional).

Assim, este capítulo aborda as informações necessárias para apresentar as implementações dos algoritmos: global (*Needleman-Wunsch*) e local (*Smith-Waterman*) descritos em VHDL, utilizando a técnica de programação dinâmica.

São gerados dados para que seja possível fazer a comparação com outros sistemas de hardware propostos por outros autores que já implementaram os algoritmos de alinhamento global (*Needleman-Wunsch*) e local (*Smith-Waterman*).

5.2 Operações e Funções dos Algoritmos Global e Local

Com a nossa implementação dos algoritmos Global e Local na linguagem de programação C (chamados de AGM e ALM), apresentados no capítulo 4, é possível apresentar as operações e funções utilizadas nos desenvolvimentos dos mesmos.

Dadas duas seqüências, exemplificadas pelas letras A e B, conforme ilustrados na figura 5.1, cujos tamanhos correspondem a 10 caracteres respectivamente, representam as entradas nas seqüências, as quais passam por várias operações e funções que resultam na "Saída" que dá origem no alinhamento ótimo entre as seqüências A e B.

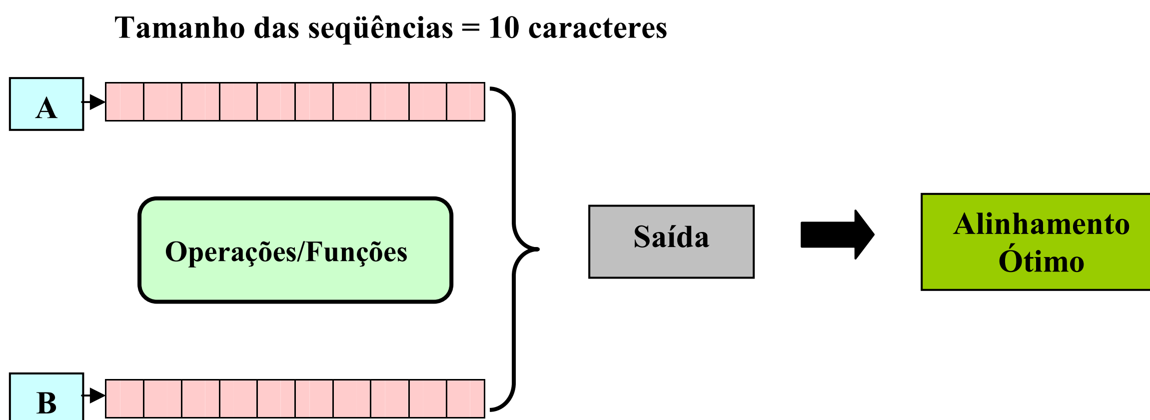


Figura 5.1 Representação das operações dos algoritmos Morony-Global e Morony-Local.

São as seguintes operações realizadas nos algoritmos Global e Local na linguagem C:

- Somador;
- Subtrator;

- Divisor;
- Comparador;
- Contador;
- Posicionador de posição - fseek (trabalhou-se com arquivos na linguagem C);
- Condicional (if - else);
- Laço (while; do - while);
- Chamador de função;
- Atribuidor de variáveis;
- Escrita no arquivo (fwrite);
- Leitura no arquivo (fread); e
- Impressão na tela (printf).

A figura 5.2 esboça um fluxograma que usa da representação gráfica, por meio de símbolos geométricos, da solução algorítmica sendo possível visualizar do início ao final a funcionalidade dos algoritmos Global e Local. Nesse fluxograma, os retângulos, quadrados e losângulos identificados representam as funções e, as setas as ligações dos algoritmos acima descritos.

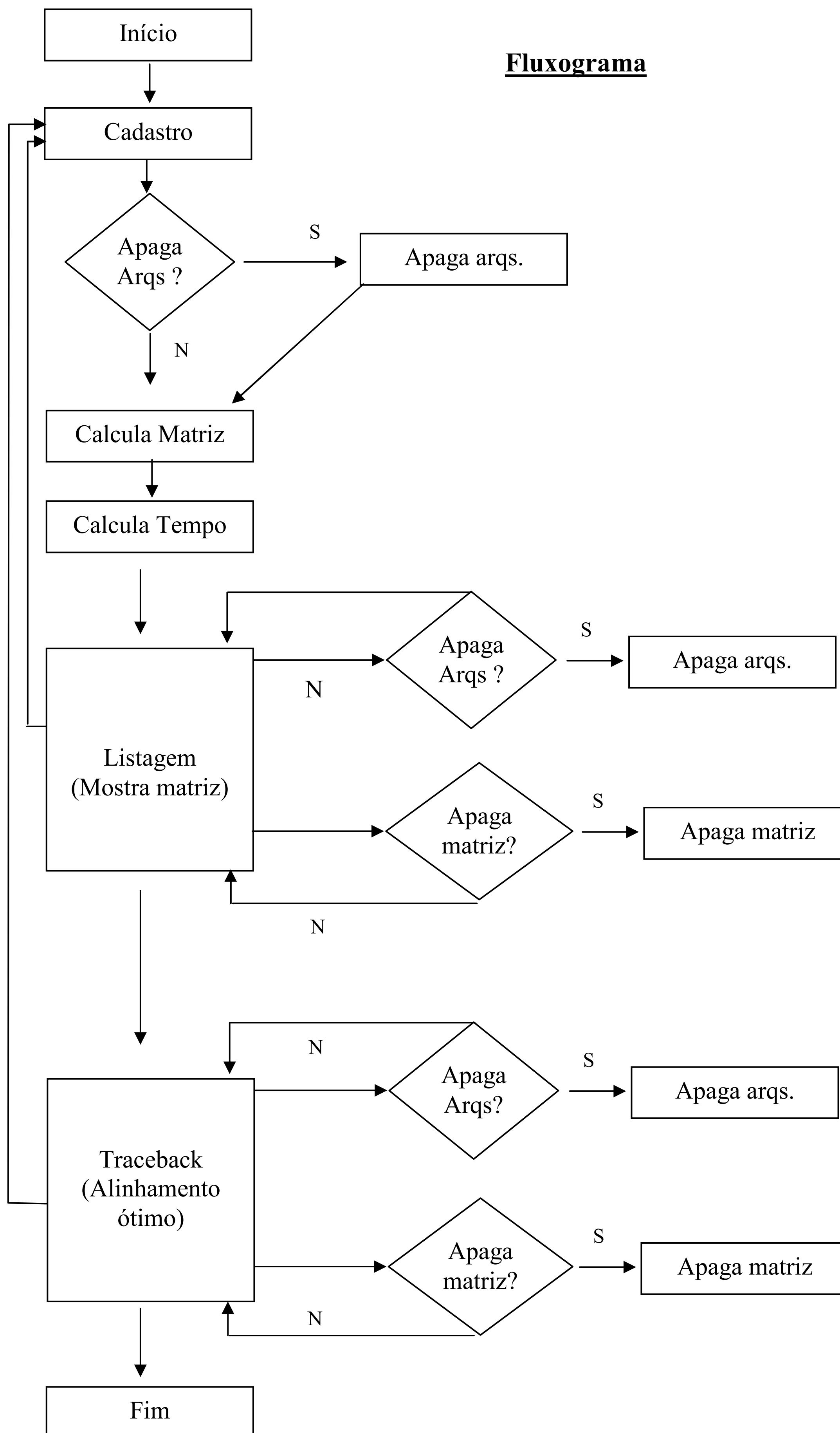


Figura 5.2 Fluxograma das operações no AGM e ALM

5.3 Implementação dos Algoritmos Global e Local em *Hardware*

Após ter mostrado todo o caminho percorrido do desenvolvimento dos algoritmos Global e Local (ver apêndice A e figura 5.2), até chegar aos procedimentos em alto nível, o objetivo desta etapa é apresentar a implementação da estrutura dos algoritmos acima mencionados. A figura 5.3 representa graficamente a estrutura voltada para o alinhamento global.

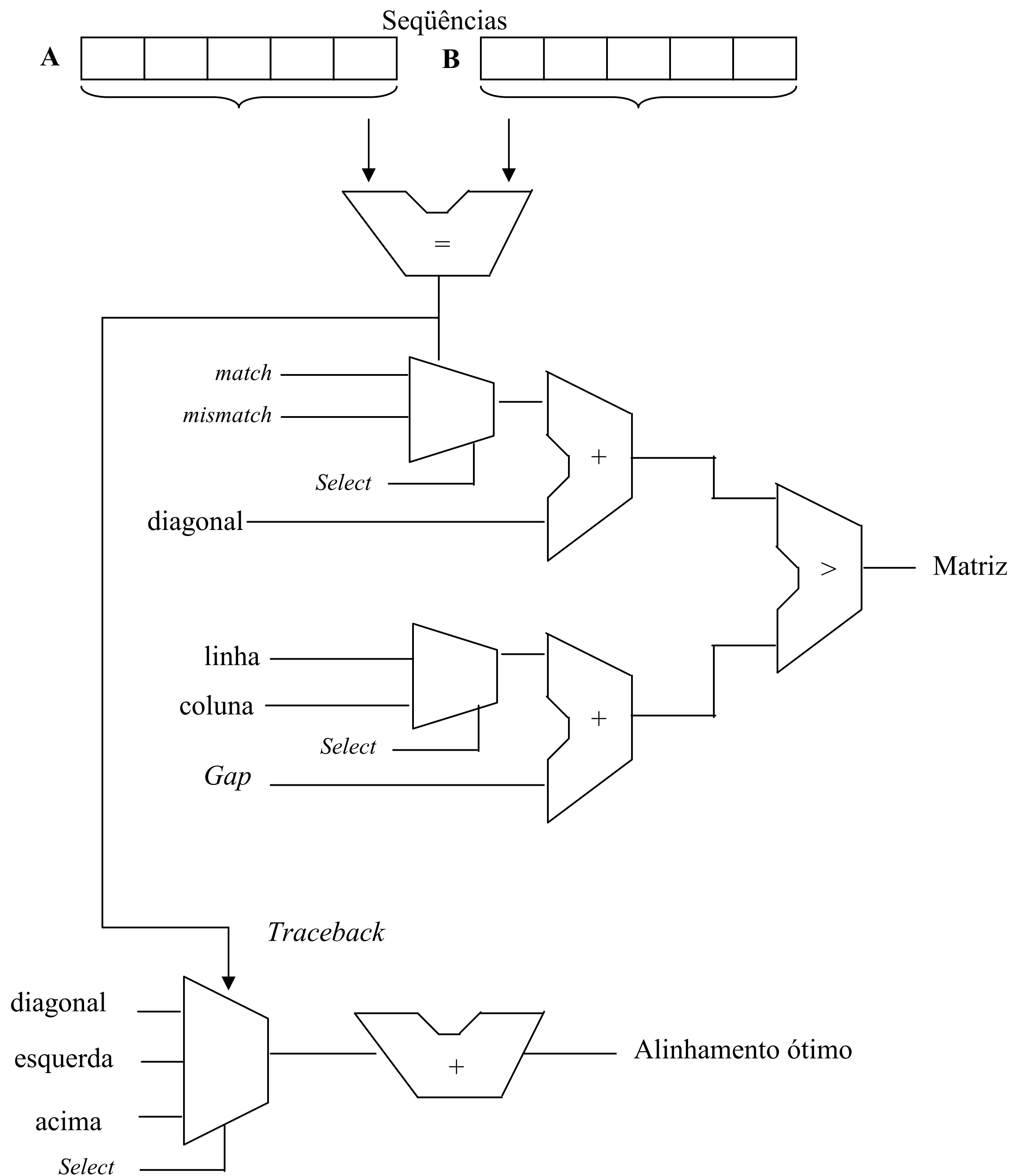


Figura 5.3 Representação estrutural para o alinhamento global.

Esta implementação é composta pela entrada das duas seqüências (A e B) com cinco caracteres cada que entram no comparador a fim de comparar se os caracteres são iguais ou não. Em seguida, utilizou-se de dois multiplexadores com a presença dos sinais de entrada *match*, *mismatch*, linha e coluna, sendo a saída direcionada para dois somadores que realizam a soma da função MAX (equação 2.4). Depois desta soma, o comparador compara qual dos três valores desta função é o maior valor e este valor é armazenado na matriz.

Para realizar o procedimento *traceback* é necessário dispor novamente da comparação de caracteres e de um terceiro multiplexador com sinais de entrada correspondentes a: esquerda, diagonal e acima. O sinal de saída serve de entrada em outro somador que realiza a soma para obter o alinhamento ótimo.

Entretanto, na implementação para o alinhamento local, necessita da inclusão do valor zero para que este seja utilizado no somador da função MAX e assim que a matriz estiver sido preenchida precisa-se de um comparador para verificar qual dos valores da matriz é o maior valor. O procedimento *traceback* é o mesmo para o alinhamento local.

Existem diferenças ao comparar a implementação da figura 5.3 com a implementação do filtro HScan ilustrada pela figura 3.9, pois o filtro não faz cálculos exatos e tem como objetivo localizar segmentos similares entre duas seqüências de DNA, ou seja, detectar áreas potencialmente interessantes onde as similaridades podem aparecer.

A arquitetura do processador do filtro HScan é baseado principalmente sob um somador, o qual adiciona um valor positivo ou negativo de acordo com a igualdade ou diferença dos dados de entrada. Quando a saída torna-se negativa, o somador é ressetado para zero. Observa-se na figura 3.9 que a computação de cada diagonal pode ser desempenhada em paralelo e suportada sob uma rede linear sistólica, onde cada processador é responsável pela computação do *score* da diagonal principal.

5.4 Descrição da Implementação em FPGA

A implementação interna deve ser modelada para executar operações específicas dos algoritmos residentes no sistema. Conforme descrito na seção 2.3 a programação dinâmica voltada para o alinhamento global e local é composta de três fases: i) Inicialização; ii) Preenchimento da matriz de programação dinâmica para obter

o alinhamento global e local; e iii) *Traceback*. Similarmente, as operações em *hardware* descritas usando a linguagem de programação VHDL.

