Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Edgar José Garcia Neto Segundo

Hardware paralelo reconfigurável para identificação de alinhamentos de sequências de DNA

Rio de Janeiro
2012

Edgar Jose Garcia Neto Segundo

Hardware paralelo reconfigurável para identificação de alinhamentos de sequências de DNA

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Orientadora: Prof.ª Dr.ª Nadia Nedjah
Coorientadora: Prof.ª Dr.ª Luiza de Macedo Mourelle

Rio de Janeiro
2012

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

G216 Garcia Neto Segundo, Edgar José.

Hardware paralelo reconfigurável para identificação de alinhamentos de sequências de DNA / Edgar José Garcia Neto Segundo. - 2012.

114 f.

Orientadora: Nadia Nedjah.

Coorientador: Luiza de Macedo Mourelle.

Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica. 2. Hardware paralelo – Dissertação. I. Nedjah, Nadia. II. Universidade do Estado do Rio de Janeiro. III. Título.

CDU 004.3

Assinatura

Data

Edgar José Garcia Neto Segundo

Hardware paralelo reconfigurável para identificação de alinhamentos de sequências de DNA

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Aprovado em 09 de Agosto de 2012

Banca Examinadora:

Prof.ª Dr.ª Nadia Nedjah (Orientadora)
Faculdade de Engenharia - UERJ

Prof.ª Dr.ª Luiza de Macedo Mourelle (Co-orientadora)
Faculdade de Engenharia - UERJ

Prof. Dr. Carlos Raimundo Erig Lima
Universidade Tecnológica Federal do Paraná - UTFPR

Prof. Dr. Leandro Augusto Justen Marzulo
Instituto de Matemática e Estatística, UERJ

Rio de Janeiro
2012

AGRADECIMENTOS

Agradeço aos meus pais, que são sem dúvida os professores mais importantes da minha vida.

A quem está do meu lado todos os dias, que me conforta, entende e atura, muito obrigado! Nada seria sem vocês que me amam de verdade, que são amigos para todas as horas. Obrigado pelo amor incondicional. Ninguém faz nada sozinho, essa conquista é nossa!

Agradeço às professoras Nadia Nedjah e Luiza de Macedo Mourelle, pela orientação neste trabalho, pela paciência e confiança que sempre depositaram em mim, e principalmente pela compreensão ao lidar com minha limitações. A experiencia e metodologia delas foi imprescindível para a conclusão desse projeto. Obrigado por tudo, professoras!

Aproveito para estender os meus fraternos agradecimentos aos professores do PEL-UERJ, por despertar em nós, alunos sonolentos e atarefados, o interesse pela inovação e pela busca contínua do conhecimento.

Por fim, agradeço a Deus, que me permitiu ousar e não me deixou desistir. Nada fazia sentido, mas Ele foi ligando todos os fatos, todas as pessoas, me concedendo sabedoria, entendimento, conselho, fortaleza, ciência, piedade e respeito. Obrigado Senhor por me permitir mais esse sonho.

RESUMO

NETO SEGUNDO, Edgar Jose Garcia *Hardware paralelo reconfigurável para identificação de alinhamentos de sequências de DNA*. 2012. 114f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2012.

Amostras de DNA são encontradas em fragmentos, obtidos em vestígios de uma cena de crime, ou coletados de amostras de cabelo ou sangue, para testes genéticos ou de paternidade. Para identificar se esse fragmento pertence ou não a uma sequência de DNA, é necessário compará-los com uma sequência determinada, que pode estar armazenada em um banco de dados para, por exemplo, apontar um suspeito. Para tal, é preciso uma ferramenta eficiente para realizar o alinhamento da sequência de DNA encontrada com a armazenada no banco de dados. O alinhamento de sequências de DNA, em inglês DNA *matching*, é o campo da bioinformática que tenta entender a relação entre as sequências genéticas e suas relações funcionais e parentais. Essa tarefa é frequentemente realizada através de *softwares* que varrem *clusters* de base de dados, demandando alto poder computacional, o que encarece o custo de um projeto de alinhamento de sequências de DNA. Esta dissertação apresenta uma arquitetura de *hardware* paralela, para o algoritmo BLAST, que permite o alinhamento de um par de sequências de DNA. O algoritmo BLAST é um método heurístico e atualmente é o mais rápido. A estratégia do BLAST é dividir as sequências originais em subsequências menores de tamanho w. Após realizar as comparações nessas pequenas subsequências, as etapas do BLAST analisam apenas as subsequências que forem idênticas. Com isso, o algoritmo diminui o número de testes e combinações necessárias para realizar o alinhamento. Para cada sequência idêntica há três etapas, a serem realizadas pelo algoritmo: semeadura, extensão e avaliação. A solução proposta se inspira nas características do algoritmo para implementar um *hardware* totalmente paralelo e com *pipeline* entre as etapas básicas do BLAST. A arquitetura de *hardware* proposta foi implementada em FPGA e os resultados obtidos mostram a comparação entre área ocupada, número de ciclos e máxima frequência de operação permitida, em função dos parâmetros de alinhamento. O resultado é uma arquitetura de *hardware* em lógica reconfigurável, escalável, eficiente e de baixo custo, capaz de alinhar pares de sequências utilizando o algoritmo BLAST.

Palavras-chave: Alinhamento de DNA. Bioinformática. Hardware. Arquiteturas Paralelas. Heurística.

ABSTRACT

DNA samples are found in fragments, obtained in traces of a crime scene, collected from hair or blood samples, for genetic or paternity tests. To identify whether this fragment belongs or not to a given DNA sequence it is necessary to compare it with a determined sequence which usually come from a database, for instance, to point asuspect. To this end, we need an efficient tool to perform the alignment of the DNA sequence found with the ones stored in the database. The alignment of DNA sequences, which is a field of bioinformatics that helps to understand the relationship between genetic sequences and their functional relationships and parenting. This task is often performed by software that scan clusters of databases, which requires high computing effort, thus increasing the cost of DNA sequences alignment projects. This work presents a parallel hardware architecture, for BLAST algorithm, to DNA pairwise alignment. This is the original version of the BLAST algorithm, that resulted in several other versions. The BLAST algorithm is a heuristic method and is the fastest algorithm for sequence alignment. The strategy of BLAST is to divide the sequences into smaller subsequences of size w. After making comparisons in these subsequences, algorithm steps analyzes only the subsequences that are identical. Thus, reducing the number of tests and combinations needed to perform the alignment. For each identical sequence found, three steps are followed by the algorithm: seeding, extension and evaluation. The proposed hardware architecture is based on the characteristics of the algorithm to implement a fully parallel hardware, where the basic steps of BLAST are pipelined. The proposed architecture was implemented in FPGA and the results show a comparison between the area occupied, number of cycles and maximum frequency of operation permitted, as a function of alignment parameters. The result is a hardware architecture in reconfigurable logic, scalable, efficient and with low cost, capable of aligning the pairs of sequences using BLAST algorithm.

Keywords: DNA Matching. Bioinformatics. Hardware. Parallel Architecture. Heuristic.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

INTRODUÇÃO

APESAR das descobertas sobre o DNA terem sido feitas há alguns anos atrás, os computadores da época não eram capazes de prover o desempenho necessário para realização de algumas tarefas, o que tornava algumas etapas das aplicações biológicas simplesmente inviáveis. Os avanços recentes na tecnologia da informação permitiram que os cientistas retomassem os estudos utilizando técnicas de informática como nova abordagem para a solução de antigos problemas do campo biológico, ou ainda para aprimorar métodos já utilizados no passado (MARKEL; LEON, 2003).

O campo que combina informática e biologia é denominado bioinformática, que tem entre os exemplos mais difundidos a pesquisa de genes em uma sequência de DNA, o desenvolvimento de métodos de predição de estruturas e/ou funções baseadas em conhecimentos biológicos, a clusterização de sequências de dados baseado em modelos de proteínas, além do alinhamento de sequências genéticas. Um dos principais desafios dessa área, e foco deste trabalho, consiste em alinhar sequências de DNA e entender a relação funcional que elas podem ter entre si.

Com esse intuito, algoritmos foram desenvolvidos especificamente para tentar reduzir o tempo levado para o alinhamento dessas sequências e para a interpretação dos resultados encontrados. Esses algoritmos são, em sua maioria, baseados em programação dinâmica e trabalham bem, com um tempo e custo aceitável para pequenas sequências de DNA, mas, em geral, a medida que as sequências aumentam, o tempo de processamento cresce exponencialmente (MOUNT, 2008a) (MOUNT, 2008b) (BAXEVANIS; OUELLETTE, 2004).

Por serem massivamente construídos em software, esses algoritmos submetem uma sequência de entrada, a comparação com uma sequência alvo, que pode estar armazenada em um banco de dados, procurando fragmentos da sequência de entrada na sequência alvo, através de métodos locais ou globais de busca, e obtendo como resposta um alinhamento ótimo entre as duas sequências. São algoritmos bastante difundidos na comunidade cientifica o de Needlman-Wunsh (NEEDLMAN; WUNSH, 1970), e o de Smith-Waterman (SMITH; WATERMAN, 1981), que trabalham com busca global e local, respectivamente. A maior diferença entre eles

é que enquanto o método global varre todo o espaço de busca, os métodos locais trabalharam em regiões reduzidas, zonas onde é maior a probabilidade que exista similaridade, e por isso acabam se tornando mais adequados para identificar regiões que os métodos globais não identificam (SEARLS, 2002).

O grande diferencial dos métodos baseados em programação dinâmica é que eles buscam sempre o melhor resultado possível. É claro que esse compromisso requer mais computação. Já os métodos baseados em heurística surgiram como uma alternativa mais viável em termos de custo e tempo se comparados a programação dinâmica (GIEGERICH, 2000). Ao invés de procurar o melhor alinhamento, esse tipo de método busca um grupo de soluções aceitáveis, descartando os alinhamentos indesejáveis, o que torna esse grupo de soluções uma opção que reduz muito o tempo para a realização do DNA *matching*. A estratégia desses algoritmos é bastante semelhante aos algoritmos genéticos, e os dois mais conhecidos são o BLAST e o FASTA (RUBIN; PIETROKOVSKI, 2003), que tem como característica macro etapas semelhantes de semeadura, extensão e avaliação bem definidas. Atualmente o BLAST é o mais rápido deles, e por isso foi escolhido como foco desse artigo que propõe: uma arquitetura paralela de *hardware* para implementação do algorítimo BLAST (BALDI; BRUNAK, 2001) (BAXEVANIS; OUELLETTE, 2004).

O algoritmo BLAST é um método heurístico de busca que procura alinhar sequências a partir de palavras coincidentes de um comprimento determinável que atinjam uma pontuação mínima, de acordo com uma tabela de pontos predeterminada. Fazendo analogia aos algoritmos genéticos, o BLAST trabalha com um processo correspondente ao de semeadura, onde procura regiões de similaridade para iniciar o processo de comparação. O processo de semeadura constitui uma grande vantagem dos métodos heurísticos, pois como poucos alinhamentos atingem a pontuação mínima,o espaço de busca é reduzido drasticamente logo na primeira etapa, porém, corre-se o risco de desconsiderar o melhor alinhamento possível. Após descartar os pares que não atingem a pontuação mínima, o BLAST tenta aumentar a pontuação desses alinhamentos e depois detecta a significância biológica que uma determinada similaridade pode ter (ALTSCHUL et al., 1990).

Comparado a outros métodos heurísticos, o BLAST executa o alinhamento de sequências de DNA e proteínas mais rapidamente do que o FASTA, considerando-se a mesma sensibilidade (RUBIN; PIETROKOVSKI, 2003). Por essa característica de agilidade no tempo de resposta, foi escolhido para se implementar uma arquitetura de *hardware* visando estabelecer a relação entre o tempo de processamento do algoritmo e os parâmetros de área e frequência

de uma plataforma FPGA (WOLF, 2004). Ao estabelecer essas relações, é possível verificar a viabilidade da implementação de uma arquitetura dedicada realizando o algoritmo BLAST em um sistema embutido (KORF; YANDELL; BEDELL, 2003). No seguimento, o restante deste trabalho está organizado da seguinte forma:

No Capítulo 1 apresenta-se o problema de alinhamento de sequências. É exposta a importância desse assunto na biologia, e o princípio dos métodos mais utilizados. São definidos nesse capítulo os conceitos fundamentais para o entendimento do problema, detalhada a representação e forma de codificação das sequências de DNA e RNA e, por fim, são descritos os principais algoritmos de alinhamento de sequência existentes.

No Capítulo 2 é feito uma explanação acerca do algoritmo BLAST. São apresentados as variações do algoritmo, distinguidas as etapas e detalhados os parâmetros do mesmo. É apresentado, além do funcionamento do algorítimo, sua formalização e complexidade matemática.

O Capítulo 3 apresenta a macro-arquitetura proposta para implementar o algoritmo BLAST em *hardware*. É descrito o paralelismo estrutural criado e o *pipeline* criado para otimizar a divisão do algoritmo em etapas. São descritos também os componentes utilizados para interface e controle dessas etapas. Alguns componentes chaves são detalhados, como o controlador global e o árbitro.

O Capítulo 4 foca na arquitetura da etapa de semeadura. É descrito o controlador de etapa, apresentados seus sinais de entrada e saída, e é detalhado o *hardware* paralelo criado para realizar as funções dessa etapa. São apresentados os resultados fundamentais de simulação para validação do modelo de *hardware* proposto.

No Capítulo 5 aborda-se a etapa de extensão. São explorados os processadores de extensão criados exclusivamente para solucionar a demanda dessa etapa. Há também o detalhamento dos sinais de entrada e saída, do controlador de etapa, e os resultados de simulação inerentes à mesma.

O Capítulo 6 apresenta os testes realizados através de simulação. Também é apresentado o resultado da síntese da arquitetura proposta em FPGA, explorando as possíveis variações para estabelecer relação entre os requisitos de tempo e área com os parâmetros de funcionamento do algoritmo BLAST.

O Capítulo 7 conclui esta dissertação, apresentando as principais conclusões obtidas com o desenvolvimento do presente trabalho. São apresentadas também possíveis melhorias para a arquitetura de *hardware* proposto e as direções para futuros estudos envolvendo o tema.