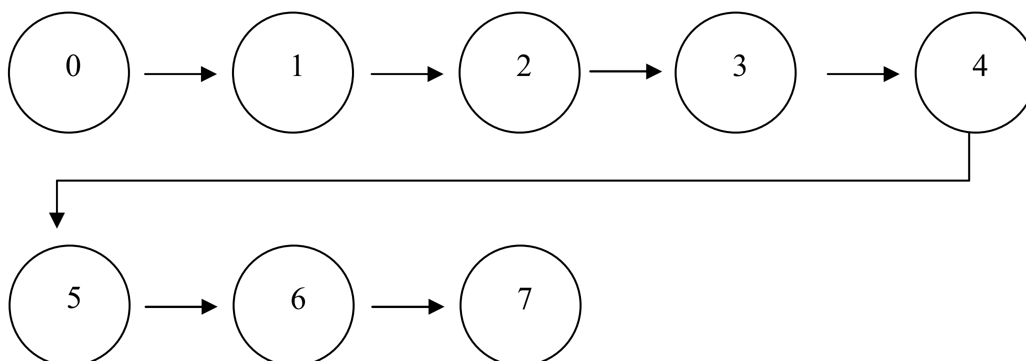
As seqüências que foram utilizadas para serem analisadas e compardas nos algoritmos de alinhamento global e local são compostas de 8 caracteres de DNA o que corresponde a uma matriz de 8x8. Como exemplo, consideramos as seguintes seqüências:

Seqüência 1: **GAATTGCA e Seqüência 2**: **GGATCATG.**

Na descrição em VHDL usamos máquina de estado finito (*Finite State Machine* – FSM) [59], ou seja, a máquina de estado permite descrever os estados da máquina e as suas respectivas transições de maneira síncrona. Com a máquina de estado é possível: i) ficar em um estado sempre após ocorrer um evento; ii) mudar de estado conforme mudança de um valor; iii) mudança de estado em seqüência lógica; e iv) mudança de estado sem seqüência.

5.4.1 AGM – Algoritmo Global em *Hardware*

A figura 5.4 exibe a representação do diagrama dos estados do algoritmo Global de maneira seqüencial implementados em VHDL, visto que, após a execução do estado 0 dá-se início ao próximo estado e assim sucessivamente, que também segue a mesma metodologia no algoritmo Local, os estados para o algoritmo Global são melhor definidos na tabela 5.1.

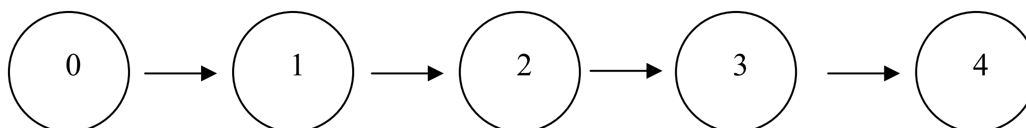


Fgiura 5.4 Representação do diagrama de estados do algoritmo Global utilizando o recurso de máquina de estado.

Tabela 5.1 Descrição dos estados do algoritmo Global.

Nº do Estado	**Descrição dos Estados**
0	Inicializa a máquina de estados e zera todas as posições da matriz.
1	Preenche a primeira linha do inicio ao fim.
2	Preenche a primeira coluna do inicio ao fim.
3	Inicialização dos caracteres das seqüências 1 e 2 que serão comparadas nos respectivos vetores da linha e coluna.
4	Apresentação dos caracteres das seqüências 1 e 2 que serão comparadas nos respectivos vetores da linha e da coluna.
5	Cálculo para realizar o preenchimento do restante da matriz.
6	Atribuição de valores a sinais e variáveis.
7	Cálculo para realizar o procedimento *traceback*, o qual retorna o melhor alinhamento entre as seqüências analisadas e comparadas.

Realizamos também otimização dos algoritmos Global e Local em VHDL, ou seja, a diminuindo a quantidade de estados, onde priorizamos as operações efetuadas nos algoritmos. Após a otimização do algoritmo Global em VHDL conseguiu-se o seguinte diagrama de estados de maneira seqüencial (ver figura 5.5).



Fgiura 5.5 Representação do diagrama de estados do algoritmo Global otimizado.

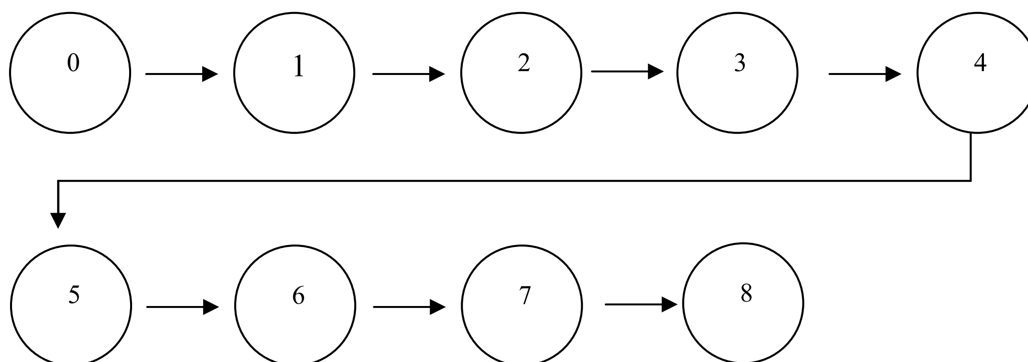
A tabela 5.2 apresenta o número dos estados e suas respectivas descrições, visto que, anteriormente o número de estados era igual a 8 otimizando para apenas 5 estados.

Tabela 5.2 Descrição dos estados do algoritmo Global otimizado.

Nº do Estado	**Descrição dos Estados**
0	Inicializa a máquina de estados e zera todas as posições da matriz.
1	Preenche a primeira linha e primeira coluna do inicio ao fim.
2	Inicialização dos caracteres das seqüências 1 e 2 que serão comparadas e apresentação dos mesmos nos respectivos vetores da linha e da coluna.
3	Cálculo para realizar o preenchimento do restante da matriz e atribuição de valores a sinais e variáveis.
4	Cálculo para realizar o procedimento *traceback*, o qual retorna o melhor alinhamento entre as seqüências analisadas e comparadas.

5.4.2 ALM Algoritmo Local em *Hardware*

O algoritmo Local em VHDL de início foi composto de 9 estados (ver figura 5.6) de maneira seqüencial para poder conseguir realizar todos os seus cálculos. A tabela 5.3 apresenta a descrição de cada um desses estados.

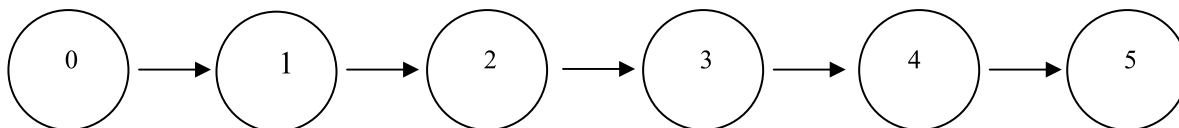


Fgiura 5.6 Representação do diagrama de estados do algoritmo Local.

Tabela 5.3 Descrição dos estados do algoritmo Local.

Nº do Estado	Descrição dos Estados
0	Inicializa a máquina de estados e zera todas as posições da matriz.
1	Preenche a primeira linha do início ao fim.
2	Preenche a primeira coluna do inicio ao fim.
3	Inicialização dos caracteres das seqüências 1 e 2 que serão comparadas nos respectivos vetores da linha e da coluna.
4	Apresentação dos caracteres das seqüências 1 e 2 que serão comparadas nos respectivos vetores da linha e da coluna.
5	Cálculo para realizar o preenchimento do restante da matriz.
6	Busca e retorna o maior valor dentro da matriz.
7	Atribuição de valores a sinais e variáveis.
8	Cálculo para realizar o procedimento *traceback*, o qual retorna o melhor alinhamento entre as seqüências analisadas e comparadas.

Da mesma forma como realizado com o algoritmo Global em VHDL, após a otimização do algoritmo Local conseguiu-se diminuir o número de estados de 9 para 6, conforme se observa na figura 5.7 e tabela 5.4.



Fgiura 5.7 Representação do diagrama de estados do algoritmo Local otimizado.

Tabela 5.4 Descrição dos estados do algoritmo Local otimizado.

Nº do Estado	Descrição dos Estados
0	Inicializa a máquina de estados e zera todas as posições da matriz.
1	Preenche a primeira linha e primeira coluna do inicio ao fim.
2	Inicialização dos caracteres das seqüências 1 e 2 que serão comparadas e apresentação dos mesmos nos respectivos vetores da linha e da coluna.
3	Cálculo para realizar o preenchimento do restante da matriz
4	Busca e retorna o maior valor dentro da matriz.
5	Cálculo para realizar o procedimento *traceback*, o qual retorna o melhor alinhamento entre as seqüências analisadas e comparadas.

Portanto, utilizando máquina de estado, os algoritmos Global e Local, versão otimizada são compostos por um único processo, 5 e 6 estados. Uma vez que a matriz de alinhamento local precisa encontrar o maior valor dentro da matriz e a partir deste valor realizar o procedimento *traceback*, justifica-se que o algoritmo Local otimizado tenha um estado a mais que o Global otimizado.

Desta forma, os algoritmos Global e Local em VHDL foram descritos e simulados usando o recurso de síntese do FPGA e obteve-se dados de desempenho em *hardware*. Após esta descrição fez-se o uso do recurso da implementação dos mesmos, usando apenas duas famílias da ferramenta *Xilinx Foundation Series*. É importante ressaltar que *arrays* (arranjos) multidimensionais geralmente não são suportados pela ferramenta de síntese (*synthesis*) e por este motivo as famílias da ferramenta *Xilinx* que conseguiram suportar a implementação dos *arrays* usados nos algoritmos foram apenas:

❖ **Família** : VIRTEX **Dispositivo**: V800FG680 **Velocidade**: -6; e	❖ **Família**: VIRTEXE **Dispositivo**: V812EFG900 **Velocidade**: -8

Maiores detalhes do algoritmo Local otimizado são descritos no apêndice B, já no apêndice C aparecem todos os cálculos obtidos na implementação em VHDL.

5.4.3 Simulação do Algoritmo AGM Otimizado em *Hardware*

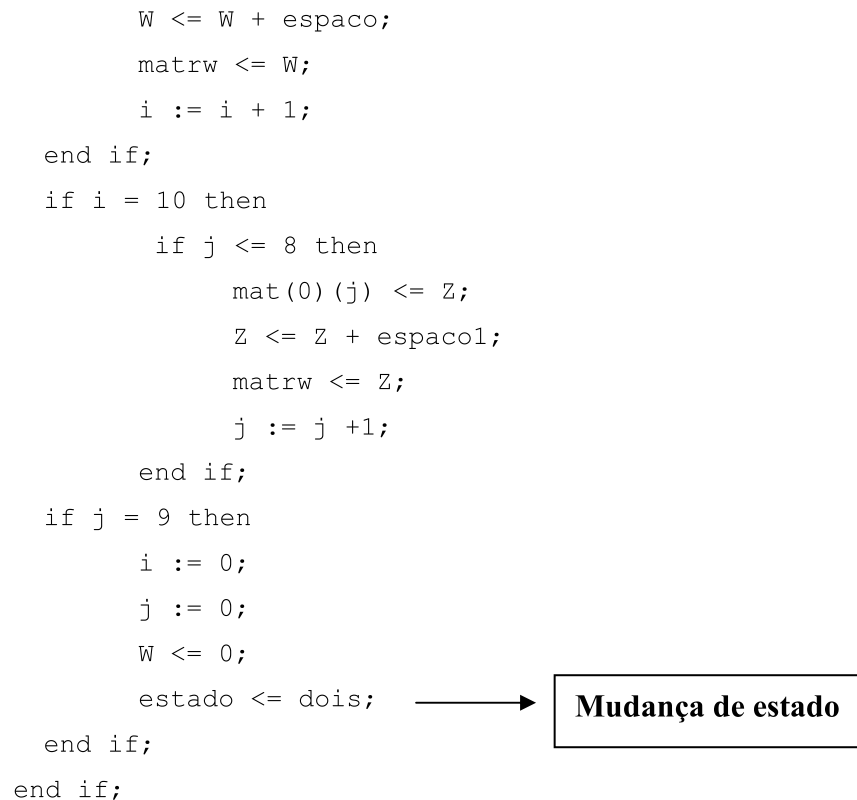
A seguir apresetam-se detalhes de implementação e simulação do algoritmo Global otimizado, utilizando a família VIRTEX.

- **Estado 0**: inicializa todos os sinais, variáveis e todas as posições da matriz com valores iguais a zero, isto é, para que os índices da matriz não inicializem com valores errados "sujos". Assim que este estado é finalizado dá-se início ao próximo estado. Este estado é igual para os algoritmos Global e Local bem como para as suas respectivas otimizações.

```
--ESTADO ZERO (Linha de comentário)
if i <= 8 then  --Zerar todas as posicoes da matriz.
   if j <= 8 then
      mat(i)(j)<= 0;
      j := j +1;
   end if;
   if j = 9 then
      j := 0;
      i := i + 1;
   end if;
   if i = 9 then
      estado <= um;      ———→      Mudança de estado.
      i := 0;
      j := 0;
   end if;
 end if;
```

- **Estado 1**: Preenchimento da primeira linha e primeira coluna do início ao fim.

```
--ESTADO UM
       elsif estado = um then -- Preenchimento da primeira
linha e primeira coluna do inicio ao fim.
       if i <= 9 then
             mat(i)(0) <= W;
```



```
          W <= W + espaco;
          matrw <= W;
          i := i + 1;
    end if;
    if i = 10 then
           if j <= 8 then
                mat(0)(j) <= Z;
                Z <= Z + espaco1;
                matrw <= Z;
                j := j +1;
          end if;
    if j = 9 then
          i := 0;
          j := 0;
          W <= 0;
          estado <= dois;
    end if;
  end if;
```


A figura 5.8 refere-se à simulação do estado 1 utilizando a ferramenta de síntese da *Xilinx*. Nessa figura é possível observar que a porta de saída representada por **x3** (em hexadecimal) mostra o estado corrente da execução, sendo que este precisa de 18 *clocks* para realizar o preenchimento da primeira linha e primeira coluna do início, onde o primeiro valor a ser preenchido é o 252 em seguida 248, e assim sucessivamente até chegar ao valor 0 (representado por 000). Da mesma maneira dá-se início ao preenchimento da primeira posição da coluna também representado pelo valor 252, conforme nota-se na figura.

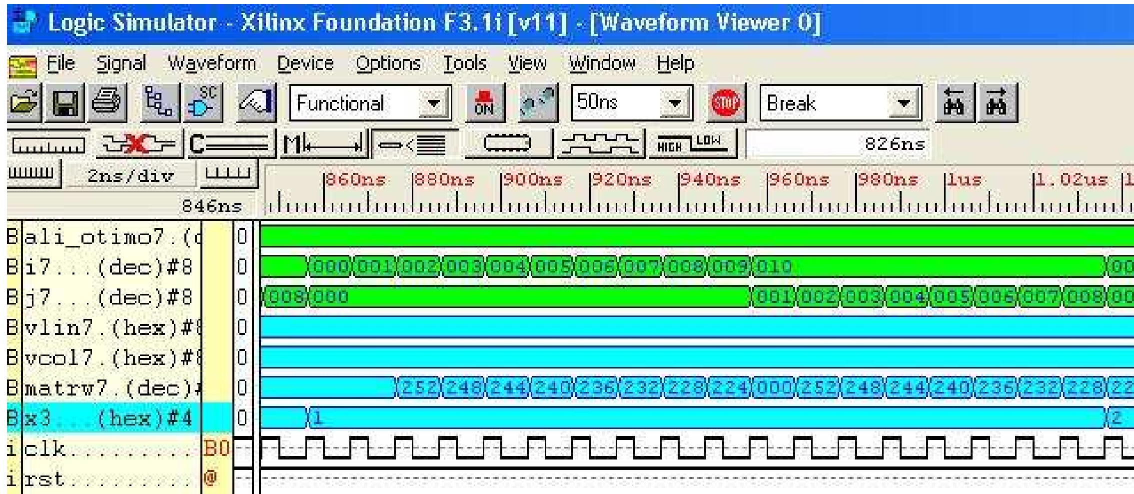


Figura 5.8 Representação do estado 1 na ferramenta de simulação *Xilinx*.

Ao adentrar no próximo estado, estado dois:

- **Estado 2**: Inicialização das letras e apresentação destes nos vetores de linha e coluna.

```
--ESTADO DOIS
  elsif estado = dois then  -- Inicializao das letras nos
vetores vlinha e vcoluna
        vlinha(1) <= G; vcoluna(1) <= G;
        vlinha(2) <= A; vcoluna(2) <= G;
        vlinha(3) <= A; vcoluna(3) <= A;
        vlinha(4) <= T; vcoluna(4) <= T;
        vlinha(5) <= T; vcoluna(5) <= C;
        vlinha(6) <= G; vcoluna(6) <= A;
        vlinha(7) <= C; vcoluna(7) <= T;
        vlinha(8) <= A; vcoluna(8) <= G;

  case i is -- Apresentacao das letras nos vetores vlinha e
vcoluna
        when 1 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 2 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 3 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 4 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 5 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 6 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 7 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when 8 => vlin <= vlinha(i); vcol <= vcoluna(i);
        when others => null;
  end case;
```

```
i := i + 1;
      if i = 9 then
            i := 1;
      j := 1;
      S := 0;
      V1 := 0;
      V2 := 0;
      V3 := 0;
      estado <= tres;
end if;
```

Mudança de estado

A figura 5.9 exibe a simulação deste estado, cujos valores estão representados no formato hexadecimal, ou seja, o caracter A = 41, T = 54, G = 47 e C = 43. É possível observar que a porta de saída representada por **x3** (em hexadecimal) mostra o estado corrente da execução, sendo que este estado precisa de 9 *clocks* para realizar a inicialização e apresentação dos respectivos vetores das letras (**vlin e vcol**) nos vetores para linha e coluna.

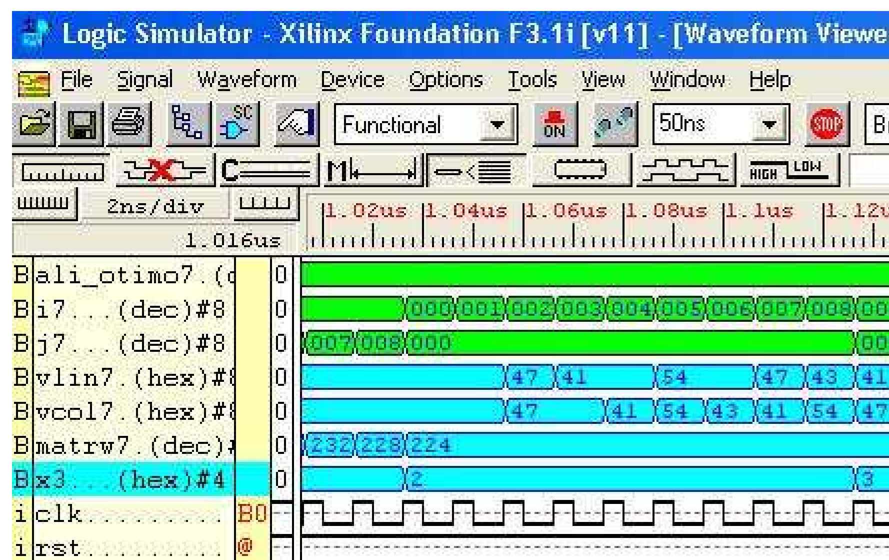


Figura 5.9 Representação do estado 2 na ferramenta de simulação *Xilinx*.

- **Estado 3**: Cálculo para realizar o preenchimento do restante da matriz.

```
--ESTADO TRES
  elsif estado = tres then -- Cálculo para realizar o
preenchimento do restante da matriz
```

```
if i <= 8 then
   if j <= 8 then
      if (vlinha(i) = vcoluna(j)) then
            S := igual;
            matrw <= S;
            vlin <= vlinha(i);
            vcol <= vcoluna(j);
      else
            S := diferente;
            matrw <= S;
            vlin <= vlinha(i);
            vcol <= vcoluna(j);
     end if;
     V1 := mat(i-1)(j-1) + S;
      V2 := mat(i)(j-1) + espaco;
      V3 := mat(i-1)(j)+ espaco;

      if ((V1 >= V2) and (V1 >= V3)) then
         mat(i)(j) <= V1;
         matrw <= V1;
      elsif ((V2 >= V1) and (V2 >= V3)) then
          mat(i)(j) <= V2;
          matrw <= V2;
      elsif ((V3 >= V1) and (V3 >= V2)) then
         mat(i)(j) <= V3;
          matrw <= V3;
      end if;
   end if;
      j := j + 1;
     if j = 9 then
        j := 1;
         i := i + 1;
      end if;
      if i = 9 then
         estado <= seis;
          i :=0;
         j :=0;
     end if;
   end if;
```

Desta forma, a matriz de programação dinâmica no algoritmo otimizado em VDHL, é totalmente preenchida tendo os seguintes valores, conforme ilustrados em um trecho da figura 5.10. É possível verificar que a porta de saída representada por **x3** (em hexadecimal) mostra o estado corrente da execução, sendo que este precisa de 64 *clocks* para realizar o preenchimento dos valores no restante da matriz, onde o vetor que armazena a linha (**vlin**) inicia na primeira posição trazendo o valor 41 em hexadecimal que corresponde à letra A, que compara com todas as letras do vetor coluna da matriz (**vcol**). Nota-se que o valor 41 = A (**vlin**) é comparado com os valores do vetor (**vcol**): 47, 47, 54, 43, 41, 54 e 47.

Depois de comparar todo o vetor (**vcol**), inicia-se a comparação da segunda letra do vetor linha (**vlin**) com todas as letras do vetor (**vcol**) e assim sucessivamente até preencher a matriz inteira.

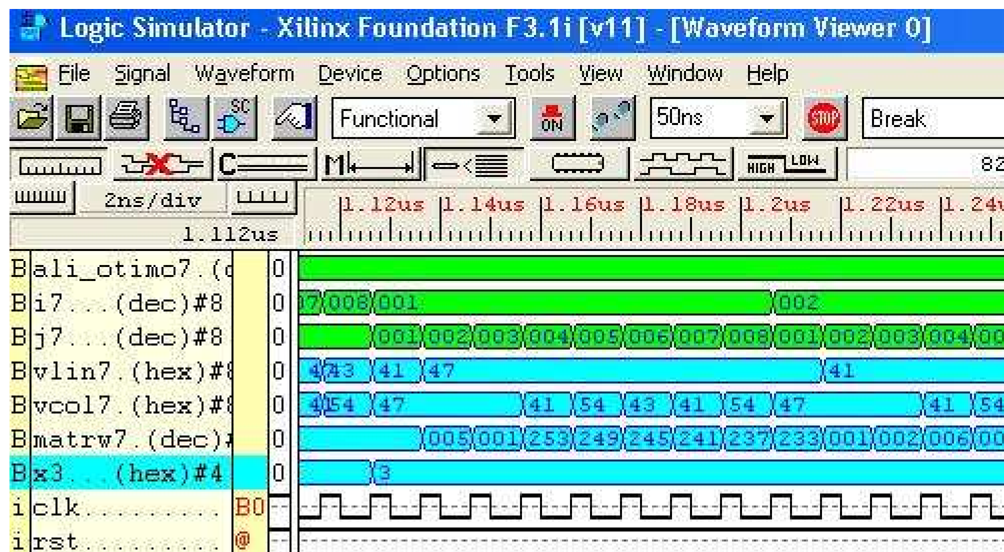


Figura 5.10 Representação do estado três na ferramenta de simulação *Xilinx*.

Para uma melhor compreensão, a figura 5.11 exibe a matriz do algoritmo Global otimizado descrito em VHDL com todos os seus dados.

			1	2	3	4	5	6	7	8
		-	G	A	A	T	T	G	C	A
	-	0	252	248	244	240	236	232	228	224
1	G	252	5	1	253	249	245	241	237	233
2	G	248	1	2	6	2	254	250	246	242
3	A	244	253	254	7	3	255	3	255	251
4	T	240	249	250	3	12	8	4	8	4
5	C	236	245	246	255	8	9	5	9	5
6	A	232	241	250	251	4	5	6	5	14
7	T	228	237	246	247	0	9	5	3	10
8	G	224	233	242	251	252	5	14	10	6

Figura 5.11. Representação da matriz do algoritmo Global preenchida utilizando a linguagem VDHL

Percebe-se que esta matriz apresenta valores grandes, tais como: 252, 240, 232, etc, os quais representam valores negativos em VHDL. Ao verificar a matriz do algoritmo Global implementado em C (ver capítulo 4), nota-se que a primeira linha e primeira coluna são representadas por valores negativos. Ao contrário de VHDL, os valores negativos são todos representados em valores positivos, não obastante todos os cálculos realizados estão corretos.

Conforme descrito no capítulo 2, após ter preenchido a matriz, inicia-se o processo de *traceback*, referente ao estado 4, o qual identifica os segmentos bem como produz o alinhamento correspondente que determinará qual é o resultado do sistema de pontuação máxima da matriz de programação dinâmica, denominado melhor alinhamento entre as duas seqüências ou alinhamento ótimo.

- **Estado 4**: Realizar o procedimento *traceback*.

```
--ESTADO QUATRO
  elsif estado = quatro then   -- Realizar o procedimento
TRACEBACK
  if i > 0 then
        if j > 0 then
        if vlinha(i) = vcoluna(j) then
              ali_otimo := ali_otimo + igual;
              mat(i)(j) <= ali_otimo;
              matrw <= ali_otimo;
              vlin <= vlinha(i);
              vcol <= vcoluna(j);

        elsif ((vlinha(i) = A ) or (vlinha(i) = T)) and
((vcoluna(j) = A) or (vcoluna(j) = T )) then
              ali_otimo := ali_otimo + diferente;
              mat(i)(j)<= ali_otimo;
              matrw <= ali_otimo;
              vlin <= vlinha(i);
              vcol <= vcoluna(j);

        elsif ((vlinha(i) = C ) or (vlinha(i) = G)) and
((vcoluna(j) = C) or (vcoluna(j) = G )) then

              ali_otimo := ali_otimo + diferente;
              mat(i)(j)<= ali_otimo;
              matrw <= ali_otimo;
```

```
                    vlin <= vlinha(i);
                    vcol <= vcoluna(j);

              else ali_otimo := ali_otimo + espaco;
                    mat(i)(j)<= ali_otimo;
                    matrw <= ali_otimo;
                    vlin <= vlinha(i);
                    vcol <= vcoluna(j);
              end if;
              if (vlinha(i) = vcoluna(j)) then
                    j := j - 1;
                    i := i - 1;
              else
                    linha := j;
                    coluna := i;
                    esquerda := mat(i-1)(j);
                    diagonal := mat(i - 1)(j - 1);
                    acima := mat(i)(j-1);

              if ((esquerda >= diagonal) and (esquerda >= acima))
        then
                    i := i - 1;
              elsif ((acima>= diagonal) and (acima >= esquerda))
        then
                   j := j - 1;
              elsif ((diagonal >= acima) and (diagonal >=
        esquerda)) then
                    j := j - 1;
                    i := i - 1;
              end if;
         end if;
      end if;
    end if;
```

Na figura 5.12 é possível observar os valores que são obtidos no estado 4. Assim, os valores correspondentes ao melhor alinhamento entre as seqüências de DNA analisadas e comparadas são exibidas nesta figura. Por exemplo, no sinal de saída **matrw foram** apresentados os valores da matriz que o procedimento *traceback* percorreu, as variáveis **i** e **j** mostraram as posições da matriz que o *traceback* seguiu, os sinais de saída **vlin** e **vcol** exibem os números em hexadecimal que correspondem as

letras (A, T, C, G) que foram comparadas e finalmente o alinhamento ótimo que apresentou seu resultado, o valor 7.

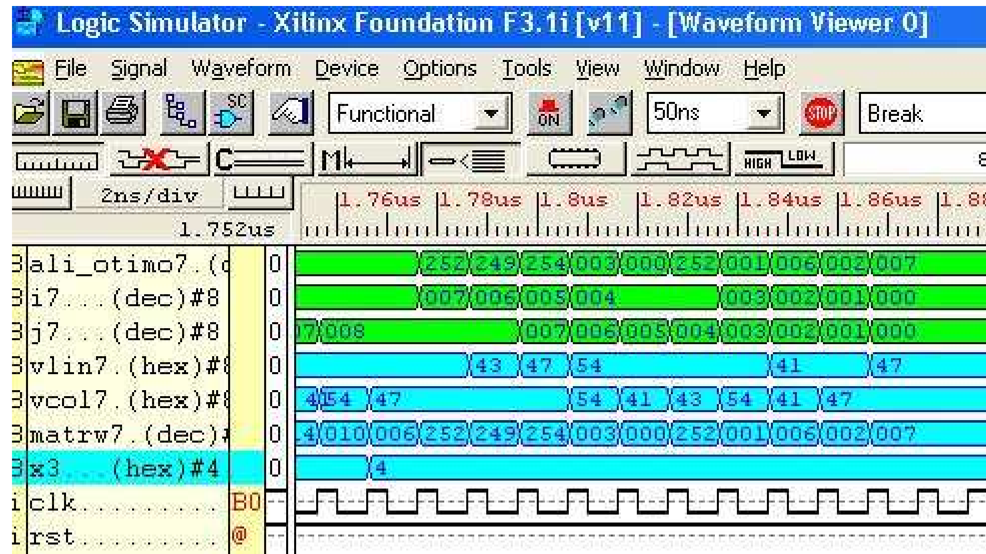


Figura 5.12 Representação do estado 4 na ferramenta de simulação em VHDL.