Capítulo 1

ALINHAMENTO DE SEQUÊNCIAS BIOLÓGICAS

ALINHAMENTO de sequências é uma forma de organizar as sequências de interesse para determinar alguma relação ou similaridade entre elas. Na biologia, é muito importante identificar e entender quais as relações que os elementos podem ter entre eles. O alinhamento da sequências biológicas é a base de muitos estudos evolutivos e comparativos, e os erros nos alinhamentos levam a erros na interpretação de informação evolutiva em genomas e nas aplicações subsequentes (LOYTYNOJA; GOLDMAN, 2008) (WATERMAN, 1995).

A bioinformática emerge como forma de suportar os problemas críticos de biologia, através de sistemas e aplicações otimizadoras. Uma das tarefas mais básicas, porém não menos importante, é fazer a comparação de sequências de espécies que estejam em estudo. Uma das maneiras de realizar essa comparação, e fruto dessa dissertação, é pela construção de um alinhamento das sequências de interesse e, para isso, existem diversos métodos. Portanto, um alinhamento de sequência biológica é uma ferramenta que, além de ser usada para analise de regiões conservadas e de regiões que sofreram mutações em sequências homologas, também serve como ponto de partida para outras aplicações em Biologia Computacional, como o estudo de estruturas secundárias de proteínas e a construção de árvores filogenéticas, sempre baseado na medida de similaridade (KORF; YANDELL; BEDELL, 2003) (MARKEL; LEON, 2003) (SETUBAL; MEIDANIS, 1997).

A forma de se medir a similaridade entre duas sequências é através da comparação entre as mesmas. Quando um par de sequências a ser comparado possui o mesmo tamanho, é possível comparar estas duas sequências sem nenhuma alteração ou adequação, no processo que se denomina *comparação exata*. Existem vários algoritmos que executam a comparação exata, embora seja mais comum encontrar sequências de tamanhos diferentes, e necessitar compará-las (BAXEVANIS; OUELLETTE, 2004). Quando isso ocorre, faz-se necessário primeiramente

encontrar uma posição em que as duas sequências melhor se relacionam. Como as sequências são de tamanhos diferentes, para que elas fiquem do mesmo tamanho, é necessário inserir espaços no início, no fim ou no meio das sequências, num processo denominado *inserção de espaços*. O processo de encontrar a melhor posição relativa entre sequências, com ou sem a inserção de espaços, denomina-se alinhamento de sequências. Apesar do termo correto ser comparação de sequências, como a maioria dos métodos de alinhamento de sequências utiliza algum critério de pontuação para determinar a melhor posição entre duas sequências, o processo de alinhamento acaba executando também um processo de comparação. Por esse motivo, e pelo fato da maioria das sequências de interesse serem de tamanhos diferentes, o termo alinhamento de sequências é mais utilizado por ser mais abrangente (MARKEL; LEON, 2003) (SEARLS, 2002).

O problema da comparação inexata de sequências pode ser formalizado em termos de alinhamentos e similaridade. Dadas duas sequências s e t sobre o mesmo alfabeto, um alinhamento entre elas consiste na inserção de zero ou mais espaços em quaisquer posições de s e t, de tal forma que s e t tenham o mesmo tamanho e tal que se s_i é um espaço então t_i não é um espaço e vice-versa. Depois do acréscimo dos espaços, já que as sequências tem o mesmo tamanho, elas podem ser colocadas uma sobre a outra, mostrando a correspondência entre os caracteres e espaços na mesma posição, realizando uma comparação e classificando-as por algum método de pontuação pré-determinado.

Algumas importantes aplicações são derivadas do alinhamento de sequências genéticas, tais como:

- Comparação de sequências;
- Predição de função, predição de secundária e inferência filogenética;
- Localização de trechos conservados entre genomas;
- Comparação de uma sequência desconhecida com bancos de dados de sequências com funções conhecidas;
- Reconstrução da sequência original a partir da sobreposição de fragmentos de sequências (montagem).

Embora o foco da dissertação seja apresentar uma arquitetura de hardware para o alinhamento de sequências genéticas, antes de direcionar o foco para aplicação, apresentação dos algoritmos, paralelização das etapas, descrição do *hardware*, e toda a parte que será abordada nos capítulos posteriores é preciso recorrer à Biologia para elucidar o problema e entender a

importância do alinhamento de sequências e suas aplicações, a partir de um exemplo ilustrativo, que segue:

No estudo de evolução e de funções de moléculas biológicas é comum a necessidade de comparar moléculas de várias espécies ou seres. Por exemplo, pode-se estar interessados em conhecer quais são as relações de parentesco entre o HIV presente em uma dada pessoa e as cepas [1] de HIV para as quais se sabe da eficácia ou ineficácia de determinadas drogas antes de prescrever um coquetel de AIDS. Dessa maneira, drogas que não fossem úteis para o tratamento da pessoa não seriam prescritas ou seriam substituíveis. Com isso, o tratamento poderia ser mais bem sucedido, resultando também em uma possível redução de recursos destinados à compra de medicamentos e em uma diminuição dos efeitos colaterais das drogas sofridos pelo paciente. Em geral, as moléculas que se consideram nesse tipo de estudo são moléculas de DNA, de RNA e de proteínas. Como tais moléculas são polímeros, que podem ser representadas de maneira fácil por uma sequência de caracteres, comparar as moléculas resume-se, na pratica, a fazer uma comparação das sequências correspondentes (BRITO, 2003). A forma de representação de cada tipo de molécula em caracteres é apresentada na Seção 1.1.

A partir da interpretação das sequências biológicas como sequências de caracteres, é possível relacionar que uma sequência biológica é, por sua vez, uma sequência de bits. Essa ponte será feita durante todo o curso dessa dissertação, de forma que ao se trabalhar com os bits, buscando-os, comparando-os e armazenando-os estar-se-a trabalhando sobre uma informação biológica. Entretanto, a partir do momento que essa sequência é representada em bits, pode-se abstrair o significado biológico e enfrentar os requerimentos de um problema de alinhamento como qualquer outro problema na área computacional. Antes porém, apresenta-se na Seção 1.1 o alfabeto e a nomenclatura utilizada no curso da presente Dissertação e os conceitos básicos indispensáveis ao entendimento de uma aplicação de biologia molecular (SEARLS, 2002).

1.1 Conceitos Básicos de Biologia Molecular

Biologia é o estudo da vida. Os organismos vivos, tanto nas formas complexas quanto nas mais simples possuem a mesma bioquímica, onde os principais componentes são as moléculas chamadas proteínas e ácidos nucleicos. A grosso modo, as proteínas são responsáveis pela estruturação física dos organismos vivos, enquanto os ácidos nucleicos codificam a informação necessária para produção proteínas e são responsáveis por passar esta informação às gerações

[1] Quando um vírus sofre uma mutação, o fruto dessa mutação é denominado cepa.

seguintes (SETUBAL; MEIDANIS, 1997) (SMITH; WATERMAN, 1981) (ZAHA; FERREIRA; PASSAGLIA, 2003) (WATERMAN, 1995).

Nas Seções seguintes serão definidos importantes termos da Biologia Molecular, e também serão definidas as representações que nortearão esta dissertação. Na Seção 1.1.1 é abordado o alfabeto de representação, e como é feita a analogia entre sequência biológica e sequência de caracteres. Na Seção 1.1.2 são apresentados os conceitos chaves da comparação de sequências genéticas.

1.1.1 Alfabeto

Existem basicamente dois grupos nos quais há interesse na comparação e alinhamento de suas sequências: DNA e proteínas. Uma molécula de DNA é composta por nucleotídeos, enquanto a uma molécula de proteína é composta por aminoácidos. Uma molécula de DNA pode ser descrita completamente se a sequência de nucleotídeos que a compõem for conhecida, e, no caso das proteínas se for conhecida a sequência de aminoácidos que a compõem. A determinação da sequência exata chama-se sequenciamento desta molécula (ZAHA; FERREIRA; PASSAGLIA, 2003).

Cada nucleotídeo pertencente a sequências de DNA é representado por uma das quatro letras de um alfabeto denominado alfabeto base das sequências de DNA, definido como A_{DNA} = {A, C, G, T}. Na Tabela 1 são apresentadas as bases e os nucleotídeos correspondentes.

Tabela 1: Nucleotídeos encontrados em DNA

Símbolo	Nucleotídeo
A	Adenina
C	Citosina
G	Guanina
T	Timina

Cada aminoácido pertencente à sequência de Proteínas é representado por uma das vinte letras do alfabeto denominado alfabeto base das sequências de proteínas, definido como: $A_{Proteinas}$ = {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}. Na Tabela 2 são apresentadas as bases e seus respectivas abreviações e aminoácidos correspondentes (BAXEVANIS; OUELLETTE, 2004).

Denomina-se *base* cada letra (nucleotídeo ou aminoácido) do alfabeto de sequências de DNA ou Proteínas que compõem A_{DNA} ou $A_{Proteinas}$ respectivamente. A comparação de sequências é a comparação das bases em busca de igualdade.

Tabela 2: Aminoácidos usualmente encontrados em proteínas

Símbolo	Abreviação	Aminoácido
A	Ala	Alanina
C	Cys	Cistina
D	Asp	Ácido Aspártico
E	Glu	Glutamato
F	Phe	Fenilalanina
G	Gly	Glicina
H	His	Histidina
I	Ile	Isoleucina
K	Lys	Lysina
L	Leu	Leucina
M	Met	Metionina
N	Asn	Asparagina
P	Pro	Prolina
Q	Gln	Glutamina
R	Arg	Arginina
S	Ser	Serina
T	Thr	Treonina
V	Val	Valina
W	Trp	Triptofano
Y	Tyr	Tirosina

Uma *sequência* ou cadeia é uma sucessão ordenada de símbolos de um alfabeto. O tamanho ou comprimento de uma sequência s, é o número de símbolos, bases, contidos em s, representado por $|s|$. De forma análoga, o comprimento de uma sequência t, é o número de símbolos, bases, contidos em t, representado por $|t|$. Quando se realiza a comparação entre duas sequências, s e t, busca-se encontrar a similaridade entre ambas as sequências. Nessa dissertação, será utilizado $|s| = m$ e $|t| = n$, onde m e n definirão os comprimentos das sequências, refletindo o número de bases de cada uma delas. O termo *similaridade* refere-se a medida do quanto as sequências são parecidas, expressando uma medida da porcentagem de nucleotídeos ou de aminoácidos com propriedades químicas semelhantes. Logo, a partir de duas sequências de DNA ou proteínas, a forma de verificar o quanto essas duas sequências são quimicamente semelhantes, identificando sua similaridade, é através da comparação entre elas.

1.1.2 Comparação de sequências

A comparação de sequências é uma operação primitiva de suma importância em bioinformática, pois serve como subsídio para outras operações mais complexas. Comparar sequências significa encontrar quais partes das sequências são parecidas e quais são diferentes. Uma das utilidades da comparação de sequências é determinar um ancestral comum a partir de um conjunto

de sequências. Por exemplo, a partir das sequências de seres humanos e de alguns primatas é possível saber a sequência hipotética de uma espécie ancestral de ambos extinta. Se duas sequências têm um ancestral comuns são ditas *homologas*. É muito provável que duas sequências muito parecidas sejam homologas. A descoberta de homologia entre sequências de uma proteína ou famílias de proteínas frequentemente traz as primeiras pistas a respeito da função de um novo gene sequenciado. Todavia, é possível que duas sequências não muito parecidas sejam homologas. Ainda mais, também se faz possível que duas sequências sejam parecidas embora não sejam homologas (MOUNT, 2007) (GIBAS; JAMBECK, 2001). A interpretação da homologia e o significado biológico é uma etapa posterior a comparação das sequências.

Uma forma de realizar a comparação de sequências é mediante o alinhamento. A tarefa de alinhamento é utilizada tanto na comparação de sequências de DNA quanto na de proteínas, portanto independe do tipo de molécula em face, se são moléculas de DNA ou de proteínas, o que muda é o alfabeto utilizado. Todavia, conforme exposto na Seção 1.1.1, as moléculas de DNA possuem menos símbolos ou bases que uma molécula de proteínas. Dessa forma, a complexidade de uma comparação entre sequências de proteínas e de DNA são diferentes, e essa peculiaridade é descrita na Seção 1.2.

Como exposto, alinhar sequências é colocar uma sequência sobre a outra de forma que a correspondência entre elas fique evidente. Existem inúmeros métodos para realizar o alinhamento de sequências, dentre os quais se destacam os métodos de matriz de pontos, força bruta, programação dinâmica, além dos métodos baseados em heurística. A maior parte desses métodos alinham duas sequências biológicas e por isso são denominados alinhamento de pares de sequências. Existem também métodos para alinhar múltiplas sequências, porém esse não é o foco dessa Dissertação.

Um alinhamento de sequências de DNA ou proteínas pode ser obtido introduzindo espaços dentro ou nos extremos das sequências a serem alinhadas, de forma que a ordem das bases em cada sequência seja preservada e as sequências tenham o mesmo comprimento, visando realizar uma comparação exata. Esse é o princípio da matriz de pontos, que dispõe as duas sequências a serem alinhadas, uma na horizontal e outra na vertical, de forma que a identificação possa ser visual. O método do alinhamento por matriz de pontos é intuitivo, e ilustra bem o objetivo de uma comparação de sequências através do alinhamento das mesmas. Na Tabela 3, estão duas sequências, $s = \{\text{ACGTACGTAC}\}$ e $t = \{\text{GGTTACGGTC}\}$, alinhadas através de matriz de pontos, obedecendo os passos: dispor uma sequência em uma linha e a outra sequência em uma coluna; preencher com 1 em todas as posições onde houver similaridade.

Ao fim, as diagonais revelam a similaridade entre as duas sequências, e o melhor alinhamento é visto em: ACG**TACG**TAC e GGT**TACG**GTC. O processo é visto na Tabela 3 (ZAHA; FERREIRA; PASSAGLIA, 2003).