O caminho percorrido pelo *traceback* está representado pela figura 5.13, cujos valores encontram-se especificados em hexadecimal, e a figura 5.14 exibe os cálculos do alinhamento ótimo.

VLIN	41	43	47	54	54	54	54	41	41	47
VCOL	47	47	47	54	41	43	54	41	47	47
VLIN	A	C	G	T	T	T	T	A	A	G
VCOL	G	G	G	T	A	C	T	A	G	G

Figura 5.13 Procedimento *traceback* na simulação em VHDL no algoritmo Global otimizado.

A	-	-4
C	G	-3
G	G	5
T	T	5
T	A	-3
T	-	-4
T	T	5
A	A	5
A	-	-4
G	G	5
	TOTAL =	7

Figura 5.14 Cálculo do alinhamento ótimo obtido no algoritmo Global otimizado em VHDL.

Utilizando-se a ferramenta de implementação do *Xilinx* e usando as famílias VIRTEX V800FG680 e VIRTEXE V812EFG900 obtiveram-se diversos dados de desempenho, tais como os recursos espaciais utilizados e a temporização no FPGA.

A tabela 5.5 apresenta o tempo gasto no processo de síntese e de implementação dos algoritmos Global e Local e nos otimizados, usando as famílias VIRTEX e VIRTEXE, que conseguiram suportar a implementação dos algoritmos mencionados.

Tabela 5.5 Tempo consumido dos algoritmos Global e Local otimizados em VHDL.

	VIRTEX V800FG680		**VIRTEXE** V812EFG900	
	Tempo de Síntese	**Tempo de Implementação**	**Tempo de Síntese**	**Tempo de Implementação**
Global	12 minutos	4 minutos	12 minutos	4 minutos
Global Otimizado	15 minutos	3 minutos	14 minutos	3 minutos
Local	21 minutos	4 minutos	22 minutos	4 minutos
Local Otimizado	19 minutos	4 minutos	20 minutos	5 minutos

A diferença de tempo dos algoritmos tanto na síntese como na implementação é pequena, devido ao algoritmo global ter um estado a menos e desta forma, obtendo assim o maior tempo consumido na ferramenta VIRTEXE no algoritmo Local.

Contudo, as tabelas 5.6 e 5.7 apresentam a quantidade de *clocks* que são consumidos em todos os estados dos algoritmos Global e Local e também nos algoritmos otimizados.

Tabela 5.6 Quantidade de *clocks* dos estados nos algoritmos Global e Local em VHDL.

	VIRTEX V800FG680		**VIRTEXE** V812EFG900	
	Global	**Local**	**Global**	**Local**
Números de *clocks*				
Estado 0:	85 *clocks*	85 *clocks*	85 *clocks*	85 *clocks*
Estado 1:	9 *clocks*	9 *clocks*	9 *clocks*	9 *clocks*
Estado 2:	9 *clocks*	9 *clocks*	9 *clocks*	9 *clocks*
Estado 3:	1 *clock*	1 *clock*	1 *clock*	1 *clock*
Estado 4:	9 *clocks*	9 *clocks*	9 *clocks*	9 *clocks*
Estado 5:	64 *clocks*	64 *clocks*	64 *clocks*	64 *clocks*
Estado 6:	1 *clock*	64 *clocks*	1 *clock*	64 *clocks*
Estado 7:	11 *clocks*	1 *clock*	11 *clocks*	1 *clock*
Estado 8:	---------	10 *clocks*	---------	10 *clocks*
Total de *clocks*	**189 *clocks***	**252 *clocks***	**189 *clocks***	**252 *clocks***

Tabela 5.7 Quantidade de *clocks* dos estados nos algoritmos Global e Local otimizados em VHDL.

	VIRTEX V800FG680		VIRTEXE V812EFG900	
	Global Otimi	**Local Oti**	**Global Oti**	**Local Oti**
Números de *clocks*				
Estado 0:	85 *clocks*	85 *clocks*	85 *clocks*	85 *clocks*
Estado 1:	18 *clocks*	18 *clocks*	18 *clocks*	18 *clocks*
Estado 2:	9 *clocks*	9 *clocks*	9 *clocks*	9 *clocks*
Estado 3:	64 *clocks*	64 *clocks*	64 *clocks*	64 *clocks*
Estado 4:	*11clocks*	64 *clocks*	*11clocks*	64 *clocks*
Estado 5:	-----------	10 *clocks*	-----------	10 *clocks*
Total de *clocks*	**187 *clocks***	**250 *clocks***	**187 *clocks***	**250 *clocks***

Nota-se que a quantidade de *clocks* consumidas pelos algoritmos tanto na família VIRTEX e VIRTEXE são iguais, tendo o algoritmo Local como o algoritmo otimizado, um estado a mais, o qual consome 10 *clocks*, o que resulta numa quantidade total acima que o algoritmo Global e o seu otimizado.

5.4.4 Recursos FPGAs usados pelos Algoritmos

As tabelas 5.8 e 5.9 informam os principais recursos utilizados no FPGA, denominados de recursos espaciais, com a implementação dos algoritmos Global e Local e suas otimizações no processo de implementação em VHDL usando as famílias VIRTEX V800FG680 e VIRTEXE V812EFG900. A ferramenta *Xilinx* foi usada em um computador *Pentium* IV 1.66 GHz, AMD Athlon (tm) XP 2000+, 240 MB de memória RAM, 256 Kbytes de *cache*, com sistema operacional *Microsoft Windows* XP.

Tabela 5.8 Recursos utilizados no FPGA VIRTEX V800FG680.

RECURSOS VIRTEX V800FG680	**GLOBAL**	**GLOBAL Oti**	**LOCAL**	**LOCAL Oti**
Número de *Slices*	1.288 (14%)	1.377 (14%)	1.493 (15%)	1.455 (15%)
Número de *Slice* dos *Flip Flops*	749 (3%)	757 (4%)	806 (4%)	806 (4%)
Número de LUTs [4]de 4 entradas	2.486 (13%)	2.662 (14%)	2,864 (15%)	2,788 (14%)
Número de LUTs	2.484	2.660	2.862	2.786
Número de IOBs[5]	29 (5%)	29 (5%)	45 (8%)	45 (8%)
Total de portas usadas no projeto	21.511	22.721	24.382	23.926

[4] LUTs (*Look Up Table*) : realiza funções lógicas combinacionais.
[5] IOBs (*In/Out Block*): bloco de entrada e saída, localizado na periferia dos FPGAs, são responsáveis pela interface com o ambiente.

Portanto, nota-se que o algoritmo Local precisa de um maior número de *slices*, de LUTs de 4 entradas, de LUTs bem como o total de portas utilizadas no projeto, entretanto, o algoritmo Local e sua otimização tiveram o maior números de *flip-flops* (só podem armazenar um *bit*, isto é, 0 ou 1) e de IOBs (*In/Out Block*) na utilização da família VIRTEX V800FG680.

A figura 5.15 permite uma melhor visualização e compreensão dos recursos utilizados nos FPGAs, cujos dados são os da tabela 5.8.

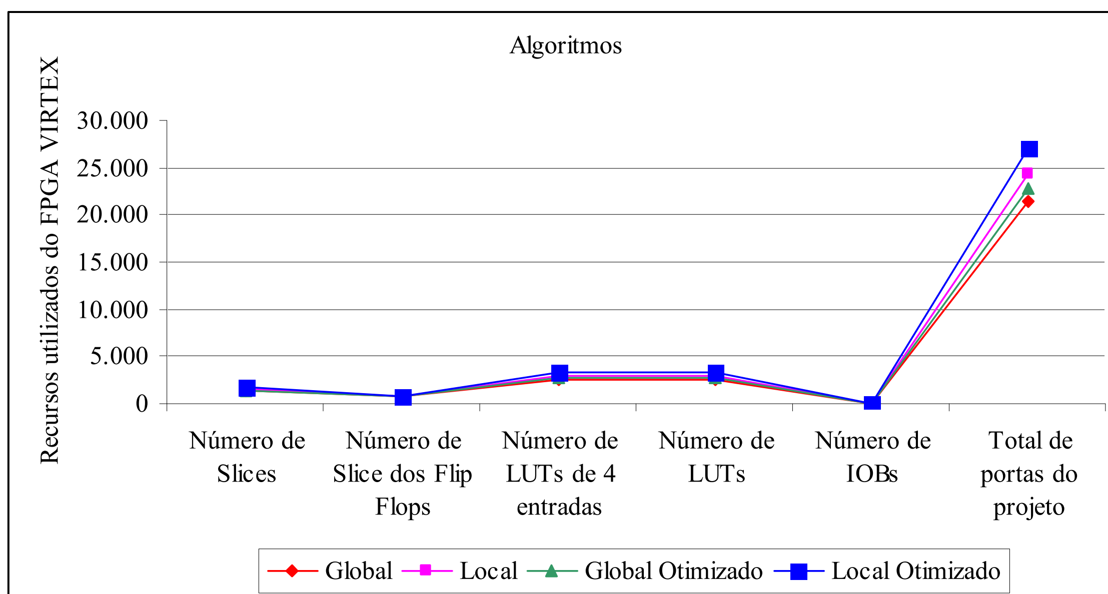


Figura 5.15 Recursos utilizados da família VIRTEX V800FG680 nos algoritmos Global, Global Otimizado, Local e Local Otimizado.

Da mesma forma, procurou-se analisar todos os recursos utilizados da família VIRTEXE V812EFG900, conforme pode-se observar na tabela 5.9.

Tabela 5.9 Recursos utilizados no FPGA VIRTEXE V812EFG900.

RECURSOS VIRTEX E V812EFG900	**GLOBAL**	**GLOBAL Otimizado**	**LOCAL**	**LOCAL Otimizado**
Número de *Slices*	1.287 (13%)	1.375 (14%)	1.493 (15%)	1.708 (18%)
Número de *Slice* dos *Flip Flops*	749 (3%)	757 (4%)	806 (4%)	821 (4%)
Número de LUTs de 4 entradas	2.486 (13%)	2.657 (13%)	2.863 (15%)	3.279 (17%)
Número de LUTs	2.484	2.655	2.861	3.277
Número de IOBs	29 (5%)	29 (5%)	45 (8%)	45 (8%)
Total de portas usadas no projeto	21.511	22.691	24.736	27.127

Percebe-se que todos os recursos em seu maior número foram utilizados pelo algoritmo Local e nota-se novamente poucas alterações com relação às famílias do FPGA em questão, mesmo porque não existem diferenças nas implementações dos algoritmos (ver figura 5.16).

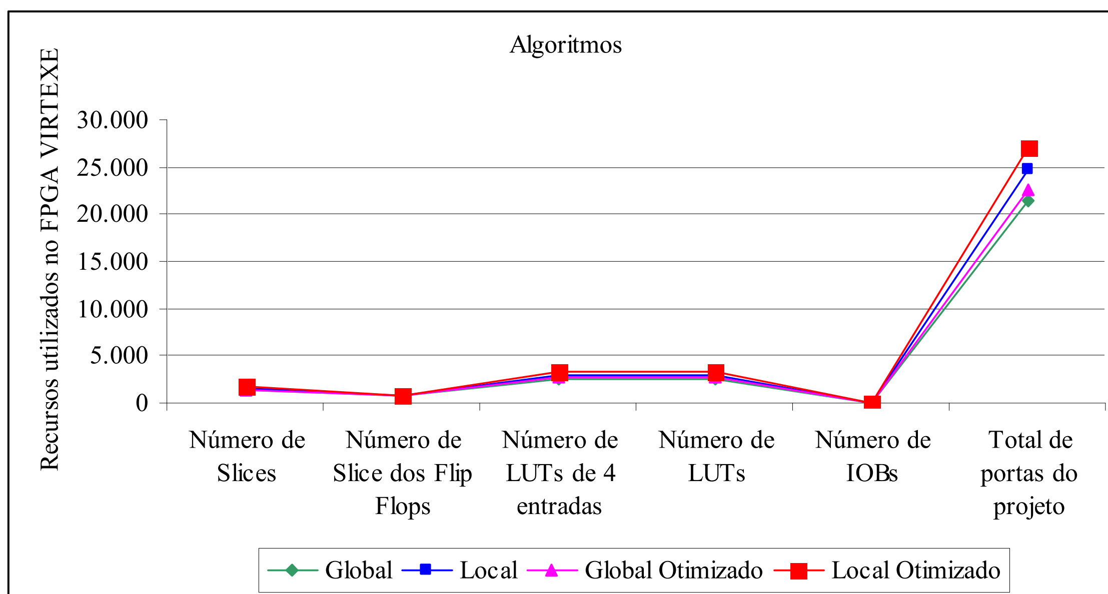


Figura 5.16 Recursos utilizados da família VIRTEXE V812EFG900 nos algoritmos Global, Global Otimizado, Local e Local Otimizado.

Os tempos gerados pela ferramenta *Xilinx* representam o tempo necessário para executar todas as operações dos algoritmos Global e Local e suas respectivas otimizações. Importante ressaltar que as tabelas 5.10 e 5.11 exibem as estatísticas temporais em nanossegundos, ou seja, que estão relacionadas aos atrasos para gerar a saída desejada de um determinado circuito. Este tempo é o resultado da soma do tempo de lógica mais o de roteamento em FPGA do circuito digital.

Tabela 5.10 Temporização dos algoritmos Global e Local das versões otimizadas em VHDL com a família VIRTEX

	Algoritmos	Tempo de Lógica	Tempo de Roteamento	Atraso Total	Freqüência de Operação
VIRTEX V800FG680	**Global**	10.034 ns 20.4%	39.103 ns 79.6%	49.137 ns	20.351 MHz
	Global Otimizado	10.878 ns 22.9%	36.590 ns 77.1%	47.468 ns	21.067 MHz
	Local	10.139 ns 17.1%	49.149 ns 82.9%	59.228 ns	16.867 MHz
	Local Otimizado	10.900 ns 20.8%	41.518 ns 72.9%	52.418 ns	19.077 MHz

Tabela 5.11 Temporização dos algoritmos Global e Local e as otimizações em VHDL com a família VIRTEXE

	Algoritmos	Tempo de Lógica	Tempo de Roteamento	Atraso Total	Freqüência de Operação
VIRTEXE V812EFG900	**Global**	7.022 ns 15.3%	38.997 ns 84.7%	46.019 ns	21.730 MHz
	Global Otimizado	7.161 ns 16.0%	37.674 ns 84.0%	44.835 ns	22.304 MHz
	Local	6.715 ns 14.1%	40.970 ns 85.9%	47.685 ns	20.971 MHz
	Local Otimizado	7.089 ns 14.9%	40.542 ns 85.1%	47.631 ns	20.995 MHz

Observa-se um tempo de lógica menor para a família VIRTEXE em relação à família VIRTEX. Outro destaque é que o algoritmo Local requer mais tempo de execução em relação ao algoritmo Global.

Pode-se concluir que as estatísticas temporais comparadas não sofrem grandes contrastes, quando se mapeam com diferentes FPGAs, uma vez que, os algoritmos implementados são os mesmos.

5.5 Comparação *Software* e *Hardware*

Na tabela 5.12 tem-se o tempo nanosegundos da implementação dos algoritmos de alinhamento global e local em *software*, usando a linguagem de programação C, entretanto, a tabela 5.13 apresenta os tempos de implementação em *hardware* (FPGAs), visto que a ferramenta *Xilinx* suportou seqüências com tamanho até de 15 caracteres por seqüência.

Tabela 5.12 Performance da implementação em *software* (em C).

Tamanho da Seqüência	Global	Local
8	**216.000** ns	**156.000** ns
10	**220.000** ns	**160.000** ns
15	**750.000** ns	**744.000** ns

Tabela 5.13 Performance da implementação em *hardware* (em FPGA).

	VIRTEX V800FG680		VIRTEX E V812EFG900	
Tamanho da Seqüência	**Global Otimizado**	**Local Otimizado**	**Global Otimizado**	**Local Otimizado**
8	47.468 ns	52.418 ns	44.835 ns	47.631 ns
10	45.902 ns	55.065 ns	44.748 ns	51.117 ns
15	57.445 ns	67.476 ns	50.887 ns	62.626 ns

Para uma melhor visualização, a figura 5.17 ilustra o tempo de execução das nossas implementações, dos algoritmos Global, Local (em C), Global Otimizado e Local Otimizado (em FPGA), utilizando a família VIRTEX V800FG680.

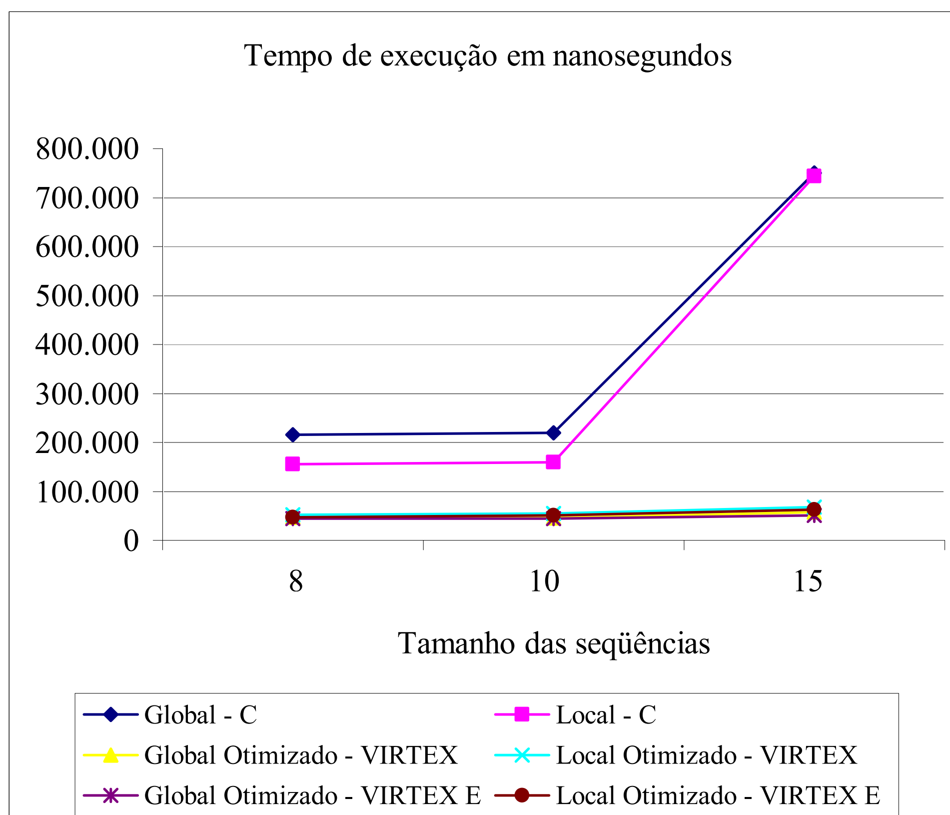


Figura 5.17 Tempo de execução em nanosegundos das implementações em *software* e *hardware*

Após ter realizado as implementações dos algoritmos de alinhamento global e local das seqüências genéticas de DNA utilizando a técnica de programação dinâmica tanto em nível de *software* (em C) quanto em nível de *hardware* (VHDL e prototipados em FPGA) foi possível comparar os dados com os obtidos da tabela 3.1, que oferece informações referentes a caracteres processados por segundo (CPS – *Characters Processed per Second*).

Nós utilizamos a seguinte consideração: se para uma seqüência de 10 caracteres, o circuito precisou um tempo medido em nanosegundos, então, calculamos quantos caracteres ele consegue processar em um segundo. No algoritmo global, usando a família VIRTEX V800FG680, obteve-se um tempo de 45.902 ns.

Então: (**1 seg * 10 caracteres / 45.902 ns) = 10/45.902*10^{-9} = 217.864.923 CPS.**

Comparados com os sistemas descritos no capítulo 3 (ver tabela 3.1), onde aparece o valor de CPS para as respectivas implementações desses algoritmos em linguagem de alto nível e plataformas especiais de *hardware*, e observando o mesmo tamanho de seqüência – 10 caracteres, podemos dizer que a nossa implementação em

hardware mostrou-se mais rápida do que os sistemas apresentados na tabela 3.1. A tabela 5.14 permite verificar a ordem em que nossa implementação foi mais rápida.

Tabela 5.14 Desempenho de tempo em nanosegundos.

Sistemas	**CPS**	**Quantidade de vezes mais lento**
Morony	**217.864.923**	1
BioSCAN	1.979.583	110
Sun 690	5.776.729	37.7
Convex 240	6.423.184	33.9
MasPar-1	1.162.529	187.4

O autor [11] propôs uma estrutura sistólica para resolver o algoritmo local, não obstante não é possível comparar a nossa implementação com essa proposta uma vez que não é claro como eles obtiveram os dados de desempenho. Os testes foram feitos com seqüências fictícias cuja matriz de similaridade era conhecida ou poderia ser construída facilmente. Para o teste de desempenho, vetores de diferentes tamanhos foram sintetizados e suas freqüências máximas de operação foram obtidas a partir de informações fornecidas pela ferramenta de síntese da ALTERA. Observando os dados da tabela 3.3, pode-se perceber que quanto maior a quantidade de células do vetor, menor sua freqüência máxima de operação. Porém, o autor menciona também que o aumento do número de células não deveria afetar a freqüência de operação do circuito, o que parece contraditório.

Finalmente, ao comparar os dados em *software* e *hardware* nesta dissertação, nota-se a diferença significativa nos tempos de execução, visto que as performances em *hardware* são melhores do que as versões em *software*. Não obstante, os bons resultados obtidos, seria interessante analisar seqüências com tamanhos variados e maiores em *hardware*, já que não se conseguiu que a ferramenta *Xilinx* suportasse a implementação para tamanhos de seqüências maiores a 15 caracteres.

CAPÍTULO

6

CONCLUSÕES

Na segunda metade da década de 90, com o surgimento dos seqüenciadores automáticos de DNA, houve uma explosão na quantidade de seqüências a serem armazenadas, exigindo recursos computacionais cada vez mais eficientes. Além do armazenamento, ocorria paralelamente a necessidade de análise desses dados, o que tornou indispensável a utilização de plataformas computacionais eficientes para a interpretação dos resultados obtidos.

Desta forma nasceu a bioinformática, essa nova ciência envolveria a união de diversas linhas de conhecimento, tais como: a engenharia de *softwares* a matemática, a estatística, a ciência da computação e a biologia molecular. Os primeiros projetos na área foram compostos por profissionais de diferentes áreas da biologia e informática e percebia-se certa dificuldade de comunicação; enquanto o biólogo procurava uma solução que levasse em consideração as incertezas e erros que ocorrem na prática, o cientista da computação procurava uma solução eficiente para um problema bem definido.

Assim, surgiu a necessidade de um novo profissional que entendesse bem ambas as áreas e fizesse a ponte entre elas, o bioinformata. Esse profissional deveria ter o conhecimento suficiente para saber quais eram os problemas biológicos reais e quais seriam as opções viáveis de desenvolvimento e abordagem computacional dos problemas em questão.

Sabendo que a biologia computacional está ligada diretamente com a bioinformática, então é de extrema importância lembrar quais são os sistemas operacionais, linguagens de programação e os principais banco de dados que realizam o armazenamento de milhares de informações dos inúmeros projetos de pesquisas na área acadêmica da bioinformática.

O desafio apresentado pela bioinformática é encontrar a melhor forma de armazenamento e de pesquisa para os dados gerados por projetos de pesquisa na área da bioinformática, como o projeto genoma humano, que possui centenas de *gigabytes* de dados a espera para serem armazenados e tratados.

Para tanto, surgiu a necessidade de possuir formas de armazenamento, acesso e pesquisa sobre tais dados, para que se consiga trazer a informação desejada da melhor maneira possível, devendo existir assim, técnicas diferenciadas para o tratamento destes dados, que são nada mais do que grandes cadeias de DNA (em banco de dados, grandes cadeias de *caracteres*).

Nesta dissertação utilizou-se de um dos principais métodos de alinhamento de seqüências, a programação dinâmica, cujo método computacional calcula o melhor alinhamento possível entre as seqüências genéticas, sendo este método um caso especial da técnica de indução, onde resolve-se o problema utilizando a solução de problemas menores, que tem como objetivo guardar todas as subsoluções numa tabela (matriz) e determinar quais subsoluções são necessárias para a solução global, tendo que existir essa tabela (matriz) para que não seja preciso re-calcular as subsoluções.

A técnica de programação dinâmica aplicada nos algoritmos de alinhamento global, conhecido como algoritmo de *Needleman-Wunsch* tem como finalidade a similaridade global entre duas seqüências de DNA, onde todas as bases são alinhadas umas com as outras ou com *gaps*. Isto é, os valores globais requerem que o alinhamento comece no início e se estenda até o final de todo o comprimento da seqüência.

O algoritmo de alinhamento local, como o algoritmo de *Smith-Waterman*, possui similaridade local entre duas seqüências, não necessita alinhar todas as bases em

todas as seqüências, ou seja, os valores locais requerem identificação na região mais similar entre duas seqüências e também a penalidade por falta atribuída ou não.

Esses algoritmos acima mencionados foram implementados tanto em *software* usando a linguagem de programação C quanto em *hardware* (FPGA). Com todos os dados de performances obtidos mostrou que este trabalho em termos de tempo de execução em nível de *hardware* usando VHDL e FPGAs e sintetizando o circuito com a ferramenta *Xilinx Series Fundation* é mais rápida do que a nível de *software* utilizando a linguagem de programação C e computadores pessoais modernos.

Foi possível implementar os algoritmos de alinhamento global e local de seqüências genéticas em FPGAs, porém somente as placas VIRTEX e VIRTEXE conseguiram suportar esses algoritmos, onde as seqüências fictícias de DNA analisadas e comparadas conseguiram suportar seqüências com tamanho máximo de 15 caracteres.

A comparação da implementação em *hardware* (FPGA) apresentada em nanosegundos é mais rápida do que a implementada em *software* (em C), onde foram comparadas seqüências fictícias com tamanhos de até 1000 caracteres e em FPGA o *hardware* suportou apenas tamanho com 15 caracteres.

Observando os resultados obtidos, percebe-se que a implementação em *hardware* apresentou tempo de execução mais rápido que alguns sistemas amplamente usados, tais como: BioSCAN, Sun 690, Convex 240 e o MasPar-1 que implementam os algoritmos em linguagem de alto nível e os executam em *hardware* com maior capacidade de processamento do que os algoritmos global e local implementados nesta dissertação, pois as nossas implementações utilizaram a tecnologia FPGA da *Xilinx*, cujas famílias foram a VIRTEX e VIRTEX E.

Portanto, FPGAs parecem ser uma alternativa para resolver o problema dos algoritmos destinados ao alinhamento e a comparação de seqüências genéticas, uma vez que esta tecnologia, FPGA, permite baixos custos e ganho em tempo de execução, porém possui uma desvantagem em relação ao seu paralelismo que é limitado devido ao tamanho do FPGA, mas para obter mais desempenho talvez seja necessário propor soluções usando mais placas de FGPAs funcionando em paralelo, por exemplo.

Como trabalhos futuros, sugere-se realizar mais otimizações, paralelizar os cálculos, com o objetivo de diminuir os recursos espaciais do FPGA e aumentar a velocidade de processamento dos algoritmos de seqüenciamento genético. Seria interessante propor novos algoritmos e/ou implementações, a fim de contribuir para solucionar o problema do sequenciamento de gens, implementações como por exemplo:

mais placas em paralelo, Também propor outras técnicas para otimizar a velocidade de processamento como *pipeline*, paralelismo e o uso de outras tecnologias como por exemplo SoC - *System on Chip*.

Seria interessante também propor novas arquiteturas, tais como: sistólica, MIMD e realizar as respectivas implementações em FPGAs, assim como fazer comparações e a respectiva análise de desempenho com as utilizadas nesta dissertação e em trabalhos correlatos da área.