Tabela 3: Alinhamento de um par de sequências por matriz de pontos

	A	C	G	T	A	C	G	T	A	C
G			1				1			
G			1				1			
T				1				1		
T				1				1		
A	1				1				1	
C		1				1				1
G			1				1			
G			1				1			
T				1				1		
C		1				1				1

Uma matriz de pontos simples é primariamente um método para comparação de duas sequências pela observação de possíveis alinhamentos de caracteres entre as sequências a serem analisadas. Numa matriz de pontos, utiliza-se uma sequência de tamanho m como eixo horizontal e a outra sequência de tamanho n como eixo vertical. Cada célula da matriz de pontos é resultado da comparação da posição i da sequência s com o elemento j da sequência t, conforme apresentado em pseudocódigo no Algoritmo 1.

Algoritmo 1 Algoritmo simples para matriz de pontos

1: **Para** $i = 0 \rightarrow m$ **Faça**
2: **Para** $j = 0 \rightarrow n$ **Faça**
3: **Se** $s_i = t_j$ **Então**
4: $Pontos[i, j] \leftarrow 1$
5: **Senão**
6: $Pontos[i, j] \leftarrow 0$
7: **Fim Se**
8: **Fim Para**
9: **Fim Para**
10: **Retorna** *Pontos*

A matriz de pontos pode ainda inserir espaços nas sequências para que elas fiquem do mesmo tamanho, facilitando assim a visualização das coincidências. Obviamente, o método da matriz de pontos não é escalável, pois o tempo de execução e o espaço necessário para armazenar tal matriz de pontos é dado pela multiplicação $O(m \times n)$. Uma das maiores vantagens do método para encontrar alinhamentos utilizando a matriz de pontos, é que todos os possíveis encontros de resíduos entre as duas sequências são exibidos (MOUNT, 2004).

Embora seja um método não escalável, a matriz de pontos serve apenas como introdução a qualquer método de alinhamento de sequência genética. O exemplo ilustrado na Tabela 3 evidencia a necessidade de realizar o processo de forma manual e possibilita entender o uso da computação no processo de alinhamento de sequências para tamanhos maiores. De fato, a grande utilidade da computação no processo de alinhamento entre duas sequências reside no fato que a quantidade de símbolos em uma sequência pode ser muito grande, sendo ainda que a busca do alinhamento ótimo pode levar a uma explosão combinatória, tornando o trabalho extremamente custoso.

Outro método extremamente simples, mas que é base para o entendimento dos demais algoritmos utilizados na comparação de sequências, é o método da força bruta. Ele é apenas utilizado para determinar se duas sequências são idênticas ou não, sem reportar qualquer grau de similaridade. O algoritmo segue os seguintes passos: primeiro compara-se os primeiros elementos de ambas sequências; depois os segundos, os terceiros e assim sucessivamente, até encontrar um erro e retornar que as sequências não são idênticas ou chegar no fim das sequências e retornar que ambas são idênticas. O algoritmo está especificado em pseudocódigo no Algoritmo 2, para duas sequências s e t de tamanhos m e n respectivamente.

Algoritmo 2 Algoritmo força bruta para comparação de sequências

1: **Se** $m \neq n$ **Então**
2: **Retorna** $s \neq t$
3: **Senão**
4: **Para** $i = 0 \rightarrow m$ **Faça**
5: **Se** $s_i \neq t_i$ **Então**
6: **Retorna** $s \neq t$
7: **Fim Se**
8: **Fim Para**
9: **Retorna** $s = t$
10: **Fim Se**

No tocante a quantidade máximas de operações, o algoritmo da força bruta apresenta $O(n)$ operações necessárias para obtenção do resultado neste algoritmo. Se a sequência é uma lista de elementos aleatórios de um alfabeto de K letras, a probabilidade de acerto, isto é, a probabilidade dos dois elementos das listas que estão sendo comparados serem iguais, em qualquer posição é $1/K$ (BRITO, 2003). Em sequências com alfabetos grandes e com conteúdos aleatórios, a probabilidade de acerto será muito baixa, e normalmente serão necessárias poucas comparações para determinar que as duas sequências são diferentes. Porém, se o alfabeto possuir um número reduzido de elementos, como no caso do DNA, onde seu alfabeto é constituído de somente quatro letras e os padrões de repetições são comuns, o número de comparações

irá aumentar significativamente, desta forma, aumentando o tempo de execução do algoritmo. Para o algoritmo de força bruta, o pior caso é quando as sequências s e t são iguais, portanto é necessário realizar n comparações, pois como apresentado no Algoritmo 2, ocorre o teste de tamanho das sequências, embora as sequências possam ter o mesmo tamanho e ainda assim serem diferentes.

Com uma complexidade de $O(n)$, ou $O(m)$, o tempo de execução do algoritmo de força bruta varia consideravelmente de acordo com a natureza das sequências a serem analisadas e os resultados apenas informam se elas são idênticas ou não. Apesar do campo da comparação de sequências genéticas ter como viés mais importante a identificação da homogeneidade entre as sequências, este algoritmo não deve ser descartado, pois ele é básico e utilizado dentro de outros algoritmos mais eficientes, principalmente nos métodos baseados em heurísticas. As variações deste algoritmo e outros métodos de comparação exata de sequências formam um grupo relativamente extenso (CHARRAS; LECROQ, 1998).

Na Seção 1.2, serão abordados alguns algoritmos de comparação e alinhamento de sequências genéticas, e detalhados suas diferenças e aplicações. Esses algoritmos se subdividem, quanto ao tipo de alinhamento realizado, em três grandes grupos: alinhamentos locais, alinhamentos globais e alinhamentos semi-globais, e são objeto da Seção 1.2.

1.2 Tipos de Alinhamento

Por causa de grandes projetos de sequenciamento genômicos, por exemplo, o projeto do genoma humano que visa sequenciar, um a um, os genes que codificam as proteínas do corpo humano e também aquelas sequências de DNA que não são genes, existe um crescimento exponencial na quantidade de dados disponíveis sobre sequências genéticas e proteicas. Os métodos tradicionais de laboratório para o estudo da estrutura e função destas moléculas não são capazes de acompanhar a taxa de crescimento de novas informações. Como consequência, biólogos moleculares passaram a utilizar métodos estatísticos e computacionais capazes de analisar estas grandes quantidade de dados de forma mais automatizada. O alinhamento de sequências é útil para a descoberta de informações funcionais, estruturais e evolucionárias nas sequências biológicas. Partindo da premissa de que duas sequências similares possuem estruturas e comportamentos similares, pode-se inferir informações sobre uma nova sequência comparando-a a outras já previamente conhecidas (MOUNT, 2004) (MOUNT, 2007).

Existem diversas formas de comparações e alinhamento de sequências genéticas. Nesta Seção, pretende-se dar uma introdução sobre os principais e mais utilizados métodos de com-

paração de sequências genéticas, adicionando aos mais básicos, o algoritmo de força bruta e a utilização de matriz de pontos, já explicados na Seção anterior. É importante diferenciar dois conceitos para os algoritmos de alinhamento de sequências: método e estratégia. O método que esse algoritmo utiliza para encontrar a resposta, é a ferramenta utilizada pelo algoritmo: força bruta,de matriz de pontos, programação dinâmica, heurística e modelos probabilísticos. A estratégia utilizada por um algoritmo é relacionado ao tipo de alinhamento que o algoritmo faz.

Abordagens computacionais para o alinhamento de sequências dividem-se, em geral, em duas categorias: alinhamentos globais e alinhamentos locais. Todavia, existe um intermediário entre esses dois extremos que é o alinhamento semi-global. O tipo de alinhamento é relacionado com o tamanho das sequências que são comparadas de uma única vez. É possível, portanto, encontrar um algoritmo baseado em programação dinâmica que realize o alinhamento local, da mesma forma que é viável a um algoritmo de programação dinâmica realizar um alinhamento global.

Calcular um alinhamento global é uma forma de otimização global que "força" o alinhamento a cobrir todo o comprimento de todas as sequências interrogadas (KORF; YANDELL; BEDELL, 2003) (MARKEL; LEON, 2003) (SETUBAL; MEIDANIS, 1997). Em outra vertente, os alinhamentos locais identificam regiões de similaridade dentro de sequências longas que são geralmente bastante divergentes em um todo. Os alinhamentos locais são frequentemente preferíveis, mas podem ser difíceis de calcular por causa do problema adicional de identificar regiões internas de similaridade, pois pedaços muito pequenos de sequências são difíceis de identificar.

No tocante a estratégia do algoritmo, a divisão entre os tipos de alinhamento, local e global, é a base do agrupamento entre os algoritmos de alinhamento de sequências. Existe uma grande variedade de algoritmos para abordar o problema de alinhamento de sequências, sendo os mais conhecidos os baseados em programação dinâmica, mais lentos porém teoricamente otimizadores, ou baseados em heurística, mais eficientes/rápidos mas sem prova formal de obtenção de solução ótima (KORF; YANDELL; BEDELL, 2003) (MARKEL; LEON, 2003) (SETUBAL; MEIDANIS, 1997). De forma breve, destaca-se a diferença entre os principais grupos, na definição abaixo, e após são apresentados os principais algoritmos de cada um desses grupos:

- Global: alinhamento de pontuação máxima envolvendo as duas sequências completas. Desejável em situações onde as sequências são similares, por exemplo, ao se alinhar genes ou proteínas homologas. Para o alinhamento global, o total das posições de um

alinhamento é considerado (BAFNA; LAWLER; PEVZNER, 1997) (DURBIN et al., 1998) (WATERMAN, 1995);

- Semi-Global (ou Semi-Local): Um alinhamento semi-global é aquele onde a pontuação de um alinhamento é calculada ignorando os espaços terminais nas sequências. Os espaços terminais de uma sequência são aqueles que estão localizados antes do primeiro caractere ou depois do último caractere de uma sequência. Não penaliza espaços criados nas pontas das sequências. Desejável, por exemplo, no caso de montagem de genomas, onde busca-se um alinhamento de pontuação máxima entre o prefixo de uma sequência e o sufixo da outra (ou vice-versa) (BAFNA; LAWLER; PEVZNER, 1997) (DURBIN et al., 1998) (WATERMAN, 1995);

- Local: alinhamento de pontuação máxima entre qualquer par de subsequências (das sequências originais). Desejável, por exemplo, para se identificar trechos altamente conservados entre dois genomas (BAFNA; LAWLER; PEVZNER, 1997) (DURBIN et al., 1998) (WATERMAN, 1995). Um alinhamento local entre duas sequências s e t acontece entre uma subsequência de s e uma subsequência de t. A pontuação é calculada apenas sobre a região onde as sub-sequências estão alinhadas.

1.2.1 Alinhamento Global

Pode-se definir comparação global de duas sequências como sendo a comparação das duas sequências por inteiro, sem subdividi-las. O alinhamento global consiste na inserção de espaços nas sequências para deixá-las do mesmo tamanho, permitindo a comparação dos símbolos por posição, e permitindo a utilização de um sistema de pontuação para avaliação do alinhamento obtido.

Para se alinhar duas sequências, é possível inserir um espaço em qualquer posição da sequência, como um carácter coringa. O objetivo de um alinhamento global é obter a maior pontuação possível, ou pontuação ótima. O sistema mais simples de pontuação consiste em uma penalidade por alinhar um símbolo e um espaço, uma pontuação para alinhar dois símbolos diferentes, e ainda uma pontuação para alinhar dois símbolos idênticos.

O custo de tempo e espaço de um alinhamento global é $O(mn)$, onde m e n são os tamanhos das sequências a serem alinhadas. O algoritmo utiliza uma matriz de similaridade para construir o alinhamento entre duas sequências s e t. Esta matriz contém $(m+1)(n+1)$ posições indexadas de numa matriz $m \times n$, e o valor da posição $[i, j]$ da matriz é o valor do

alinhamento dos prefixos s_i e t_i das sequências s e t (MOUNT, 2004) (BAXEVANIS; OUELLETTE, 2004).

Para definir-se o passo a passo de um alinhamento global, defini-se como *gap* o valor da pontuação que recebe a inserção de um espaço na sequência. Esse valor é escolhido de acordo com a matriz de pontuação utilizada. As matrizes de pontuação são apresentadas na Seção 1.3.

Inicialmente os elementos da primeira linha recebem valor $-(gap \times j)$, e os elementos da primeira coluna recebem o valor $-(i \times gap)$. O algoritmo percorre a matriz da esquerda para a direita e de cima para baixo, calculando a similaridade de prefixos cada vez maiores. Dada a posição (i, j), o valor da similaridade sim é dado por $sim[i, j] = max(sim[i-1, j-1] + p(i, j), sim[i-1, j] - gap, sim[i, j-1] - gap)$ onde $p(i, j)$ representa o valor do alinhamento entre $s[i]$ e $t[j]$ (MOUNT, 2004) (BAXEVANIS; OUELLETTE, 2004).

Ao final do algoritmo, o valor do alinhamento ótimo se encontra na posição [m, n] da matriz. Para reconstruir-se o alinhamento, parte-se de $[m, n]$, e percorre-se a matriz, sempre voltando para a posição que originou o valor na posição corrente.

1.2.2 Alinhamento Local

A comparação local se interessa apenas pelas subsequências das sequências originais que geram a maior pontuação possível. Para se obter um alinhamento local, devem ser feitas as seguintes modificações no algoritmo de alinhamento global:

- Iniciar com 0 a primeira linha e coluna da matriz;
- Não deixar que os valores da matriz fiquem negativos; se o máximo se revelar negativo, será substituído por 0;
- Procurar maior valor da matriz inteira, tanto para obter a similaridade quanto para utilizar como ponto de partida para reconstrução de um alinhamento ótimo;
- Pode-se parar a reconstrução de um alinhamento quando encontrar um 0 na matriz.

Dadas duas sequências s e t, pode se dizer que todo alinhamento global é também um alinhamento local, mas nem todo alinhamento local é um alinhamento global.

Por se tratar o alinhamento semi-global de um caso específico do alinhamento global, que ignora espaços criados nas pontas das sequências, nas Seções são definidos apenas os alinhamentos global e local, com os exemplos de seus algoritmos mais utilizados. Para perfeito

entendimento dos alinhamentos global e local, é explicado na Seção 1.3 o sistema de pontuação e suas matrizes mais importantes, enquanto na Seção 1.4 são apresentados os principais algoritmos para alinhamento dos pares de sequência.