REFERÊNCIAS

[1] ALTSCHUL, S. F. et al. **Basic local alignment search tool**. Journal of Molecular Biology, 215:403–410, 1990.

[2] ALTSCHUL, S. F. et al. **Gapped BLAST and PSI-BLAST: a new generation of protein database search programs**. Nucleic Acids Res. 25:3389-3402, 1997.

[3] ARNOLD, J. M. BUELL, D.A. and DAVIS, E.G. **Splash-2.** In Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, p. 316-322, 1992.

[4] BAIROCH. A and APWEILER, R. **The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000**. Nucleics Acid Reserch 28(1): p. 45-48, 2000.

[5] BAMPI, S. and NAVAUX, P. **Arquitetura Avançadas de Computadores** - INF01191 SEM: 99/2. Universidade Federal do Rio Grande do Sul-Instituto de Informática, 1992.

[6] BERTIN, P., RONCIN, D. and VUILLEMIN, J. **Introduction to programmable active memories**. In J. McWhirter J. McCanny and E. Swartzlander, editors, Systolic Array Processors, p.301-309, Pretice Hall, 1989.

[7] BERTIN, P., RONCIN, D. and VUILLEMIN, J. **Programmable active memories: a performance assessment**. In F. Meyer, B. Monien, and A.L. Rosenberg, editors, Parallel Architectures and their efficient use, p. 119-130. LNCS, Outubro de 1992.

[8] BOKHARI, S. H. and SAUER, J. R. **Sequence alignment on the cray mta-2**. Online Proceedings. Second IEEE International Workshop on High Performance Computational Biology. Nice, France, 22 de Abril de 2003. Disponível em: <http://www.hicomb.org/HiCOMB2003/proceedings.html>. Acessado em: Maio de 2004.

[9] BORAH, M. et al. **A SIMD solution to the sequence comparison problem on the MGAP**. In: ASAP, (Capello, P. et al., eds) Los Alamitos, CA: IEEE CS, p. 336-45, 1994.

[10] BROWN, S. and ROSE, J. **FPGA and CPLD Architectures: A tutorial**, IEEE Design & Test Computers, 1996.

[11] CARVALHO, L. G. A. **Uma abordagem em hardware para algoritmos de comparação de seqüências baseados em programação dinâmica**. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade de Brasília – DF, Brasil, Dezembro de 2003.

[12] COMPUGEN. **The bioccelerator machine**. Israel, 1993.

[13] CHOI, S. H. et al. **Signal processing applications using VHDL on Splash 2**. Proc. of VHDL Intl. Users' Forum, Fall Conference, Virginia, p. 6.11-6.19. 17 (a) (b) (c) (d), Novembro de 1994.

[14] CHOW, E. et al. **Biological information signal processor**. In: ASAP, (Valero, M. et al., eds) p.144-160, Los Alamitos, CA: IEEE CS, 1991.

[15] CRISTINO, A. S. **Principais Algoritmos de Alinhamento de Seqüências Genéticas**. Disponível em: <http://www.ime.usp.br/~alexsc>. Acessado em: julho de 2004.

[16] DETTLOFF, W. D. et al. **A 50 MHz 1.5m transistor asic for biosequence analysis**. International Solid State Circuit Conference Digest of Technical Papers, p. 40-42, 1991.

[17] DURBIN, R. et al. **Biological sequence analysis: probabilistic models of proteins and nucleic acids**. Cambridge, UK: CambridgeUniversity Press, p. 356, 1998.

[18] EUROPEAN BIOINFORMATICS INSTITUTE. **FASTA – Protein Database Query**. Disponível em: <http://www.ebi.ac.uk/fasta33/>. Acessado em: agosto de 2004.

[19] FPGA, **Xilinx Company**. Disponível em: <http://www.xilinx.com/>. Acessado em: agosto de 2004.

[20] GIBBS, A. J. and MCINTYRE, G. A. **The diagram: a method for comparing sequences. Its use with amino acid and nucleotide sequences**. Eur. J. Biochem., 16:1-11, 1970.

[21] GOKHALE, M. B. et al. **SPLASH: A reconfigurable linear logic arrays**. In Proceedings of the International Conference on Parallel Processing, p.526-532, Agosto de 1990.

[22] GOKHALE, M. et al. **Processing in Memory: The Terasys massively parallel PIM array**. Computer, 28 (4), p. 23-31, 1995.

[23] GOTOH, O. **An improved algorithm for matching biological sequences**. Journal of Molecular Biology, 162:705-708, 1982.

[24] GREEN, P. **Documentation for PHRAP and Cross_match** (version 0.990319), 1999. Disponível em: <http://www.phrap.org/phrap.docs/phrap.html>. Acessado em: agosto de 2004.

[25] GRUPO DE ENGENHARIA GENÔMICA. Linhas de Pesquisa. Universidade Federal de Santa Catarina. Disponível em: <http://www.intelab.ufsc.br/genomica/linhas.htm>. Acessado em: julho de 2004.

[26] GUCCIONE, S. A. and KELLER, E. **Gene Matching using JBits**, Proc. 12th Int. Workshop on Field-Programmable Logic and Applications (FPL'02), LNCS 2438,

p.1168-1171, 2002.

[27] GUERDOUX-JAMET, P. and LAVENIER, D. **Systolic filter for fast dna similarity search**. In ASAP'95, Strasbourg, Julho de 1995.

[28] GUERDOUX-JAMET, P. and LAVENIER, D. **SAMBA: hardware accelerator for biological sequence comparison**, CABIOS 12 (6), p. 609-615, 1997.

[29] HENIKOFF, S. and HENIKOFF, J. G. **Amino acid substituition matrices from protein blocks**. Proceedings of National Academy of Science USA, 89(22):10915-10919, Novembro de 1992.

[30] HIGA, R. H. **Entendendo e Interpretando os Parâmetros Utilizados por BLAST**. Instruções Técnicas. Campinas, SP. Dezembro, 2001.

[31] HIGGINS, H. G. and SHARP, P. M. **CLUSTAL: a package for performing multiple sequence alignment on a microcomputer**. Gene, 73:237–244, 1988.

[32] HIGGINS, H. G. and SHARP, P. M. **Fast and sensitive multiple sequence alignments on a microcomputer**. CABIOS, 5:151–153, 1989.

[33] HIGGINS, H. G., BLEASBY, A. J. and FUCHS, R. **CLUSTAL V**: **improved software for multiple sequence alignment**. CABIOS, 8:189–191, 1992.

[34] HIGGINS, H. G., THOMPSON, J. D. and GIBSON, T. J. **Using CLUSTAL for multiple sequence alignments**. Methods Enzymol, 266:383–402, 1996.

[35] HIRSCHBERG, J. D. et al. **Kestrel: A programmable array for sequence analysis**. In: ASAP p.25-34, Los Alamitos, CA: IEEE CS, 1996.

[36] HOANG, D. T. **A systolic array for the sequence alignment problem**. Technical Report CS-92-22, Dept. Computer Science, Brown University, Providence, RI, 1992.

[37] HOANG, D.T. **Searching genetic databases on Splash 2**. In Proc. IEEE Workshop on FPGAs for Custom Computing Machines, (Buell, D. A. & Pocek, K. L., eds), IEEE CS, p. 185-191, 1993.

[38] HOFFMAN, D. L. **A Comparison of the BioSCAN Algorithm on Multiple Architectures** – Hoffman. Department of Computer Science. University of North Carolina at Chapel Hill. 27599-3175, 1993.

[39] HUGHEY R. and LOPRESTI, D. P. **A software approach to fault detection on programmable systolic arrays**. In Proc. Symp. Parallel and Distributed Processing (B. Shirazi and H. Sudborough, eds.), p. 523-526, IEEE Computer Society, Dezembro de 1990.

[40] HUGHEY, R. and Lopresti, D. P. **B-SYS: A 470-processor programmable systolic array**. Proceedings of the International Conference Parallel Processing, p.580-583, 1991.

[41] HUGHEY, R. **Programmable Systolic Arrays**. PhD thesis, Departament of Computer Science, Brown University, Providence, RI, 1991.

[42] HUGHEY, R. **Parallel Hardware for Sequence Comparison and Alignment**, CABIOS 12 (6), p. 473-479, 1996.

[43] IEEE Standard **VHDL Language Reference Manual**. IEEE Std 1076-1987, New York, NY, 1988.

[44] JEANMOUGIN, F. et al. **Multiple sequence alignment with CLUSTAL X**. Trends Biochem Sci, 23:403–405, Outubro de 1998. Disponível em: < http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=9810230&dopt=Abstract >. Acessado em: Dezembro de 2004.

[45] KARLIN, S. and ALTSCHUL, S. F. **Methods for assessing the statistical of the National Academy of Science**, 87:2264-2268, 1990.

[46] LANG, H. **The instruction systolic array a parallel architecture for vlsi**. Journal of VLSI , 1:65–74, 1986.

[47] LAVENIER, D. **Dedicated Hardware for Biological Sequence Comparison**. Journal of Universal Computer Science, 2(2):77-86, 1996.

[48] LAVENIER, D. **SAMBA: Systolic Accelerators for Molecular Biological Applications**. Technical Report 988 IRISA 35042 Rennes Cedex, França, 1996.

[49] LEVENSHTEIN, V. I. **Binary codes capable of correcting deletions, insertions and reversals**. Cyber. Contr. Theory, 10:707-710, 1966.

[50] MAIZEL, J. V. and LENK, R. P. **Enhanced graphic matrix analysis of nucleic acid and protein sequences**. Proceedings National Academy Sciences. USA, 78:7665-7669, 1981.

[51] MATTOS, N. P. e RASKIN, S. F. **Um Comparativo Risc x PC. Aspectos da Arquitetura Risc**. 1994. Disponível em: <http://www.pr.gov.br/batebyte/edicoes/1994/bb35/aspectos.htm>. Acessado em: novembro de 2004.

[52] MCCASKILL, J. S. et al. **NGEN: A Massively Parallel Reconfigurable Computer for Biological Simulation: Towards a Sef-Organizing Computer**, In: International Conference On Evolvable Systems: From Biology To Hardware, 1., Tsukuba. Proceedings... Berlin: Springer-Verlag (Lecture Notes in Computer Science v. 1259), p.260-276, 1997.

[53] MIRSKY, E. A. **Coarse-Grain Reconfigurable computing**, PhD thesis, Massachussetts Institute of Technology (MIT), 1996.

[54] MITCHELL, M. **Introduction Genetic Algorithms**, MIT Press, 1996.

[55] National Center for Biotechnology Information – NCBI. Disponível em: <http://www.ncbi.nlm.nih.gov>. Acessado em: junho de 2004.

[56] NEEDLEMAN S. B. and WUNSCH, C. D. **A general method applicable to the search for similarities in the amino acid sequence of two proteins**. Journal Molecular Biology, 48, p. 443-453, 1970.

[57] OLIVEIRA, D. C. **Alinhamento de Seqüências**. Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtenção do título de Bacharel em Ciência da Computação. Lavras - Minas Gerais – Brasil, 2002.

[58] OLIVER, T. and SCHMIDT, B. **High Performance Biosequence Database Scanning on Reconfigurable Platforms**. School of Computer Engineering. Nanyang Technological University. Disponível em: <http://www.hicomb.org/HiCOMB2004/proceedings.html>. Acessado em março de 2004.

[59] ORDONEZ, E. D. M. et al. **Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)**. Pompéia: Bless, 2003. 300p.

[60] Paracel Inc. **FDF-3 product information**. Pasadema, CA, 1996. Disponível em: <http://www.paracel.com>. Acessado em: julho de 2004.

[61] PEARSON, W.R. and LIPMAN, D. J. **Improved tools for biological sequence comparison**. Proceedings of the National Academy of Sciences, 85:3244-3248, 1988.

[62] PEARSON, W. R. **Searching Protein Sequence libraries: comparison of the sensitivity and selectivity of the smith and waterman and fasta algorithms**. Genomics, 11:635-650, 1991.

[63] QUEIROZ, A. **Apostila de Introdução a Bioinformática**. Departamento de Biofísica e Farmacologia. Disciplina de Bioinformática**.** Universidade Federal do Rio Grande do Norte. Fevereiro de 2002. Disponível em: < labbi.uesc.br/apostilas/introducao_a_bioinformatica.pdf >. Acessado em: Agosto de 2004.

[64] RAIMBAULT, F. et al. (1993). **Fine grain parallelism on a MIMD machine using FPGA's**, Research Repport INRIA N. 1983.

[65] RAIMBAULT, E. and LAVENIER, D. **Relacs for Systolic Programming**. In ASAP'93, p. 132-135, IEEE Computer Society Press, Venice, Itália, Outubro de 1993.

[66] RAUPP. F. M. P. **Similaridade, Alinhamentos e Perfis**. Curso de Especialização Lato Sensu em Bioinformática. Laboratório Nacional de Computação Científica, 2002.

[67] RENCHER, M. and HUTCHINGS, B. L. **Automated target recognition on Splash 2**. Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97), 1997.

[68] Revista Eletrônica de Jornalismo Científico – ComCiência. **Bioinformática. Software para bioinformática ganha mercado**, nº 46, Agosto de 2003. ISSN 1519 – 7654x. Disponível em: <http://comciencia.br/reportagens/bioinformatica/bio03.shtml>. Acessado em: 05 de outubro de 2004.

[69] ROCHA, E. **Módulo de Bioiformática. Análise de Seqüências. Cadeira de Algorítmica e Programação**. Atelier de BioInformatique. U. Paris 6 & Institut Pasteur, Paris. Disponível em: <http:www.labbi.uesc.br/apostilas/bioinformatica_analise_de_sequencias.pdf>. Acessado em: julho de 2004.

[70] ROUCHKA, E. C. **Dynamic Programming**. Disponível em: <http://www.sbc.su.se/~per/molbioinfo2001/dynprog/dynamic.html>. Acessado em: junho de 2004.

[71] SANKOFF, D. **Matching sequences under deletion/insertion constraints**. Proceedings National Academy Sciences. USA, 69:4-6, 1972.

[72] SANKOFF, D. and CEDERGREN, R. J. **A test for nucleotide sequence homology**. Journal Molecular Biology, 77:159-164, 1973.

[73] SELLERS, P. H. **On the theory and computation of evolutionary distances**. SIAM J.Appl. Math., 26:787-793, 1974.

[74] SETUBAL, J.C. and MEIDANIS, J. **Introduction to computational molecular biology**. Boston: PWS Publishing, p.296, 1997.

[75] SETÚBAL, J. C. **Comparação de Seqüências I, II, III**. Notas de aula de MO640/MC931. Biologia Computacional – 1º Semestre de 2003. Instituto de Computação – UNICAMP – Campinas - SP.

[76] SCHMIDT, B., SCHRÖDER, H. and SCHIMMLER, M. **Massively Parallel Solutions for Molecular Sequence Analysis**, Proc. 1st IEEE Int. Workshop on High Performance Computational Biology, Ft. Lauderdale, Flórida, 2002.

[77] SINGH, R. K. et al. **A scalable systolic multiprocessor system for analysis of biological sequences**. Proceedings of the Symposium on Integrated Systems, p.167-182, 1993.

[78] SINGH, R. K. et al. **BIOSCAN: a network sharable computational resource for searching biosequence databases**, CABIOS, 12 (3), p.191-196, 1996.

[79] SLONIM, D. **From patterns to pathways: gene expression data analysis comes of age**. Nature Genetics, 32:502–508, 2002.

[80] SMITH, T. and WATERMAN, M. **Identification of common molecular subsequences.** Journal of Moleular Biology, 147:195–197, 1981.

[81] SONNHAMMER, E. L. and DURBIN, R. **A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis**. Gene, 167:GC1-10, 1995.

[82] STADEN, R. **An interactive graphics program for comparing and aligning nucleic acid and amino acid sequences**. Nucleic Acids Res, 10:2951-2961, 1982.

[83] TANGEN, U., r MCCASKILL, J. S. **Hardware Evolution with a Massively Parallel Dynamically Reconfigurable Computer: POLYP**, In: International Conference On Evolvable Systems: From Biology To Hardware, 2., 1998, Lausanne. Proceedings... Berlin: Springer-Verlag (Lecture Notes in Computer Science v. 1478), p.364-371, 1998.

[84] THOMPSON, J. D., HIGGINS, H. G. and GIBSON, T. J. **CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice**. Nucleic Acids Research, 22:4673–4680, 1994.

[85] THOMPSON, J. D. et al. **The CLUSTAL X windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools**. Nucleic Acids Research, 24:4876–4882, 1997.

[86] VHDL International. Disponível em: <http://www.vhdl.org/> . Acessado em: agosto de 2004.

[87] VIANNA, C. J. M. **Arquiteturas de Computadores para alinhamento de seqüências biológicas**. Relatório Técnico da Disciplina de Arquiteturas Avançadas. Departamento de Computação e Estatística – Centro de Ciências Exatas e Tecnologia. Universidade Federal de Mato Grosso do Sul. Julho de 2003.

[88] VICENTE, L., MAGALHÃES, D. and GAFEIRA, J. **Distâncias e similaridades em seqüências mRNA utilizando algoritmos evolutivos**. Instituto Superior Técnico. Licenciatura em Engenharia Eletrônica e de Computadores. Engenharia Biomédica 2002.

[89] VILLASENOR, J. and SMITH, W. H. M. (1998) **Configurable Computing**, Scientific American, USA.

[90] WARD, R. J., GALBAR, V. D. and RUIZ, J. C. **Introdução geral à disciplina de Engenharia Genética**. Faculdade de Medicina de Riberão Preto – FRMP – USP. Departamento de Bioquímica e Imonologia, 2004.

[91] WARD, R. J., GALBAN, V. D. and RUIZ, J. C. **Tópicos em Bioinformática. Introdução à Análise de Seqüências**. Faculdade de Medicina de Riberão Preto – FRMP – USP. Departamento de Bioquímica e Imonologia. Disponível em: <http://rbq.fmrp.usp.br/bioinfo/bioinfo1.htm>. Acessado em: julho de 2004.

[92] WHITE, C. T. et al. **BioSCAN: A VLSI-basead system for biosequence analysis**. Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors, p. 504-509, 1991.

[93] XILINX. **The Programmable Gate Array Data Book**. 2100 Logic Drive, San Jose, CA 95124 USA, 1992.

[94] XILINX Inc. **Virtex and Virtex-E Overview**. Disponível em: <http://www.xlinx.com/products/virtex/ss_vir.htm>. Acessado em: julho de 2004.

[95] YAMAGUCHI, Y., MARUYAMA, T. and KONAGAYA, A. **High Speed Homology Search with FPGAs**, Proc. Pacific Symposium on Biocomputing'02, p.271- 282, 2002.

Apêndice A

Neste apêndice, observa-se a descrição das operações dos algoritmos Global e Local usando a linguagem de programação em C. Nos algoritmos Global e Local utilizaram-se um total de 16 funções, cujos nomes visam identificar o funcionamento das mesmas, sendo elas:

- **Abrearq** – Abre os arquivos (Linha, Coluna e Matriz);
- **Limpaarq** – Apaga os arquivos (Linha, Coluna e Matriz);
- **Cadastro** – Cadastro da primeira linha e primeira coluna da matriz;
- **Xvalor** – Calcula o valor das posições da matriz (linha, coluna);
- **Buscavalor** – Busca o valor na matriz (maior valor);
- **Tempo** – Calcula o tempo gasto pra realizar a comparação e alinhamento das seqüências analisadas;
- **Matriz** – Realiza os cálculos e faz o preenchimento dos dados na matriz;
- **Posicionalin** – Posiciona o apontador em uma posição da linha;
- **Posicionacol** – Posiciona o apontador em uma posição da coluna;
- **Posiciona** - Posiciona o apontador em uma posição da matriz;
- **Quadrante** – Desenha os quadrantes da matriz no momento da listagem;
- **Limpamatriz** – Apaga somente o arquivo da matriz;
- **Listagem** – Mostra a matriz preenchida com os cálculos;
- **Ultimaposic** – Calcula a última posição da matriz;
- **Busc_valor** – Busca um valor na matriz (ordem pra percorrer o *traceback*); e
- ***Traceback*** – Percorre a matriz em busca do alinhamento ótimo.

Para maiores detalhes, todas as funções foram descritas no apêncide A.

Abrearq: Abre os arquivos (Linha, Coluna e Matriz).

```
fecha os arquivos (Linha, Coluna, Matriz)
condição (if) --- comparador
condição (if) --- comparador
               |impressão na tela
               |exit(1)
condição (if) --- comparador
condição (if) --- comparador
               |impressão na tela
               |exit(1)
```

condição (if) --- comparador
condição (if) --- comparador
    impressão na tela
    exit(1)

Limpaarq: Apaga os arquivos (Linha, Coluna e Matriz).
impressão na tela
comparador
condição (if) --- comparador
    atribuição (abre arquivo da Matriz)
    atribuição (abre arquivo da Linha)
    atribuição (abre arquivo da Coluna)

Cadastro: Cadastro da primeira linha e primeira coluna da matriz.
chamada da função ABREARQ
atribuição
impressão na tela

laço(do)
    comparador
    condição (if) --- comparador
        contador
        posiciona o arquivo em um byte especifico (fseek Lptr, 0,2)
        escrever no arquivo

laço(while) --- comparador

Xvalor: Calcula o valor das posições da matriz (linha,coluna).
Chamada da função ABREARQ
posiciona o arquivo em um byte especifico (fseek Lptr,0,0)
laço (while)
    condição (if) --- comparador
        break

posiciona o arquivo em um byte especifico (fseek Cptr,0,0)
laço (while)
    condição (if) --- comparador
        break

atribuição
condição (if) --- comparador
        atribuição
    condição (else)
        atribuição

atribuição --- chamada da função BUSCAVALOR(XLIN-1,XCOL-1) Somador
atribuição --- chamada da função BUSCAVALOR(XLIN,XCOL-1) Somador

atribuição --- chamada da função BUSCAVALOR(XLIN-1,XCOL) Somador

condição (if) --- comparador
return

condição (else if) --- comparador
return

condição (else if) --- comparador
return

Buscavalor: Busca o valor na matriz (maior valor).
posiciona o arquivo em um byte especifico (fseek Mptr,0,0)
laço (while)
condição (if) --- comparador
atribuição
return

Tempo: Calcula o tempo gasto para realizar a comparação e o alinhamento das seqüências.
chamada de função (putenv,tzset,ftime)
condição (if) --- comparador
atribuição (variáveis)
condição (else)
atribuição (variáveis)
impressão na tela

Matriz: Calcula e preenche a matriz de programação dinâmica.
impressão na tela
atribuição (variáveis)
chamada da função TEMPO
chamada da função ABREARQ
atribuição (variáveis)
posiciona o arquivo em um byte especifico(fseek Mptr,0,2)
escrever no arquivo MATRIZ
posiciona o arquivo em um byte especifico(fseek Lptr,0,0)

laço (while) LINHA
contador
atribuição (variáveis)
atribuição (variáveis) → Somador
posiciona o arquivo em um byte especifico(fseek Mptr,0,2)
escrever no arquivo MATRIZ

atribuição (variáveis)
posiciona o arquivo em um byte especifico (fseek Cptr,0,0)

laço (while) COLUNA
    atribuição (variáveis)
    posiciona o arquivo em um byte especifico (fseek Lptr,0,0)
    atribuição (variáveis)
    atribuição (variáveis) → Somador
    contador
    posiciona o arquivo em um byte especifico (fseek Mptr,0,2)
    esrever no arquivo MATRIZ

    laço (while) LINHA
        atribuição (variáveis)
        chamada da função XVALOR
        atribuição (variáveis)

        posiciona o arquivo em um byte especifico (fseek Lptr,XCOL,0)
        contador

        posiciona o arquivo em um byte especifico (fseek Mptr,0,2)
        escrever no arquivo MATRIZ

    posiciona o arquivo em um byte especifico (fseek Cptr,XLIN,0)
    contador

chamada da função tempo

Posicionalin: Posiciona o apontador em uma posição da linha.
posiciona o arquivo em um byte especifico (fseek Cptr,0,0)
laço (while)
    condição (if) --- comparador
        break

Posicionacol: Posiciona o apontador em uma posição da coluna.
posiciona o arquivo em um byte especifico (fseek Lptr,0,0)
laço (while)
    condição (if) --- comparador
        break

Posiciona: Posiciona o apontador em uma posição da matriz.
posiciona o arquivo em um byte especifico (fseek Mptr,0,0)
laço (while)
    condição (if) --- comparador
        break

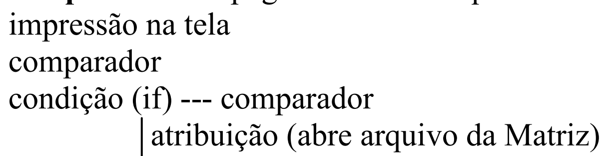
Quadrante: Desenha os quadrantes da matriz no momento da listagem.

chamada da função POSICIONA
condição (if) --- comparador
- chamada da função POSICIONALIN(LI+1)
- condição (if) --- comparador
  - impressão na tela
  - chamada da função POSICIONACOL(CI+1)
  - laço (do - while)
    - impressão na tela
  - condição (else)
    - impressão na tela
    - chamada da função POSICIONACOL
    - laço (do - while)
      - impressão na tela
- impressão na tela
- condição (else)
  - chamada da função POSICIONALIN
  - impressão na tela
  - leitura do arquivo COLUNA

laço (do - while)
- condição (if) --- comparador
  - impressão na tela
  - condição (if) --- comparador
    - impressão na tela
    - condição (if) --- comparador
      - impressão na tela
    - leitura do arquivo COLUNA

impressão na tela

Limpamatriz: Apaga somente o arquivo da matriz.


impressão na tela
comparador
condição (if) --- comparador
- atribuição (abre arquivo da Matriz)

Listagem: Mostra a matriz preenchida com os cálculos.

chamada da função ABREARQ

atribuição (variáveis)
posiciona o arquivo em um byte especifico (fseek Cptr,0,0)
laço (while) COLUNA
    | contador

atribuição (variáveis) → Divisor
posiciona o arquivo em um byte especifico (fseek Lptr,0,0)
laço (while)
    | contador

atribuição (variáveis) → Divisor
condição (if) --- comparador
    | atribuição (variáveis)
    | chamada da função QUADRANTE

condição (else)
    | laço (while) --- comparador
        | atribuição (variáveis)
        | laço (while) --- comparador
            | chamada da função QUADRANTE
            | atribuição (variáveis) → Somador
            | condição (if) --- comparador
                | atribuição
            | contador
        atribuição (variáveis) → Somador
        condição (if) --- comparador
            | atribuição
        contador

Ultimaposic: Calcula a última posição da matriz.

posiciona o arquivo em um byte especifico (fseek Mptr,0,0)
laço (while)

Busc_valor: Busca um valor na matriz (ordem pra percorrer o *traceback*).

posiciona o arquivo em um byte especifico (fseek Mptr,0,0)
laço (while)
    | condição (if) --- comparador
        | return

Traceback – Percorre a matriz em busca do melhor alinhamento.

```
impressão na tela
chamada da função ABREARQ
atribuição (variáveis)
chamada da função ULTIMAPOSIC
atribuição
laço (while) --- comparador
    | chamada da função POSICIONALIN
    | chamada da função POSICIONACOL
    | condição (if) --- comparador
    |           | atribuição (variáveis)
    |           | impressão na tela
    |  condição (else if)
    |           | atribuição (variáveis)
    |           | impressão na tela
    |    condição (else)
    |           | atribuição (variáveis)
    |           | impressão na tela
    | condição (if) --- comparador
    |           | impressão na tela
    |           | atribuição
    |
    | contador
    |
    | condição (if) --- comparador
    |     | chamada da função POSICIONA(M.linha-1,M.coluna-1)
    |     |
    |
    | condição (else)
    |     | atribuição (variáveis)
    |     | atribuição → chamada da função BUSC_VALOR(linha, coluna-1)
    |     | atribuição → chamada da função BUSC_VALOR(linha-1,coluna-1)
    |     | atribuição → chamada da função BUSC_VALOR(linha-1,coluna)
    |     | posiciona o arquivo em um byte especifico (fseek Mptr,0,0)
    |     | laço (while)
    |     |     | condição (if) --- comparador
    |     |     |     | break
    |     |     |
    |     |
    |     | condição (if) --- comparador
    |     |   |     chamada da função POSICIONA(M.linha,M.coluna-1)
    |     |   | condição (else if) --- comparador
    |     |   |     chamada da função POSICIONA(M.linha-1, M.coluna)
    |     |   | condição (else if) --- comparador
    |     |   |     chamada da função POSICIONA(M.linha-1,M.coluna-1)
  impressão na tela
```