1.3 Pontuação

Matrizes de comparação analisam as frequências relativas com que ocorrem as diferentes substituições de aminoácidos. Com bases nestas frequências e com a abundância relativa de cada aminoácido na proteína é possível atribuir um escore que reflete a probabilidade daquela mutação ocorrer (*provaveis* ← *escore positivo*). Os dois tipos mais utilizados de matrizes são a PAM (*Point Accepted Mutation*) e a BLOSUM (*BLOcks Substitution Matrix*) , detalhadas nas Seções 1.3.1 e 1.3.2 (MOUNT, 2008a) (MOUNT, 2008b).

Na Seção 1.1.2, foram apresentados métodos intuitivos para o alinhamento de sequências. Como descrito, sequências curtas podem ser alinhadas manualmente, mas há programas gratuitos que fazem este serviço. De qualquer forma, sequências de menos de 200 nucleotídeos ou menos de 100 aminoácidos devem ser evitadas - ficaria difícil assumir se a similaridade tem alguma relevância evolutiva ou se é coincidência. É difícil encontrar o significado biológico para alinhamentos de pequeno comprimento. Estes programas verificam todas as possibilidades de alinhamento e fornecem um valor relativo, chamado pontuação (ou *score*, em inglês). Quanto maior este valor, maior é o número de nucleotídeos ou aminoácidos alinhados e, em última instância, maior a similaridade destas sequências.

O problema reside no fato de não ser possível simplesmente copiar as sequências no programa e esperar o resultado. Durante a evolução, as sequências destes organismos podem ter divergido (adição ou deleção de nucleotídeos e consequentemente de aminoácidos), tornando necessária a inserção de espaços. A adição de espaços maximiza o número de nucleotídeos ou aminoácidos alinhados, mas causa uma penalidade (*gap penalty*), que diminui o valor do escore. A ideia é limitar a adição de espaços, diminuindo as chances de obter uma porcentagem de similaridade falsa. Embora os programas de alinhamento sejam ferramentas importantes, o processo ainda não foi automatizado - uma análise crítica e cautelosa feita por uma pessoa ainda é indispensável neste procedimento.

Para alinhamento de proteínas, o método de pontuação simples aplicado ao DNA não é suficiente. Os aminoácidos que compõem as proteínas possuem propriedades bioquímicas que determinam como eles são substituídos durante a evolução. Por exemplo, existe uma maior probabilidade de que um aminoácido seja substituído por um outro de igual tamanho em lugar

de um aminoácido maior. Dado que a comparação de proteínas é feita frequentemente com critérios evolutivos, é necessário um esquema de pontuação que leve em conta estas probabilidades (SETUBAL; MEIDANIS, 1997).

Uma forma de pontuação muito usada envolve as chamadas matrizes de substituição. A seguir serão brevemente descritas as duas famílias de matrizes de substituição: PAM (DAYHOFF; SCHWARTZ; ORCUTT, 1978) e BLOSUM (HENIKOFF; HENIKOFF, 1992) que são as mais usadas (MOUNT, 2008a) (MOUNT, 2008b).

1.3.1 Matrizes PAM

Uma das primeiras matrizes de substituição de aminoácidos, a matriz PAM (*Point Accepted Mutation*) foi desenvolvida por Margareth Dayhoff em 1978 (DAYHOFF; SCHWARTZ; ORCUTT, 1978). Esta matriz é calculada observando diferenças em proteínas proximamente relacionadas. A matriz PAM1 estima que taxa de substituição deverá ser esperada se 1% dos aminoácidos muda. A matriz PAM1 é usada como base para calcular outras matrizes sobre a hipótese que mutações repetidas deveram seguir o mesmo patrão que PAM1, e múltiplas substituições podem ocorrer no mesmo sítio. Usando esta lógica, Dayhoff derivou o PAM250, o número indica a unidade evolucionária, que é obtida fazendo 250 multiplicações de PAM1.

As matrizes PAM (*Point Accepted Mutations* ou *Percent of Accepted Mutations*) fornecem uma medida de evolução produzida por, em média, uma mutação por cada 100 aminoácidos. Cada matriz PAM foi desenvolvida para comparar duas sequências, as quais estão separadas por uma distancia especificada em unidades PAM. Por exemplo a matriz PAM120 serve para comparar sequências que estão separadas por 120 unidades PAM. Diz-se que duas sequências s_1 e s_2 divergem em uma unidade PAM, se a série de mutações aceitas para converter s_1 em s_2 estão na ordem de uma mutação para cada 100 aminoácidos. Uma mutação é dita aceita quando não altera o funcionamento da proteína ou quando da mudança na proteína é benéfica para o organismo. Cada valor (i, j) da matriz PAM representa a pontuação para substituir um aminoácido A_i por um A_j. A Tabela 4 mostra a matriz PAM120 (DAYHOFF; SCHWARTZ; ORCUTT, 1978).

1.3.2 Matrizes BLOSUM

A metodologia de Dayhoff comparando espécies proximamente relacionadas não é adequada para trabalhar com alinhamentos de sequências divergentes (sequências de relacionamento distante). Mudanças de sequências sobre escalas de tempo evolucionarias longas não são bem

Tabela 4: Exemplo de uma matriz de substituição da família PAM.

Cys	12																			
Ser	0	2																		
Thr	-2	1	3																	
Pro	-3	1	0	6																
Ala	-2	1	1	1	2															
Gly	-3	1	0	-1	1	5														
Asn	-4	1	0	-1	0	0	2													
Asp	-5	0	0	-1	0	1	2	4												
Glu	-5	0	0	-1	0	0	1	3	4											
Gln	-5	-1	-1	0	0	-1	1	2	2	4										
His	-3	-1	-1	0	-1	-2	2	1	1	3	6									
Arg	-4	0	-1	0	-2	-3	0	-1	-1	1	2	6								
Lys	-5	0	0	-1	-1	-2	1	0	0	1	0	3	5							
Met	-5	-2	-1	-2	-1	-3	-2	-3	-2	-1	-2	0	0	6						
Ile	-2	-1	0	-2	-1	-3	-2	-2	-2	-2	-2	-2	-2	2	5					
Leu	-6	-3	-2	-3	-2	-4	-3	-4	-3	-2	-2	-3	-3	4	2	6				
Val	-2	-1	0	-1	0	-1	-2	-2	-2	-2	-2	-2	-2	2	4	2	4			
Phe	-4	-3	-3	-5	-5	-5	-4	-6	-5	-5	-2	-4	-5	0	1	2	-1	9		
Tyr	0	-3	-3	-5	-3	-5	-2	-4	-4	-4	0	-4	-4	-2	-1	-1	-2	7	10	
Trp	-8	-2	-5	-6	-6	-7	-4	-7	-7	-5	-3	2	-3	-4	-5	-2	-6	0	0	17
	Cys	**Ser**	**Thr**	**Pro**	**Ala**	**Gly**	**Asn**	**Asp**	**Glu**	**Gln**	**His**	**Arg**	**Lys**	**Met**	**Ile**	**Leu**	**Val**	**Phe**	**Tyr**	**Trp**

aproximadas comparando com mudanças pequenas que ocorrem sobre escalas de tempo curtas. A série de matrizes BLOSUM (*BLOck SUbstitution Matrix*) resolve este problema (HENIKOFF; HENIKOFF, 1992).

A sigla BLOSUM vem do inglês *BLOck SUbstitution Matrix*. Um bloco é uma região de um alinhamento local de múltiplas sequências. Os blocos podem ser obtidos a partir de grupos de sequências de proteínas relacionadas entre elas. Cada bloco representa as regiões melhor conservadas dentro de uma família de proteínas (RUBIN; PIETROKOVSKI, 2003). As matrizes BLOSUM estão baseadas nas mudanças dentro de cada bloco. As sequências de cada bloco são agrupadas colocando duas sequências no mesmo grupo se a sua porcentagem de identidade é maior que um valor L%. Por exemplo, a matriz BLOSUM50 é obtida a partir de um alinhamento de sequências com pelo menos 50% de identidade, isto é, a matriz BLOSUM50. Cada valor (i, j) da matriz representa a pontuação por substituir um aminoácido A_i por um A_j.

A matriz BLOSUM, foi idealizada usando alinhamentos múltiplos de proteínas evolucionariamente divergentes (HENIKOFF; HENIKOFF, 1992). A matriz BLOSUM62, apresentada na Tabela 5, é calculada sobre substituições observadas entre proteínas que compartilham 62% ou menos de identidade. As matrizes BLOSUM com numeração mais alta são indicadas para alinhar sequências proximamente relacionadas e aquelas com numeração mais baixa para sequências mais divergentes.

Tabela 5: Exemplo de uma matriz de substituição BLOSUM62

	Cys	Ser	Thr	Pro	Ala	Gly	Asn	Asp	Glu	Gln	His	Arg	Lys	Met	Ile	Leu	Val	Phe	Tyr	Trp
Cys	9																			
Ser	-1	4																		
Thr	-1	1	5																	
Pro	-3	-1	-1	7																
Ala	0	1	0	-1	4															
Gly	-3	0	-2	-2	0	6														
Asn	-3	1	0	-2	-2	0	6													
Asp	-3	0	-1	-1	-2	-1	1	6												
Glu	-4	0	-1	-1	-1	-2	0	2	5											
Gln	-3	0	-1	-1	-1	-2	0	0	2	5										
His	-3	-1	-2	-2	-2	-2	1	-1	0	0	8									
Arg	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5								
Lys	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5							
Met	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5						
Ile	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4					
Leu	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4				
Val	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4			
Phe	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6		
Tyr	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	
Trp	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11

1.3.3 Comparação entre matrizes de substituição

Cada tipo de matriz de substituição é mais adequada para um tipo de alinhamento. Apesar dessa escolha ser livre, é importante observar as principais diferenças entre as matrizes PAM e as matrizes BLOSUM:

- Matrizes PAM são baseadas sobre um modelo evolucionário explícito (por ex. substituições são contadas sobre uma arvore filogenética), no entanto matrizes BLOSUM estão baseadas sobre um modelo implícito de evolução (MOUNT, 2008a) (MOUNT, 2008b);

- Matrizes PAM estão baseadas sobre mutações obtidas através de um alinhamento global, este inclui tanto regiões altamente conservadas como aquelas altamente mutáveis. As matrizes BLOSUM estão baseadas sobre regiões altamente conservadas em blocos de alinhamentos que não contêm *gaps* (MOUNT, 2008a) (MOUNT, 2008b);

- O método usado para contar as substituições também é diferente. Ao contrario da matriz PAM, o procedimento da BLOSUM usa grupos de sequências dentro do qual nem todas as mutações são consideradas com o mesmo valor de contagem. Matrizes PAM nomeadas com numeração mais alta denotam um esquema de distância evolucionária maior, por enquanto matrizes BLOSUM nomeadas com numeração maior denotam um esquema de similaridade de sequências mais alta e consequentemente uma distancia evolucionaria menor. Como exemplo, a matriz PAM150 é usada para sequências mais distantes

que PAM100; BLOSUM62 é usada para sequências mais proximamente relacionadas que BLOSUM50 (MOUNT, 2008a) (MOUNT, 2008b).

Além de utilizar as matrizes expostas, um algoritmo pode utilizar o próprio método de pontuar as similaridades encontradas. Em geral, para o alinhamento de sequências de bases de DNA, é usual a adoção do critério simples de $+1$ para igualdade e 0 ou -1 para desigualdade, independente das bases envolvidas. Na Seção 1.4 são apresentados os principais algoritmos de busca local e global.

1.4 Principais algoritmos

Existem diversos algoritmos para alinhamento de sequências genéticas, sejam elas de proteínas ou de DNA. Destacam-se entre eles os tradicionais algoritmos de Needleman-Wunsch, que busca um alinhamento global ótimo e é apresentado na Seção 1.4.1; de Smith-Waterman, que busca alinhamentos ótimos locais, e é apresentado na Seção 1.4.2. Além desses, há um grande destaque pelos algoritmos baseados em heurística, que apresentam um tempo de resposta bem melhor, apesar de não ter o compromisso de encontrar o alinhamento ótimo. Neste grupo, destacam-se o FASTA, apresentado na Seção 1.4.3; e o algoritmo BLAST, abordado no Capítulo 2, para o qual é proposta uma arquitetura de *hardware*.

1.4.1 Algoritmo de Needleman - Wunsch

O algoritmo de Needleman-Wunsch (NEEDLMAN; WUNSH, 1970), apresentado em pseudocódigo no Algoritimo 3, obtém o alinhamento global ótimo entre as sequências s e t, permitindo que espaços sejam inseridos para melhorar o alinhamento. Esse algoritmo calcula uma matriz H onde o elemento $H_{i,j}$ representa o escore do melhor alinhamento entre os prefixos s_i, para i incrementando de $0 \rightarrow m$; e t_j, para j variando de $0 \rightarrow n$. Para computar essa matriz utiliza-se a técnica de programação dinâmica, em que soluções maiores são obtidas através de soluções menores, de forma iterativa. Como primeiro passo, inicia-se o elemento $H_{0,0} = 0$, representando o escore entre duas sequências vazias. Em seguida, a primeira linha e a primeira coluna são iniciadas com os valores $H_{i,0} = -i \times gap$ e $H_{0,j} = -j \times gap$.

A fórmula de recorrência baseia-se no fato de que o valor de $H_{i,j}$ de um alinhamento terminado nas posições s_i e t_j é o maior dos escores obtidos, para uma das três opções possíveis: (1) escore do alinhamento entre s_{i-1} e t_{j-1} já calculado, acrescido do escore do alinhamento entre s_i e t_j; (2) escore do alinhamento entre s_{i-1} e t_{j-1} já calculado, acrescido da penalidade por inserir um espaço em s; (3) escore do alinhamento entre s_{i-1} e t_{j-1} já calculado, acrescido

da penalidade por inserir um espaço em t. Desta forma, a recorrência é descrita por $H_{i,j} = max(H_{i-1,j-1} + sbt(s_i, t_j), H_{i-1,j} - gap, H_{i,j-1} - gap)$, onde gap é a pontuação por inserção de um espaço, e $sbt(s_i, t_j)$ é a pontuação obtida na matriz de pontuação para a comparação das bases s_i e t_j (NEEDLMAN; WUNSH, 1970).

A equação é aplicada para preencher a matriz H, do canto superior esquerdo para o canto inferior direito (fase 1). Para encontrar o alinhamento, mantém-se um ponteiro em cada célula indicando qual célula vizinha originou este valor. Assim, percorre-se a sequência de ponteiros da matriz no sentido reverso, iniciando no canto inferior direito até chegar no canto superior esquerdo, num processo denominado de *traceback* (fase 2). Em alguns casos, o *traceback* poderá percorrer mais de um possível caminho na sequência de ponteiros, o que permite a geração de mais de um alinhamento global ótimo. O processo de *traceback* inicia-se na célula (i, j), sendo $i = m$ e $j = n$, e a escolha do caminho a ser seguido faz-se escolhendo qual célula contém o maior valor, no caso de (i, j), avalia-se entre a célula diagonal $(i-1, j-1)$, a célula a esquerda $(i-1, j)$ e a célula acima $(i, j-1)$. Se o maior valor for a célula diagonal, significa que o alinhamento é entre as bases das posições i e j; se a maior valor for da célula a esquerda o alinhamento é entre a base da posição i e um espaço; e por fim se o maior valor for o da célula acima, o alinhamento é entre a base da posição j e um espaço.

A Figura 1 apresenta um exemplo da matriz de programação dinâmica para um alinhamento global ótimo. Nela vemos a matriz de programação gerada pelo alinhamento global entre as sequências s = AGTTCCGGAGG e t = ACTTCCAGA, para os parametros $+1$ para igualdade, -2 para desigualdade e -5 para a inserção de um espaço. O algoritmo descrito necessita preencher toda a matriz de similaridade (fase 1), que contém $(m+1)(n+1)$ elementos. O *traceback* é proporcional ao tamanho do alinhamento, que possui um limite superior da ordem de $O(m+n)$. Considerando que o tamanho das duas sequências são próximos, o algoritmo completo possui uma complexidade de $O(m^2)$ tanto em tempo de processamento como em uso de memória. A complexidade inviabiliza o uso desse algoritmo para sequências muito grandes. Por exemplo, uma matriz de similaridade entre duas sequências de 1 milhão de pares de bases demandaria um uso de memória de aproximadamente 4 Tera bytes, caso cada célula ocupasse 4 bytes de memória.

1.4.2 Algoritmo de Smith - Waterman

Uma situação muito comum na Bioinformática é buscar o melhor alinhamento entre subsequências de s e t. Esse melhor alinhamento é chamado de alinhamento local ótimo e é encontrado

H (i,j)		A	C	T	T	C	C	A	G	A
	0 0	-5 -5	-5 -10	-5 -15	-5 -20	-5 -25	-5 -30	-5 -35	-5 -40	-5 -45
A	-5 -5	1 1	-2 -4	-2 -9	-2 -14	-2 -19	-2 -24	1 -29	-2 -34	1 -39
G	-5 -10	-2 -4	-2 -1	-2 -6	-2 -11	-2 -16	-2 -21	-2 -26	1 -28	-2 -33
T	-5 -15	-2 -9	-2 -6	1 0	1 -5	-2 -10	-2 -15	-2 -20	-2 -25	-2 -30
T	-5 -20	-2 -14	-2 -11	1 -5	1 1	-2 -4	-2 -9	-2 -14	-2 -19	-2 -24
C	-5 -25	-2 -19	1 -13	-2 -10	-2 -4	1 2	1 -3	-2 -8	-2 -13	-2 -18
C	-5 -30	-2 -24	1 -18	-2 -15	-2 -9	-2 -3	1 3	-2 -2	-2 -7	-2 -12
G	-5 -35	-2 -29	-2 -23	-2 -20	-2 -14	-2 -8	-2 -2	-2 1	1 -1	-2 -6
G	-5 -40	-2 -34	-2 -28	-2 -25	-2 -19	-2 -13	-2 -7	-2 -4	1 2	-2 -3
A	-5 -45	1 -39	-2 -33	-2 -30	-2 -24	-2 -18	-2 -12	-2 -6	-2 -3	1 3
G	-5 -50	-2 -44	-2 -38	-2 -35	-2 -29	-2 -23	-2 -17	-2 -11	1 -5	-2 -2
G	-5 -55	-2 -49	-2 -43	-2 -40	-2 -34	-2 -28	-2 -22	-2 -16	1 -10	-2 -7

Figura 1: Preenchimento da matriz e *traceback* para Needleman-Wunsch

através do algoritmo de Smith-Waterman (SMITH; WATERMAN, 1981).

O processo é bastante parecido com o do algoritmo de Needleman-Wunsch, havendo duas diferenças. A primeira ocorre na equação de recorrência, em que se utiliza o valor zero para impedir que números negativos apareçam na matriz de similaridade. A ocorrência de um valor zero representa o começo de um novo alinhamento, pois se este até um determinado ponto for negativo, então é mais vantajoso começar um novo alinhamento naquela posição. Como consequência, a primeira linha e a primeira coluna devem ser preenchidas com zeros. A nova equação de recorrência será $H_{i,j} = max(H_{i-1,j-1} + sbt(s_i, t_j), H_{i-1,j} - gap, H_{i,j-1} - gap, 0)$, onde gap é a pontuação por inserção de um espaço, e $sbt(s_i, t_j)$ é a pontuação obtida na matriz de pontuação para a comparação das bases s_i e t_j (SMITH; WATERMAN, 1981).

A segunda mudança em relação ao algoritmo de Needleman-Wunsch ocorre na forma de se realizar o *traceback*. Em vez de iniciar o *traceback* na posição final $H_{m,n}$, o alinhamento inicia na posição $H_{i,j}$ onde ocorre o maior valor da matriz. O percorrimento é feito até encontrar uma célula com valor zero. Dessa forma, obtém-se o melhor alinhamento desconsiderando prefixos e sufixos pouco significativos. O algoritmo é apresentado em pseudocódigo no Algoritmo 4.

Algoritmo 3 Algoritmo Needlman-Wunsch com traceback

1: **Entrada:** sequências s e t
2: **Saída:** Alinhamento Global Ótimo e escore α
3: **Inicialização:**
4: **Para** $i := 0 \rightarrow m$ **Faça**
5: $H(i, 0) := -i \times gap$
6: **Fim Para**
7: **Para** $j := 0 \rightarrow n$ **Faça**
8: $H(0, j) := -j \times gap$
9: **Fim Para**
10: **Para** $i := 0 \rightarrow m$ **Faça**
11: **Para** $j := 0 \rightarrow n$ **Faça**
12: $H_{i,j} = max(H_{i-1,j-1} + sbt(s_i, t_j), H_{i-1,j} - gap, H_{i,j-1} - gap)$
13: **Fim Para**
14: **Fim Para**
15: **Traceback:**
16: $(i, j) := (m, n)$
17: $\alpha \leftarrow H(m, n)$
18: **Repita**
19: $T_{i,j} \leftarrow \max(H_{i-1,j-1}, H_{i-1,j}, H_{i,j-1})$
20: **Se** $T_{i,j} = H_{i-1,j-1}$ **Então**
21: $(i, j) \leftarrow (i - 1, j - 1)$
22: **Senão Se** $T_{i,j} = H_{i-1,j}$ **Então**
23: $(i, j) \leftarrow (i - 1, j)$
24: **Senão Se** $T_{i,j} = H_{i,j-1}$ **Então**
25: $(i, j) \leftarrow (i, j - 1)$
26: **Fim Se**
27: **Até que** $(i, j) = (0, 0)$

Na Figura 2 há um exemplo da matriz de programação dinâmica para o alinhamento local. Nela vemos a matriz de programação gerada pelo alinhamento local entre as sequências $s =$ AGTTCCGGAGG e $t =$ ACTTCCAGA, para os parametros $+1$ para igualdade, -2 para desigualdade e -5 para a inserção de um espaço.

O processo inicia-se na célula de maior valor e a escolha do caminho a ser seguido faz-se a partir dela, de forma análoga ao algoritmo de Needleman-Wunsch, escolhendo qual célula contém o maior valor, no caso de (i, j), avalia-se entre a célula diagonal $(i - 1, j - 1)$, a célula a esquerda $(i - 1, j)$ e a célula acima $(i, j - 1)$. Se o maior valor for a célula diagonal, significa que o alinhamento é entre as bases das posições i e j; se a maior valor for da célula a esquerda o alinhamento é entre a base da posição i e um espaço; e por fim se o maior valor for o da célula acima, o alinhamento é entre a base da posição j e um espaço. O *traceback* termina quando é encontrado um valor 0 em uma das células (SMITH; WATERMAN, 1981) (NEEDLMAN; WUNSH, 1970).

$H_{(i,j)}$		A	C	T	T	C	C	A	G	A
	0 0	-5 -5	-5 -10	-5 -15	-5 -20	-5 -25	-5 -30	-5 -35	-5 -40	-5 -45
A	-5 -5	2 2	-1 0	-1 0	-1 0	-1 0	-1 0	2 0	-1 0	2 0
G	-5 -10	-1 0	-1 1	-1 0	-1 0	-1 0	-1 0	-1 0	2 2	-1 0
T	-5 -15	-1 0	-1 0	2 3	2 2	-1 0	-1 0	-1 0	-1 0	-1 1
T	-5 -20	-1 0	-1 0	2 2	2 5	-1 1	-1 0	-1 0	-1 0	-1 0
C	-5 -25	-1 0	2 2	-1 0	-1 1	2 7	2 3	-1 0	-1 0	-1 0
C	-5 -30	-1 0	2 2	-1 1	-1 0	2 3	2 9	-1 4	-1 0	-1 0
G	-5 -35	-1 0	-1 0	-1 1	-1 0	-1 0	-1 4	-1 8	2 6	-1 1
G	-5 -40	-1 0	-1 0	-1 0	-1 0	-1 0	-1 0	-1 3	2 10	-1 5
A	-5 -45	2 0	-1 0	-1 0	-1 0	-1 0	-1 0	2 2	-1 5	2 12
G	-5 -50	-1 0	-1 0	-1 0	-1 0	-1 0	-1 0	-1 0	2 4	-1 7
G	-5 -55	-1 0	-1 0	-1 0	-1 0	-1 0	-1 0	-1 0	2 2	-1 3

Figura 2: Preenchimento da matriz e *traceback* para Smith-Waterman

Embora os métodos acima sejam muito eficientes, tanto o algoritmo de Smith-Waterman como o de Needlman-Wunsch têm como deficiência o fato de serem extremamente dispendiosos em termos de complexidade, o que implica diretamente nos recursos computacionais necessários para se realizar o processo de preenchimento das matrizes e principalmente o *traceback*. Como alternativa a esses métodos, existem os métodos estatísticos, que devido a sua complexidade não são de fácil utilização, e portanto são preteridos com relação a outros métodos (PEARSON, 1995).

Para ambos os alinhamentos descritos anteriormente, os algoritmos definidos são de tempo computacional proibitivo e exigem o uso de equipamento especial com grande poder de processamento ou recursos de processamento paralelo. Esta foi uma das restrições que impulsionou a busca por novos algoritmos que preservassem a sensibilidade e seletividade do alinhamento, mas que realizassem o trabalho em tempo computacional razoável (PEARSON, 1991) (PEARSON, 1995).

Como alternativa para essa restrição, foram propostos algoritmos que através dos métodos heurísticos, se utilizam de aproximações, fazendo alinhamentos de sequências curtas, para

Algoritmo 4 Algoritmo Smith-Waterman com traceback

1: **Entrada:** sequências s e t
2: **Saída:** Alinhamento Local Ótimo e escore α
3: **Inicialização:**
4: **Para** $i := 0 \rightarrow m$ **Faça**
5: $H(i, 0) \leftarrow 0$
6: **Fim Para**
7: **Para** $j := 0 \rightarrow n$ **Faça**
8: $H(0, j) \leftarrow 0$
9: **Fim Para**
10: **Para** $i := 0 \rightarrow m$ **Faça**
11: **Para** $j := 0 \rightarrow n$ **Faça**
12: $H_{i,j} \leftarrow \max = (0, H_{i-1,j-1} \ + \ sbt(s_i, t_j), H_{i-1,j} \ - \ gap, H_{i,j-1} \ - \ gap)$
13: **Fim Para**
14: **Fim Para**
15: **Traceback:**
16: $(i, j) := \arg\max \{H(i, j) | i = 1, 2, \ldots, n \ , j = 1, 2, \ldots, m$
17: $\alpha \leftarrow H(i, j)$
18: **Repita**
19: $T_{i,j} \leftarrow \max = (H_{i-1,j-1}, H_{i-1,j}, H_{i,j-1})$
20: **Se** $T_{i,j} = H_{i-1,j-1}$ **Então**
21: $(i, j) \leftarrow (i - 1, j - 1)$
22: **Senão Se** $T_{i,j} = H_{i-1,j}$ **Então**
23: $(i, j) \leftarrow (i - 1, j)$
24: **Senão Se** $T_{i,j} = H_{i,j-1}$ **Então**
25: $(i, j) \leftarrow (i, j - 1)$
26: **Fim Se**
27: **Até que** $H_{i-1,j-1} = (0)$

somente depois realizar os alinhamentos a partir dessas regiões pré-alinhadas, minimizassem o espectro de busca. A ideia destes métodos é que, se duas sequências se parecem, então elas terão muitas janelas em comum. O FASTA utiliza-se parte dessa definição e é um caso de sucesso entre os algoritmos baseados em heurística. Na Seção 1.4.3, apresenta-se a metodologia do algoritmo, que é de grande importância para o entendimento do outro, e principal algoritmo de alinhamento de sequências, que é fruto dessa dissertação: o algoritmo BLAST, apresentado em detalhes no Capítulo 2.

1.4.3 FASTA

O algoritmo FASTA apresenta um método para busca de alinhamentos locais empregando uma matriz de substituição. A ideia central é a busca de casamentos de subsequências pequenas, de tamanho arbitrário k, chamadas *k-tuplas*, que são sequências ordenadas de k bases. Identificar os pequenos alinhamentos serve como indicativo para a possibilidade de alinhamentos maiores (PEARSON, 1990) (RUBIN; PIETROKOVSKI, 2003).