Apêndice B

Neste apêndice, apresenta-se a implementação do algoritmo de alinhamento Local Otimizado em VHDL que fez uso da técnica de programação dinâmica.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity moronyglobal is
port(rst: in std_logic;
     clk: in std_logic;
     x : out std_logic_vector(3 downto 0);  --debug do estado corrente
     vlin, vcol: out bit_vector(7 downto 0);
     --S: out integer range 0 to 60;
     matrw: out integer range -80 to 80;
     matr1: out integer range -80 to 80;
     matr2: out integer range -80 to 80

     );

end moronyglobal;

architecture arch_moronyglobal of moronyglobal is

type VL0 is array (0 to 8) of bit_vector(7 downto 0);-- Declaracao do
vetor linha da matiz
type VL1 is array (0 to 8) of bit_vector(7 downto 0); --Declaracao do
vetor coluna da matiz

type matriz is array (0 to 8) of integer RANGE -80 TO 80; --
Declaracao da matiz

type matriz2 is array (0 to 8) of matriz;

signal mat: matriz2;                       --mat eh do tipo matriz

signal vlinha: VL0;                       --vlinha eh do tipo VL0 --
vetor que armazena a linha da matriz
signal vcoluna: VL1;                      --vcoluna eh do tipo VL1 --
vetor que armazena a coluna da matriz
signal estado  : std_logic_vector(3 downto 0); --Declarao do sinal dos
estados
signal W, Z : integer range -80 to 80;


-- Declaracoes dos estados da maquina
constant zero   : std_logic_vector(3 downto 0) := "0000";
constant um     : std_logic_vector(3 downto 0) := "0001";
constant dois   : std_logic_vector(3 downto 0) := "0010";
constant tres   : std_logic_vector(3 downto 0) := "0011";
constant quatro : std_logic_vector(3 downto 0) := "0100";
constant cinco  : std_logic_vector(3 downto 0) := "0101";

-- Declaracoes das letras das sequencias
constant A : bit_vector := "01000001";
constant T : bit_vector := "01010100";
```

```
constant C : bit_vector := "01000011";
constant G : bit_vector := "01000111";

constant igual     : integer range -80 to 80 := 5;
constant diferente : integer range -80 to 80  := -3;
constant espaco    : integer range -80 to 80  := -4;
constant espaco1    : integer range -80 to 80  := -4;

begin

      process(clk)
      variable i,j, S, V1, V2, V3, esquerda, diagonal, acima, linha,
coluna, ali_otimo : integer range -80 to 80;
      variable maior, y, aux1, aux2: integer;
      begin
      if rst='0' then
            estado <= zero;
      elsif clk'event and clk= '1' then

      --estado zero
      if estado = zero then  -- zera tudo

                if i <= 8 then  --Zerar todas as posicoes da matriz.
                   if j <= 8 then
                      mat(i)(j)<= 0;

                      j := j +1;
                   end if;

                   if j = 9 then
                      j := 0;
                      i := i + 1;
                   end if;

                   if i = 9 then
                      estado <= um;
                      i := 0;
                      j := 0;

                   end if;
                end if;


      --estado um
      elsif estado = um then -- Preenchimento da primeira linha e
primeira coluna do inicio ao fim
            if i <= 9 then
                  mat(i)(0) <= W;
                  W <= W + espaco;
                  matrw <= W;
                  i := i + 1;
            end if;


            if i = 10 then
                   if j <= 8 then
                        mat(0)(j) <= Z;
                        Z <= Z + espaco1;
                        matrw <= Z;
                        j := j +1;
                  end if;
```

```
            if j = 9 then
                  i := 0;
                  j := 0;
                  W <= 0;
                  estado <= dois;

            end if;

          end if;

          --ESTADO DOIS
      elsif estado = dois then  -- Inicializao das letras nos vetores
vlinha e vcoluna
            vlinha(1) <= G; vcoluna(1) <= G;
            vlinha(2) <= A; vcoluna(2) <= G;
            vlinha(3) <= A; vcoluna(3) <= A;
            vlinha(4) <= T; vcoluna(4) <= T;
            vlinha(5) <= T; vcoluna(5) <= C;
            vlinha(6) <= G; vcoluna(6) <= A;
            vlinha(7) <= C; vcoluna(7) <= T;
            vlinha(8) <= A; vcoluna(8) <= G;

      case i is -- Apresentacao das letras nos vetores vlinha e
vcoluna
            when 1 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 2 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 3 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 4 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 5 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 6 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 7 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when 8 => vlin <= vlinha(i); vcol <= vcoluna(i);
            when others => null;
      end case;
      i := i+1;

      if i = 9 then
            i := 1;
            j := 1;
            S := 0;
            V1 := 0;
            V2 := 0;
            V3 := 0;
            estado <= tres;
      end if;

      --ESTADO TRES
      elsif estado = tres then -- calculo para realizar o
preenchimento do restante da matriz

      if i <= 8 then
         if j <= 8 then

                  if (vlinha(i) = vcoluna(j)) then

                        S := igual;
                              matrw <= S;
                              vlin <= vlinha(i);
                              vcol <= vcoluna(j);
```

```
                        else
                              S := diferente;
                              matrw <= S;
                              vlin <= vlinha(i);
                              vcol <= vcoluna(j);
                        end if;

                        V1 := mat(i-1)(j-1) + S;

                  V2 := mat(i)(j-1) + espaco;

                  V3 := mat(i-1)(j)+ espaco;


                        if ((V1 >= V2) and (V1 >= V3)) then
                              mat(i)(j) <= V1;
                        matrw <= V1;

                        elsif ((V2 >= V1) and (V2 >= V3)) then
                        mat(i)(j) <= V2;
                        matrw <= V2;

                        elsif ((V3 >= V1) and (V3 >= V2)) then
                              mat(i)(j) <= V3;
                        matrw <= V3;

                        end if;

                  end if;

                  j := j + 1;

                  if j = 9 then
                        j := 1;
                        i := i + 1;

                  end if;
                  if i = 9 then
                        estado <= quatro;
                        i :=1;
                        j :=1;
                        y := 0;
                        maior :=0;
                        aux1 := 0;
                        aux2 := 0;
                  end if;

            end if;


      --ESTADO QUATRO
      elsif estado = quatro then  -- Busca e retorna o maior valor
dentro da matriz

      if i <= 8 then
            if j <= 8 then
                  y := mat(i)(j);
                  if y <= maior then
                              mat(i)(j) <= maior;

                        elsif y >= maior then
```

```
                        aux1 := i;
                        aux2:= j;

                        maior := y;
                        mat(i)(j) <= y;

                  matrw <= y;
                  matr1 <= aux1;
                  matr2 <= aux2;

            end if;
       end if;

            j := j + 1;

                  if j = 9 then
                        j := 1;
                        i := i + 1;

                  end if;

            if i = 9 then
                  estado <= cinco;
                  i :=0;
                  j :=0;
                  i := aux1;
            j := aux2;
            ali_otimo := 0;
            linha := 0;
            coluna := 0;
            esquerda := 0;
            diagonal := 0;
            acima := 0;
            end if;

      end if;


      --ESTADO CINCO
elsif estado = cinco then  -- Realizar o procedimento TRACEBACK

if i > 0 then
   if j > 0 then

      if vlinha(i) = vcoluna(j) then
            ali_otimo := ali_otimo + igual;
            mat(i)(j) <= ali_otimo;
            matrw <= ali_otimo;
            vlin <= vlinha(i);
            vcol <= vcoluna(j);

      elsif ((vlinha(i) = A ) or (vlinha(i) = T)) and
((vcoluna(j) = A) or (vcoluna(j) = T )) then
            ali_otimo := ali_otimo + diferente;
            mat(i)(j)<= ali_otimo;
                  matrw <= ali_otimo;
                  vlin <= vlinha(i);
                  vcol <= vcoluna(j);

      elsif ((vlinha(i) = C ) or (vlinha(i) = G)) and
((vcoluna(j) = C) or (vcoluna(j) = G )) then
```

```
                ali_otimo := ali_otimo + diferente;
                mat(i)(j)<= ali_otimo;
                matrw <= ali_otimo;
                vlin <= vlinha(i);
                      vcol <= vcoluna(j);

          else ali_otimo := ali_otimo + espaco;
                mat(i)(j)<= ali_otimo;
                      matrw <= ali_otimo;
                      vlin <= vlinha(i);
                      vcol <= vcoluna(j);

          end if;


                if (vlinha(i) = vcoluna(j)) then
                      j := j - 1;
                      i := i - 1;
                else

                      linha := j;

                      coluna := i;

                      esquerda := mat(i - 1)(j);

                      diagonal := mat(i - 1)(j - 1);

                      acima := mat(i)(j - 1);

                      if ((esquerda >= diagonal) and (esquerda >=
acima)) then

                            i := i - 1;

                      elsif ((acima>= diagonal) and (acima >=
esquerda)) then

                            j := j - 1;

                      elsif ((diagonal >= acima) and (diagonal >=
esquerda)) then
                            j := j - 1;
                            i := i - 1;
                      end if;
                end if;


      end if;

     end if;


    end if ;  -- Finaliza estados
  end if;
 end process;

  x <= estado;

end arch_moronyglobal;
```

Apêndice C

Neste apêndice, observa-se a ilustração dos cálculos realizados tanto na linguagem de programação C quanto em VHDL dos algoritmos Morony-Global e Morony-Local. Sendo que as seqüências utilizadas foram: Seqüência 1: **GAATTGCA,** Seqüência 2: **GGATCATG.** No algoritmo Morony-Global, a matriz obtida em C, cujos valores referentes a **match, mismatch e gap são: 5, -3, -4.**

			1	2	3	4	5	6	7	8
		-	G	A	A	T	T	G	C	A
	-	0	-4	-8	-12	-16	-20	-24	-28	-32
1	G	-4	5	1	-3	-7	-11	-15	-19	-23
2	G	-8	1	2	-2	-6	-10	-6	-10	-14
3	A	-12	-3	6	7	3	-1	-5	-9	-5
4	T	-16	-7	2	3	12	8	4	0	-4
5	C	-20	-11	-2	-1	8	9	5	9	5
6	A	-24	-15	-6	3	4	5	6	5	14
7	T	-28	-19	-10	-1	8	9	5	3	10
8	G	-32	-13	-14	-5	4	5	14	10	6

Logo, o alinhamento ótimo obtido corresponde ao valor 7:

A	-	-4
C	G	-3
G	G	5
T	T	5
T	A	-3
T	-	-4
T	T	5
A	A	5
A	-	-4
G	G	5
	TOTAL =	7

Da mesma maneira, com os mesmos dados o calculo em VHDL do algorimto Morony-Global apresentou a seguinte matriz de programação dinâmica.

			1	2	3	4	5	6	7	8
		-	G	A	A	T	T	G	C	A
	-	0	252	248	244	240	236	232	228	224
1	G	252	5	1	253	249	245	241	237	233
2	G	248	1	2	6	2	254	250	246	242
3	A	244	253	254	7	3	255	3	255	251
4	T	240	249	250	3	12	8	4	8	4
5	C	236	245	246	255	8	9	5	9	5
6	A	232	241	250	251	4	5	6	5	14
7	T	228	237	246	247	0	9	5	3	10
8	G	224	233	242	251	252	5	14	10	6

Contudo, o alinhamento ótimo recebe o valor 7. A representação do procedimento *traceback* está em formato hexadecimal. O que equivale em representação em Caracter, tendo como melhor alinhamento valor 7.

VLIN	41	43	47	54	54	54	54	41	41	47
VCOL	47	47	47	54	41	43	54	41	47	47
VLIN	A	C	G	T	T	T	T	A	A	G
VCOL	G	G	G	T	A	C	T	A	G	G

Logo, a representação do alinhamento ótimo é a seguinte:

A	-	-4
C	G	-3
G	G	5
T	T	5
T	A	-3
T	-	-4
T	T	5
A	A	5
A	-	-4
G	G	5
	TOTAL =	7

No algoritmo Morony-Local, a matriz obtida em C, cujos valores referentes a **match, mismatch e gap são: 5, -3, -4.**

			1	2	3	4	5	6	7	8
		-	G	A	A	T	T	G	C	A
	-	0	0	0	0	0	0	0	0	0
1	G	0	5	1	0	0	0	5	1	0
2	G	0	5	2	0	0	0	5	2	0
3	A	0	1	10	7	3	0	1	2	7
4	T	0	0	6	7	12	8	4	0	3
5	C	0	0	2	3	8	9	5	9	5
6	A	0	0	5	7	4	5	6	5	14
7	T	0	0	1	3	12	9	5	3	10
8	G	0	5	1	0	8	9	14	10	6

O maior valor da matriz correspode a 14 cuja posição encontra-se na linha 8 e coluna, cujo valor do melhor alinhamento corresponde a 14.

G	G	5
T	T	5
T	A	-3
T	-	-4
T	T	5
A	A	5
A	-	-4
G	G	5
	TOTAL =	14

Da mesma maneira, com os mesmos dados o calculo em VHDL do algoritmo Morony-Local apresentou a seguinte matriz de programação dinâmica.

			1	2	3	4	5	6	7	8
		-	G	A	A	T	T	G	C	A
	-	0	0	0	0	0	0	0	0	0
1	G	0	5	5	1	253	253	253	253	5
2	G	0	1	2	10	6	2	2	254	1
3	A	0	253	254	7	7	3	7	3	255
4	T	0	253	250	3	12	8	4	12	8
5	C	0	253	250	255	8	9	5	9	9
6	A	0	5	2	254	4	5	6	5	14
7	T	0	1	2	255	0	9	5	3	10
8	G	0	253	254	7	3	5	14	10	6

Contudo, o alinhamento ótimo recebe o valor 14. A representação do procedimento *traceback* está em formato hexadecimal.

VLIN	47	54	54	54	54	41	41	47
VCOL	47	54	41	43	54	41	47	47
VLIN	G	T	T	T	T	A	A	G
VCOL	G	T	A	C	T	A	G	G

Logo, o melhor alinhamento ou o alinhamento ótimo é 14.

G	G	5
T	T	5
T	A	-3
T	-	-4
T	T	5
A	A	5
A	-	-4
G	G	5
	TOTAL =	14