A estratégia de criar palavras menores a partir de pedaços das sequências originais também é usada pelo BLAST, e subdivide o espaço de busca, simplificando o processo de obtenção dos resultados. No Capítulo sobre o algoritmo BLAST, apresenta-se em detalhes como essa mudança ajuda no tempo de execução e quais os seus impactos. De forma geral, para minimizar o esforço empreendido, o algoritmo identifica as regiões de maior incidências das *k-tuplas* e concentra o esforço nestas regiões. Por meio de um procedimento heurístico, atribuem-se valores a estas regiões de alta incidência. Como a análise é otimizada, alguns alinhamentos podem não serem percebidos pelo algoritmo. Assim, uma busca com uma faixa de k variantes pode aumentar a sensibilidade do resultado. Em algumas abordagens, o FASTA é utilizado como precursor no alinhamento. As sequências de melhor alinhamento são revisitadas utilizando-se outros métodos, aumentando-se assim a confiança e sensibilidade dos resultados.

Além de fornecer os escores e segmentos, FASTA calcula também uma estimativa de significância estatística dos segmentos encontrados, esse valor é conhecido como *i probabilidade* p, de que o alinhamento tenha sido obtido por coincidência. Esse valor varia de 0 a 1, e no limite 0 indica maior confiança, no limite 1 indica um resultado espúrio (PEARSON, 1990).

O procedimento a seguir resume o a estratégia e o funcionamento do FASTA, quando se tem duas sequências s e t a serem comparadas:

- Passo 1: Dadas as sequências s e t, nesta etapa o algoritmo busca regiões que possuam identidades, com no mínimo k bases consecutivas entre as sequências comparadas. Através de um dos métodos de alinhamento simples, matriz de pontos por exemplo, calcula-se pontuações para estas regiões e retorna as 10 regiões cujas pontuações são maiores. Nesse passo, a pontuação é calculada apenas através da posição de cada base nas sequências s e t. Se na posição s_1 temos a base A, e na posição t_4 temos a base A, a pontuação desse alinhamento é 3, e ele só será levado em consideração se o parâmetro $k = 1$. Essa pontuação baseada em posição, é semelhante a realizada pelo algoritmo de Smith-Waterman, que só utiliza $k = 1$, como é visto nas Tabelas 6, 7 e 8;

- Passo 2: Nesta etapa, o algoritmo submete as 10 regiões com mais alta densidade à comparação utilizando uma matriz de pontuação, usualmente PAM250, e determina a pontuação do alinhamento sem a inserção de espaços. Estas regiões são alinhamentos parciais sem espaços. A pontuação melhor é chamada de $init1$;

- Passo 3: Descobre-se quais regiões iniciais vindas do passo 2 podem ser agrupadas, através da análise de suas extremidades. Calcula-se o alinhamento ótimo destas regiões agrupa-

das, levando em consideração os espaços que foram inseridos. A pontuação resultante deste alinhamento é chamada de *initn*;

- Passo 4: Classifica-se as sequências do banco de dados. O FASTA faz isso levando em consideração *initn*. Calcula-se a média e o desvio padrão de *init*1 e *initn* e apresenta um histograma. Calcula-se o alinhamento ótimo local da sequências que obtiveram melhor classificação através do algoritmo de Smith-Waterman. Esta comparação final considera todas as possibilidades que estão dentro de uma certa banda centralizada ao redor da região inicial com maior pontuação encontrada no passo 2.

Tabela 6: FASTA: Matriz de pontos para $k = 1$

	A	C	G	T	A	C	G	T	A	C
G			1				1			
G			1				1			
T				1				1		
T				1				1		
A	1				1				1	
C		1				1				1
G			1				1			
G			1				1			
T				1				1		
C		1				1				1

Tabela 7: FASTA: Matriz de pontos para $k = 2$

	A	C	G	T	A	C	G	T	A	C
G										
G			1				1			
T										
T				1				1		
A	1				1				1	
C		1				1				
G										
G			1				1			
T				1						
C						1				

Tabela 8: FASTA: Matriz de pontos para $k = 3$

	A	C	G	T	A	C	G	T	A	C
G										
G										
T										
T				1				1		
A	1				1					
C										
G										
G										
T										
C										

1.5 Considerações Finais do Capítulo

Este Capítulo apresentou os princípios do alinhamento de sequências biológicas, seus conceitos fundamentais, o alfabeto e outras definições. Também apresentou-se os primeiros algoritmos para comparação exata e inexata de sequências, os métodos de alinhamento global e local, e os algoritmos de Needlman-Wunsch e Smith-Waterman, que são a base de estudo no campo de alinhamento de sequências.

O Capítulo 2 irá expandir as vantagens dos métodos heurísticos frente aos demais, e apresentar e detalhar o algoritmo BLAST, que é o escolhido para a implementação de uma arquitetura de *hardware*.

Capítulo 2

ALGORITMO BLAST

O PRINCIPAL desafio de um algoritmo de alinhamento é encontrar a posição onde duas ou mais sequências melhor se relacionam, quando comparadas entre si. Para tal, os métodos de alinhamento necessitam de uma maneira de classificar as posições, e mensurar se um alinhamento é adequado ou não. A forma que os algoritmos de sequenciamento têm para avaliar se uma posição é boa ou não, é atribuir uma pontuação para cada comparação feita. Essa avaliação é feita durante o processo de comparação, onde um trecho recebe um escore de acordo com uma matriz de pontuação previamente escolhida (OEHMEN; NIEPLOCHA, 2006).

É intuitivo esperar que um algoritmo ideal para o alinhamento de sequências seja capaz de testar todas as combinações possíveis e pontua-las, de forma iterativa, apresentando em sua saída os melhores alinhamentos entre as sequências analisadas. Um bom algoritmo de alinhamento deve encontrar nas sequências submetidas para comparação quais serão as posições relativas capazes de gerar maior pontuação, e consequentemente, maior similaridade. Todavia, por limitações de *hardware*, *software*, tempo de execução, ou até mesmo por característica do método de comparação escolhido, nem sempre é possível testar todas as possibilidades de comparação entre um par ou grupo sequências, pois a realização da permutação de todas as possibilidades depara com uma das limitações supracitadas.

Na literatura, existe uma clara segmentação entre os algorítimos de alinhamento de sequências. Os algoritmos clássicos, que testam todas as combinações possíveis, são usualmente mais lentos enquanto os algoritmos recentes, utilizam de técnicas estatísticas, probabilísticas e heurísticas para diminuir o número de testes visando emitir uma resposta mais ágil.

No Capítulo anterior, apresentou-se os algoritmos de Needlman-Wunsch e Smith-Waterman, que são a base de estudo no campo de alinhamento de sequências e se enquadram no grupo dos algoritmos tradicionais. Dentre os algoritmos baseados em heurística, foi descrito o FASTA, que traz em sua estratégia algumas peculiaridades também utilizadas pelo algoritmo BLAST.

Tanto o BLAST como o FASTA são os métodos baseados em heurística, altamente difundidos na comunidade científica, e limitam-se a testar um grupo restrito de possibilidades, com critérios bem definidos, melhorando consequentemente o tempo de resposta, em detrimento à sensibilidade da comparação (KORF; YANDELL; BEDELL, 2003).

Neste ponto, aparece um conceito importante dos algoritmos de alinhamento de sequências: a sensibilidade. Entende-se por sensibilidade de um algoritmo de alinhamento a capacidade que ele tem de encontrar resultados compostos por pequenos trechos de sequência. Para uma comparação entre duas sequências numéricas, os algoritmos menos sensíveis necessitam que a parcela de igualdade entre as sequências seja consideravelmente maior, para que esse trecho seja entendido como solução. Um algoritmo mais sensível, já entende como resposta, uma posição com a parcela de igualdade menor. Por exemplo, dadas duas sequências hipotéticas s=AAA e t=AAG, um algoritmo com sensibilidade de no mínimo 3 bases não encontraria alinhamento entre s e t, enquanto um algoritmo com sensibilidade de no mínimo 2 bases encontraria o alinhamento entre s e t.

Para um algoritmo de sequenciamento que busca a igualdade entre as sequências, a sensibilidade pode parecer um conceito irrelevante, porém, a maioria dos algoritmos heurísticos tem como meta a busca de uma solução razoável, e não a solução ideal de um problema de sequenciamento. Uma sensibilidade muito baixa, pode descartar a solução de alinhamento que alcançaria maior pontuação. Vale ressaltar que no mundo biológico, uma das principais aplicações do alinhamento de sequências é identificar a relação parental entre organismos, representados por sequências de proteínas ou DNA, e que essas sequências podem evoluir ou sofrer uma mutação, que vá alterar, por exemplo, um ou mais caracteres da sequência em relação à original. Com isso, ao escolher um algoritmo de baixa sensibilidade, pode-se descartar uma desigualdade, que pode ser exatamente a posição ideal para o melhor alinhamento dentre as possibilidades. Portanto, um bom algoritmo de alinhamento de sequências deve apresentar uma boa sensibilidade, e é ideal que esse parâmetro possa ser ajustado.

A sensibilidade tem uma relação estreita com o tempo de resposta dos algoritmos. O aumento da sensibilidade acarreta num aumento do número de combinações possíveis, e consequentemente num aumento do processamento necessário para encontrar a resposta. Isso significa dizer, que ao escolher um método mais rápido, porém menos sensível, irá reduzir o número de combinações a serem testadas, e poderá deixar de testar justamente a condição que traduzir-se-ia no melhor alinhamento ou pontuação. A sensibilidade do algoritmo caracteriza-se como um importante parâmetro para o processo de comparação. Afim de aumentar a sensi-

bilidade dos algoritmos, há a possibilidade de utilizar algoritmos em sequência, executando-o em vários estágios. Esse principio é utilizado pelo FASTA, que examina o espaço de busca e depois, com uma sensibilidade maior, analisa os dez melhores resultados.

Além do problema da comparação, para realizar um bom alinhamento, é necessário encontrar a máxima pontuação dos segmentos analisados, num problema que remete à análise de valor máximo. Os métodos baseados em heurística tem conhecida eficiência nessa análise, e são portanto mais eficientes que os algoritmos tradicionais na busca do melhor resultado, pois os algoritmos tradicionais trabalham com a construção de matrizes e *traceback* baseado em programação dinâmica.

O estudo e a combinação das técnicas de grupos diferentes de algoritmos teve sempre como objetivo a busca de um algoritmo que alie sensibilidade e tempo de resposta. A partir da criação do algoritmo FASTA, conseguiu-se também acrescentar à pesquisa o caráter intuitivo trazido pelos algoritmos genéticos, o que levou ao desenvolvimento do algorítimo BLAST (*Basic Local Alignment Search Tool*), que utiliza algumas etapas do FASTA em adição a algumas funções próprias que são apresentadas na Seção 2.1.

2.1 Estratégia do algoritmo do BLAST

Dadas duas sequências s e t, o objetivo do algoritmo BLAST é alinhar s e t, sem a inserção de buracos, utilizando uma matriz de pontuação (BLOSUM ou PAM), para o caso de sequências de proteínas (ALTSCHUL et al., 1990). Para alinhamento de sequências de DNA, o critério de pontuação pode ser outro, atribuindo-se um valor positivo para a igualdade e um valor negativo para a não-igualdade, por exemplo.

O algoritmo parte do princípio que, por exemplo, para que duas sequências tenham alto grau de similaridade, elas devem ter vários pequenos trechos idênticos. Para as duas sequências s e t dadas, define-se por *segmento* uma simples subdivisão de s ou t em subsequências de tamanhos menores. Um *par de segmento* (s, t) ou *hit* consiste em dois segmentos, um em s e um em t, de mesmo tamanho. A alusão à linguagem dos algoritmos genéticos permite que o *hit* seja chamado de semente. Quando uma sequência é subdividida em segmentos de tamanho w, esse segmentos de tamanho w são chamados de *palavra*. A pontuação obtida pelo alinhamento do par de segmento (s, t) é representada por $\alpha(s, t)$. Denomina-se *par de segmentos localmente máximo*, representado pela sigla *LMSP* quando algum par de segmentos (s, t) atinge uma pontuação que não pode mais ser otimizada, seja aumentando ou diminuindo o par de segmento; e um *Par de segmento máximo*, representado pela sigla *MSP* , é qualquer

par de segmento (s, t) cujo alinhamento atinge a máxima pontuação $\alpha(s, t)$ (ALTSCHUL et al., 1990) (KORF; YANDELL; BEDELL, 2003).

Além das definições *LMSP* e *MSP*, é determinado ainda, um valor mínimo para as pontuações. Esse valor é a pontuação de corte S. Um par de segmento (s, t) é chamado de *HSP* , se ele é localmente máximo e se $\alpha(s, t) \geq S$. Em termos práticos, a meta do algoritmo BLAST é, portanto, computar todos os *HSPs* (ALTSCHUL et al., 1990).

O procedimento a seguir resume a estratégia e o funcionamento do BLAST, quando se tem duas sequências de proteínas s e t a serem comparadas (ALTSCHUL et al., 1997) (ALTSCHUL et al., 1990) (KORF; YANDELL; BEDELL, 2003):

- Pré-Processamento: A sequência t é dividida em palavras de tamanho w. Para cada palavra gerada, é elaborada uma lista com todas as *w-tuplas* de tamanho w. Uma *w-tupla* é a geração de uma nova palavra de tamanho w a partir da permutação da palavra original sobre o alfabeto $A_{Proteinas}$. Para cada palavra gerada, é calculada uma pontuação, e essa palavra irá compor a lista de *w-tuplas* se, e somente se, a pontuação atingida for superior ao limite mínimo (do inglês *threshold*, representado por T). Na Tabela 9 há um exemplo desse preenchimento, também chamado de cálculo da vizinhança;

- Passo 1: Semeadura. É a etapa de localização dos *hits*. Nessa etapa, pesquisa-se em s por todos os *w-tuplas* da lista pré-processada. Armazena-se a posição onde todos os *hits* são encontrados;

- Passo 2: Extensão. Nessa etapa, cada semente (s, t) (*hit*), encontrada na etapa de semeadura é revisitada com o objetivo aumentar a quantidade de bases alinhadas. Cada semente estendida em ambas as direções até que o escore $\alpha(s, t)$ não possa mais ser melhorado, gerando assim um *LMSP*. As extensões que superarem o escore S são reportadas pelo algoritmo como *HSP*, e possíveis soluções de alinhamento;

- Passo 3: Avaliação. Nessa etapa é feita uma avaliação estatística sobre as respostas encontradas pelo algoritmo. Para algumas versões, a etapa de avaliação consiste apenas em comparar quais respostas são maiores e menores que o escore S. Para outras versões, a etapa de avaliação determina o *E-Value*, representado por E, que determina a quantidade de *HSPs* que pode ter ocorrido aleatoriamente. Essa avaliação é feita através da fórmula $E = K \times n \times m \times e^{y \times S}$, onde K e y são variáveis dependentes do modelo, m e n são os tamanhos da sequências e S o limite superior.

No procedimento descrito, o grande gargalo para a execução do algorítimo é a fase de pré-processamento. Na Tabela 9 observa-se que para um sequência pequena, $n = 7$ e $w = 2$, existe número considerável de palavras geradas. Isto infere que sequências de tamanho maiores podem levar a uma explosão combinacional. A forma usual de minimizar o problema do pré-processamento é através da construção de uma tabela de pesquisa, *lookup table*.

O que justifica a criação dessa tabela de pesquisa é a possibilidade de se buscar uma sequência em um banco de dados, no que é determinado alinhamento múltiplo de sequências. Dessa forma, a fase de pré-processamento, otimiza e evita o processo repetitivo de operação nas sequências relacionadas.

Tabela 9: Exemplo de pré-processamento para t = RQCSAGW, e a lista de palavras de $w = 2$ com escore $T > 8$ utilizando a matriz BLOSUM62

Palavra	*pares* com escore $T > 8$
RQ	RQ
QC	QC, RC, EC, NC, DC, KC, MC, SC
CS	CS, CA, CN, CD, CQ, CE, CG, CK, CT
SA	-
AG	AG
GW	GW, AW, RW, NW, DW, QW, EW, HW, KW, PW, SW, TW, WW

Após entender o pré-processamento, é possível vislumbrar o impacto do tamanho das sequências, definidos por m e n; das palavras de tamanho w e dos parâmetros escolhidos T e S no tempo de execução do algoritmo. Na Seção 2.2 é apresentado o detalhamento e o significado de cada parâmetro nas etapas do BLAST. Embora os parâmetros sejam universais, eles impactam de forma diferente em cada versão do BLAST. Isso ocorre pois a versão destinada ao alinhamento de proteínas apresentada diferenças se comparada a versão para o alinhamento de sequências de DNA.

Nessa Dissertação, o objetivo é propor uma arquitetura de *hardware* para o alinhamento sequências de DNA. O algoritmo BLAST é simplificado para as sequências de DNA, pois não há etapa de pré-processamento, e a fase de semeadura consiste em listar apenas as palavras de tamanho w idênticas encontradas tanto em s como em t. Isso implica dizer que a pontuação mínima T para que um trecho seja considerado semente é $T = w \times I$ sendo I o valor atribuído quando se encontra uma similaridade.

Na Seção 2.2 são apresentados os detalhes de como cada etapa funciona para o alinhamento de sequências de DNA, e a importância de cada parâmetro.

2.2 Parâmetros e Etapas do BLAST

O algoritmo BLAST é um método heurístico de busca que procura alinhar sequências genéticas. A metodologia é identificar pequenas palavras coincidentes de um comprimento w que atinjam uma pontuação mínima T, quando submetidas a uma tabela de avaliação predeterminada. Essa é a primeira etapa do algoritmo, onde na versão para sequências de DNA o BLAST divide as sequências s e t em pequenos intervalos sobrepostos de tamanho w, como visto nos algorítimos 5 e 6, e a partir de um critério de pontuação, compara e elimina os trechos que não obtiverem, no mínimo T. As sequências remanescentes, denominadas sementes, compõe o grupo restrito de possibilidades onde o algoritmo tentará maximizar a pontuação.

Nos algoritmos 5 e 6, a sequência s, de tamanho m, é referenciada como sequência alvo. Cada base da sequência s é representada pelo índice que informa a posição desta base na respectiva sequência. Supondo que a sequência alvo s seja s= ATG, a base representada em $s_0 = A$, a base representada em $s_1 = T$ e a base representada em $s_2 = G$. O mesmo ocorre para a sequência t, de tamanho n, referenciada como sequência buscada.

Algoritmo 5 Divisão da sequência alvo s em palavras

Entrada sequência s:$[s_0, s_1, s_2, \ldots, s_i, \ldots, s_{m-1}]$
1: **Para** $i = 0 \rightarrow (m - w)$ **Faça**
2: $\quad alvo \leftarrow [alvo_0, alvo_1, alvo_2, \ldots, alvo_i, \ldots, alvo_{m-w}]$
3: $\quad$ **Para cada** $alvo_i$ **Faça**
4: $\quad\quad alvo_i \leftarrow [s_i, s_{i+1}, s_{i+2}, \ldots, s_{i+w-1}]$
5: $\quad$ **Fim Para**
6: **Fim Para**
7: **Retorna** $alvo$

De forma análoga aos algoritmos genéticos, esse processo corresponde a semeadura, onde o BLAST procura localmente pequenas regiões de similaridade, onde é mais provável que estejam os alinhamentos consideráveis, para iniciar o processo de comparação. O processo de semeadura constitui uma grande vantagem dos métodos heurísticos, pois como poucos alinhamentos atingem a pontuação mínima, denominada *threshold* e representado por T, o espaço de busca, e consequentemente o tempo de resposta, é reduzido drasticamente logo na primeira etapa. Em contrapartida, existe o risco do algoritmo desconsiderar o melhor alinhamento possível. Como descrito, os métodos heurísticos não tem o compromisso de fornecer o melhor alinhamento e sim uma série de bons alinhamentos possíveis.

Para alinhar duas sequências de tamanhos distintos m e n, o BLAST irá dividir a sequência buscada em palavras de tamanho w, algorítimo 6, e comparar essas palavras com a sequência alvo também dividida, como apresentada no algorítimo 5, pontuando as igualdades

Algoritmo 6 Divisão da sequência buscada t em palavras

Entrada Sequência t:$[t_0, t_1, t_2, \ldots, t_j, \ldots, t_{n-1}]$
1: **Para** $j = 0 \rightarrow (n - w)$ **Faça**
2: $palavras \leftarrow [palavra_0, palavra_1, palavra_2, \ldots, palavra_j, \ldots, palavra_{n-w}]$
3: **Para cada** $palavra_j$ **Faça**
4: $palavra_j \leftarrow [t_j, t_{j+1}, t_{j+2}, \ldots, t_{j+w-1}]$
5: **Fim Para**
6: **Fim Para**
7: **Retorna** $palavras$

e desigualdades de acordo com uma tabela de pontuação. Esse processo de comparação é apresentado em pseudo-código no algorítimo 7. Os ajustes nos parâmetros w e T possibilitam aumentar a sensibilidade do algoritmo, tornando-o mais lento com valores de w e T mais baixos ou, diminuir a sensibilidade a sensibilidade do algoritmo, tornando-o mais rápido com valores de w e T mais elevados.

O resultado da etapa de semeadura é uma matriz preenchida com valor da pontuação obtida nas posições onde se encontrou igualdade. Essas posições são as sementes do algoritmo, e serão consideradas como possíveis pontos iniciais de alinhamento. Estes pontos iniciais serão reavaliadas, verificando as bases imediatamente subsequentes, no processo onde esses trechos estendidos para ambos os lados visando aumentar a pontuação do alinhamento proposto. Todos dos demais pares, que não atingiram a pontuação mínima na etapa de semeadura serão desconsiderados. Para o alinhamento de sequências de DNA, usualmente se utiliza o critério de pontuação próprio, em detrimento a utilização da matiz de pontuação. Nessa Dissertação, cada similaridade é pontuada com 1, enquanto a desigualdade é pontuada com 0.

Na etapa denominada *Extensão*, o objetivo é encontrar o melhor alinhamento local possível, isto é, maximizar a pontuação. Partindo de uma sequência semente com pontuação T ou maior, o algoritmo estende o alinhamento em ambos as direções na tentativa de aumentar a pontuação até encontrar uma desigualdade. Quando a encontra, armazena os índices e pontuação dos pares encontrados e inicia a terceira e última etapa, denominada avaliação.

Algoritmo 7 Semeadura

Entrada $alvo$ e $palavras$
1: **Para** $i = 0 \rightarrow (m - 1)$ **Faça**
2: **Para** $j = 0 \rightarrow (n - w)$ **Faça**
3: **Se** $palavra_i = alvo_j$ **Então**
4: $Pos[i, j] \leftarrow 1$
5: **Fim Se**
6: **Fim Para**
7: **Fim Para**
8: **Retorna** Pos

A partir da matriz preenchida gerada na etapa de semeadura, na forma do Algorítimo 7, esta etapa irá realizar a extensão dessas sementes. Se uma par de palavras foram consideradas sementes na etapa anterior, o procedimento é verificar as próximas letras à direita dessas palavras, incrementando uma posição na matriz, e compará-las novamente. Se a nova comparação resultar em igualdade, faremos novo incremento e nova comparação, até que seja encontrada uma desigualdade na comparação. Algumas versões do BLAST tem maior tolerância, e só dão por encerrada a etapa de extensão após várias desigualdades. Existe um parâmetro que determina esse critério de parada, denominado *drop-off*, representado por x, que é calculado somando a pontuação das desigualdades encontradas. Quando essa pontuação atinge x, a etapa é encerrada. Para fins de continuidade do algorítimo, é realizada a substração do valor máximo da pontuação das igualdades por x.

Na avaliação o algoritmo compara os pares de maior pontuação, denominados pelo acrônimo HSP (*High-Scoring Segment Pairs*) , submetendo-os a um parâmetro S, e descartando alinhamentos que não atingirem esse valor (ALTSCHUL et al., 1997). A etapa de avaliação tem o objetivo de detectar a significância biológica que uma determinada similaridade pode ter, pois o parâmetro S reflete o número de alinhamentos esperados, e é baseado também em critérios bem definidos que representam a probabilidade de certas mutações acontecerem, tornando um alinhamento improvável biologicamente seja eliminado. Detalhes sobre o parâmetro S são abordados na Seção 2.2.1.

Comparado a outros métodos heurísticos, o BLAST executa o alinhamento de sequências de DNA e proteínas mais rapidamente que todos os outros algoritmos conhecidos, considerando-se a mesma sensibilidade (ALTSCHUL et al., 1997) (KORF; YANDELL; BEDELL, 2003). A velocidade do BLAST aliada a sua semelhança a um algoritmo genético tradicional, que o tornava intuitivo, alavancou os estudos no algoritmo que se tornou o mais utilizado na comunidade cientifica. Com o aperfeiçoamento e a elaboração de novas funções, os BLAST se tornou uma poderosa ferramenta para pesquisa de informações biológicas em bancos de dados públicos. Todas as versões do BLAST estão disponíveis através do programa que executa o algoritmo na versão *stand alone* ou *online* no repositório mantido pelo NCBI (*National Center for Biotechnology Information*). Nesse repositório também estão disponíveis diversos bancos de dados públicos com informações de sequências genéticas (SHPAER et al., 1996). No Capítulo de resultados, algumas dessas sequências são utilizadas para validar o modelo de arquitetura proposto. Quanto as variações do BLAST, serão apresentadas na Seção 2.2.3.

Algoritmo 8 Extensão para Direita

Entrada Pos, Sequência s:$[s_0, s_1, \ldots, s_j, \ldots, s_{m-1}]$ e Sequência t:$[t_0, t_1, \ldots, t_j, \ldots, t_{n-1}]$

1: **Para** $i = 0 \rightarrow (m-1)$ **Faça**
2: **Para** $j = 0 \rightarrow (n-w)$ **Faça**
3: **Se** $Pos[i,j] = 1$ **Então**
4: $E_A \leftarrow [Concatenar(alvo_i, s_{i+w})]$
5: $E_B \leftarrow [Concatenar(palavra_j, t_{j+w})]$
6: **Enquanto** $(E_A) = (E_B)$ **Faça**
7: $i \leftarrow (i+1)$
8: **Se** $i = (m-1)$ **Então**
9: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
10: $Pos[i,j] \leftarrow Pontos[Comparar(E_A, E_B)]$
11: **Fim Se**
12: **Fim Se**
13: $j \leftarrow (j+1)$
14: **Se** $j = (n-w)$ **Então**
15: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
16: $(Pos[i,j]) \leftarrow Pontos[Comparar(E_A, E_B)]$
17: **Fim Se**
18: **Fim Se**
19: $E_A \leftarrow Concatenar(E_A, s_{i+w+1})$
20: $E_B \leftarrow Concatenar(E_B, t_{j+w+1})$
21: **Fim Enquanto**
22: **Senão**
23: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
24: $Pos[i,j] \leftarrow Pontos[Comparar(E_A, E_B)]$
25: **Fim Se**
26: **Fim Se**
27: **Fim Para**
28: **Fim Para**

2.2.1 Espaço de Busca e Complexidade

A escolha dos parâmetros têm impacto direto no tempo de execução do algoritmo. O algoritmo BLAST é composto por 3 fases: semeadura, extensão e avaliação. Na fase de semeadura o algoritmo calcula a pontuação resultante do alinhamento de cada subsequência, de tamanho w derivada a partir da sequências de entrada, com todas as possíveis $|\Sigma|^w$ sequências de tamanho w, onde Σ é o alfabeto de símbolos considerado, ou seja, as bases nucleotídicas, [A, T, C e G] usadas para os alinhamentos de sequencias de DNA, ou os vinte aminoácidos [G, A, L, M, F, W, K, Q, E, S, P, V, I, C, Y, H, R, N, D, T], utilizadas nos alinhamentos de sequências de proteínas.

A fase de semeadura precisa, para uma sequência de consulta t de tamanho m, varrer todas as $m - w + 1$ subsequências de tamanho w e calcular a pontuação em relação a todas as $|\Sigma|^w$ sequências possíveis no conjunto Σ de bases nucleotídicas ou de aminoácidos. Logo, a

Algoritmo 9 Extensão para Esquerda

Entrada Pos, Sequência s:$[s_0, s_1, \ldots, s_j, \ldots, s_{m-1}]$ e Sequência t:$[t_0, t_1, \ldots, t_j, \ldots, t_{n-1}]$

1: **Para** $i = 0 \rightarrow (m-1)$ **Faça**
2: **Para** $j = 0 \rightarrow (n-w)$ **Faça**
3: **Se** $Pos[i,j] = 1$ **Então**
4: $E_A \leftarrow [Concatenar(s_{i-1}, alvo_i)]$
5: $E_B \leftarrow [Concatenar(t_{j-1}, palavra_j)]$
6: **Enquanto** $(E_A) = (E_B)$ **Faça**
7: $i \leftarrow (i-1)$
8: **Se** $i = 0$ **Então**
9: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
10: $Pos[i,j] \leftarrow Pontos[Comparar(E_A, E_B)]$
11: **Fim Se**
12: **Fim Se**
13: $j \leftarrow (j-1)$
14: **Se** $j = 0$ **Então**
15: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
16: $(Pos[i,j]) \leftarrow Pontos[Comparar(E_A, E_B)]$
17: **Fim Se**
18: **Fim Se**
19: $E_A \leftarrow Concatenar(s_{i-1}, E_A)$
20: $E_B \leftarrow Concatenar(t_{j-1}, E_B)$
21: **Fim Enquanto**
22: **Senão**
23: **Se** $Pontos[Comparar(E_A, E_B)] \geq T$ **Então**
24: $Pos[i,j] \leftarrow Pontos[Comparar(E_A, E_B)]$
25: **Fim Se**
26: **Fim Se**
27: **Fim Para**
28: **Fim Para**

primeira fase do algoritmo ocorre em $O((m-w+1)|\Sigma|^w)$.

Um valor padrão inicial para esses parâmetros são, em geral, $w = 3$ e $|\Sigma| = 20$ para aminoácidos e, $w = 12$ e $|\Sigma| = 4$ para nucleotídios, o que permite que a complexidade pode ser representada por $O(m)$.

Na fase de extensão, cada semente pode iniciar um alinhamento que, no caso extremo, é do tamanho m da sequência s. Portanto, a complexidade desse passo resulta em $O(mn)$. Na etapa final, a avaliação de significância de cada *HSP* pode ser realizada em tempo constante $O(1)$. Assim, o tempo total do algoritmo BLAST é $O(m) + O(mn) + O(1) = O(mn)$, sendo então, para os valores usuais de w, os fatores impactantes para o tempo total do algoritmo o tamanho das sequências de entrada (BRITO, 2003) (SHPAER et al., 1996) (ZAHA; FERREIRA; PASSAGLIA, 2003).

2.2.2 Um Exemplo Ilustrativo

Supondo que a matriz de substituição seja a apresentada na Tabela 10, $w = 2$ e a sequência de consulta t= ACCGTA. Dividindo a sequência t em palavras de tamanho w, tem-se as cinco subsequências (AC, CC, CG, GT, TA) e as sequências possíveis de tamanho w sobre o alfabeto Σ de bases nucleotídicas totalizam $4^2 = 16$. Tomando a subsequência AC com a sequência AG, a pontuação resultante seria $5 - 4 = 1$, enquanto a pontuação resultante entre CC e GA seria $-4 - 4 = -8$. Sempre que uma pontuação é maior ou igual ao parâmetro T, que é ajustado de acordo com a sensibilidade almejada para um tipo de comparação, essa subsequências é usada como semente de alinhamento local na fase de extensão. Uma das estratégias para realizar isso, é construir um autômato finito determinístico que permita localizar *hits* entre as subsequências com elevado escore e a base de sequências conhecidas. Na prática, como exposto, há a construção uma lista pré-processada com as possíveis variações e suas respectivas pontuações, se tornando uma semente, aquelas encontradas nas sequências e atendam ao critério de pontuação T. Na fase de extensão, o algoritmo passa pela sequência alvo s no caso de um alinhamento

Tabela 10: Exemplo de matriz de substituição.

	A	G	C	T
A	5	-4	-4	-4
G	-4	5	-4	-4
C	-4	-4	5	-4
T	-4	-4	-4	5

entre duas sequências ou por cada sequência da base de dados no caso de uma versão de múltiplas sequências, buscando por subsequências, de tamanho w que sejam idênticas a alguma das subsequências da etapa anterior. Quando o autômato encontra uma subsequência idêntica, o algoritmo estende cada *hit* em ambas as direções com o intuito de aumentar sua pontuação. Quando essa extensão encontra caracteres diferentes, porém, a pontuação tende a diminuir, segundo a matriz de substituição.

A extensão de um *hit* termina quando a sua pontuação começa a diminuir, e existe um critério de parada simbolizado pelo parâmetro X que delimita a maior distancia em relação à maior pontuação encontrada em extensões menores desse *hit*. A etapa de extensão armazena a máxima pontuação e continua estendendo em ambos os lados, até que o valor X seja alcançado, fazendo com que o algorítimo retorne ao máximo valor armazenado. Após estender o *hit*, na fase de avaliação o *hit* estendido é avaliado quanto à sua significância estatística. O ajuste no valor de X retorna o quão tolerante será a busca no caso de trechos desiguais. Isto quer dizer

que *hits* espúrios devem ser eliminados do resultado. De forma simples, a significância de um *hit* pode ser avaliada separando os *hits* entre aqueles com pontuação maior do que um certo parâmetro S e aqueles com pontuação menor do que S (ALTSCHUL et al., 1997) (ALTSCHUL et al., 1990) (KORF; YANDELL; BEDELL, 2003).

Os *hits* significativos são chamados *High-Scoring Segment Pairs* (HSP) e, na prática, o valor de S é determinado por uma análise estatística sobre quantos *hits* espúrios são esperados. De todos os pares HSP, aquele com pontuação mais alta é chamado MSP (*Maximal-scoring Segment Pair*) . A saída do algoritmo informa as sequências que obtiveram a pontuação mínima necessária, S (ALTSCHUL et al., 1997) (ALTSCHUL et al., 1990) (KORF; YANDELL; BEDELL, 2003).

2.2.3 Versões do algoritmo BLAST

Desde sua invenção, o algorítimo BLAST apresentou algumas variações de acordo com o tipo de abordagem. Essas variações, diferem entre si no tipo de sequência analisada, proteínas ou DNA; ou se o algorítimo permite ou não a inserção de buracos, funcionalidade importante na identificação de relações parentais e de mutação entre as sequências.

Por se tratar de um sistema aberto, e com diversos parâmetros a serem definidos, há a possibilidade de uma série de personalizações no BLAST para sua aplicação. Todavia, a título de organização, consideram-se oficiais as versões do *NCBI-BLAST* desenvolvidas pelo *National Center for Biotechnology Information BLAST* e as divulgadas pelo *WU-BLAST Washington University BLAST*. Estas versões são bastante diferentes. Nestas versões do BLAST há um tratamento especial no banco de dados que a análise vai ser feita antes de sua execução, isto é, pré-computando os dados do banco. O algoritmo propriamente dito, continua obedecendo as etapas descritas, exceto para a versão do *Gapped BLAST* (ALTSCHUL et al., 1997) , ou *BLAST 2.0* que tem sua própria sequência de dados. Dentre as versões de destaque do BLAST, estão:

- BLAST para sequências de Proteínas: Na literatura é denominado BLASTp, sendo a letra P indicando que é uma variação de proteínas. Utiliza como sequência de entrada uma sequência de aminoácidos contra um banco de dados de proteínas. Esse tipo de formato do BLAST é muito utilizado quando se tem uma proteína e deseja-se saber se existem, em outros organismos, proteínas similares. Utiliza portanto a matriz de substituição completa, necessitando de uma etapa de pré-processamento para pontuar todas as possíveis sementes. Geralmente, a pesquisa do BLAST, em qualquer versão, é feita entre uma sequência buscada, e um banco de dados, que armazena diversas sequências conhecidas. São instanciadas diversas comparações, e o resultado é o conjunto dos sequências

que apresentarem similaridade, definida através da pontuação, satisfatórias para superar o parâmetro T definido (MOUNT, 2008a) (LOYTYNOJA; GOLDMAN, 2008).

- BLAST para sequências de DNA: Assim como a versão de proteínas, essa versão é denominada BLASTn, com o N indicando que a comparação é feita entre bases nucleotídicas. Possui um espaço de busca reduzido, pois o alfabeto é limitado a 4 letras. Com isso, também utiliza matrizes de pontuação simplificadas. Essa particularidade das comparações com bases de DNA, nos permitiu uma importante simplificação que será apresentada na arquitetura proposta, pois, baseado em uma matriz de substituição simplificada, pudemos atribuir valor binário 1 para as similaridades encontradas, *matches* e valor binário 0 para as desigualdades, *mismatches*. O resultado da etapa de semeadura é uma matriz preenchida com valor da pontuação obtida, 1 para a nossa generalização para DNA (ALTSCHUL et al., 1990).

- Uma versão do BLAST é o TBLASTN que compara uma sequência buscada de proteína contra um banco de dados de sequências de nucleotídeos dinamicamente traduzido em todas as fases de leitura (MOUNT, 2008a) (KORF; YANDELL; BEDELL, 2003).

- Outra versão é o TBLASTX que compara as traduções das seis fases de leitura de uma sequência buscada de nucleotídeos contra as traduções das seis fases de leitura de um banco de dados de sequências de nucleotídeo (MOUNT, 2008a) (KORF; YANDELL; BEDELL, 2003).

- O *Gapped BLAST*, é a versão do BLAST que aceita a inserção de espaços vazios entre os alinhamentos, e embora tenha incorporado novas heurísticas que melhoraram a sensibilidade e o desempenho, ainda é mais importante por considerar *gaps* provocados por inserções ou deleções na sequência. Isso significa dizer que a versão do *Gapped BLAST*, permite alinhar sequências genéticas, mesmo que essas tenham sofrido mutação ou evolução. A inserção de um buraco, em inglês *gap* funciona como um caractere coringa, *wildcard*, que permite continuar a comparação, mesmo que encontrado um *mismatch*. Nas matrizes de pontuação é comumente atribuído valor 0 a inserção ou deleção de um *gap*, de modo que ele não influencie na avaliação do escore. Uma outra possibilidade do *Gapped BLAST* é considerar as extremidades dos alinhamentos encontrados pelo BLAST sem *gaps* e utilizar o algoritmo de *Smith-Waterman* para maximizar esse alinhamento local. Com as modificações realizadas, o *Gapped BLAST* realiza a seguinte sequência (ALTSCHUL et al., 1997):

1. Remoção de regiões de baixa complexidade ou repetições na sequência a ser pesquisada;
2. Formação da lista de *k-tuplas* a partir da sequência a ser pesquisada e da matriz de substituição;
3. Montagem de autômato finito determinístico a partir da lista de *k-tuplas*;
4. Varredura do banco de dados de sequências a procura de *hits*;
5. Extensão dos *hits* para formar *HSP*;
6. Avaliação da significância do *HSP*;
7. Alinhamento de vários *HSP* próximos;
8. Execução do *Smith-Waterman* nos alinhamentos locais.

Além das versões descritas, existem ainda outras versões, como por exemplo RPS-BLAST, PSIBLAST, PHIBLAST, BLASTX, TBLASTN, TBLASTX que trabalham, em cada caso, com uma variação das versões supracitadas (MOUNT, 2008a) (LOYTYNOJA; GOLDMAN, 2008) (ALTSCHUL et al., 1990) (ALTSCHUL et al., 1997).

2.3 Trabalhos Relacionados

Inúmeras abordagens e técnicas para o alinhamento de sequências genéticas utilizam dos mais diversos algoritmos e, apresentam-se como alternativa para solução do problema da comparação de sequências. Além das soluções por *software*, algumas implementações em *hardware* dos algoritmos de alinhamento de sequências se destacam, e são brevemente descritas a seguir, por serem convergentes com o objeto dessa dissertação.

Para uma implementação em *hardware*, independente da aplicação, a lógica reconfigurável tem sido utilizada visando solucionar problemas que demandem alto custo computacional. No domínio da comparação de sequências, o trabalho de Hoang (HOANG, 1992) foi precursor ao propor a utilização de FPGAs para construir a máquina SPLASH2 (HOANG, 1993), visando implementar o algoritmo de Needlman-Wunsch.

O algoritmo de Smith-Waterman também possui relevantes resultados obtidos com a utilização de *hardware* reconfigurável. Inicialmente o algoritmo foi implementando utilizando a plataforma it JBits (GUCCIONE; KELLER, 2002) no trabalho de Guccione. Após, o mesmo algoritmo foi implementado em *Virginia Tech.* (PUTTEGOWDA, 2003), e mais recentemente

na *Nanyang Technological University* (OLIVER, 2005). Todas essas implementações tiram proveito da programação dinâmica para encontrar de forma eficiente a solução de alinhamento, utilizando o processador com um vetor sistólico.

Por suas características de construção, FPGAs são altamente susceptíveis para o design de processador sistólicos, e em alguns casos, as estruturas das FPGAs são otimizadas para criar tabelas de pesquisas (*lookup tables*). Essas características construtivas, aliada ao baixo custo relativo das FPGAs estimulam a construção de sistemas para o alinhamento de sequências nos mais diversos algoritmos. Em contrapartida, os trabalhos com lógica reconfigurável, abordando especificamente o algoritmo BLAST, ainda são bastante reduzidos. Nesse sentido, cabe destacar o trabalho realizado por Muriki (MURIKI, 2005) que apresenta de forma detalhada em FPGA BLAST, a possibilidade de elaborar uma arquitetura para o algoritmo utilizado nessa dissertação. A proposta do trabalho de Muriki é melhorar a performance do algoritmo BLAST realizando em *hardware* as partes onde o *software* consome mais tempo de processamento. Após implementar em *software* o algoritmo BLAST, Muriki detectou o consumo de cada etapa, e criou um procedimento de chamada, no qual o programa utiliza a plataforma FPGA. Embora tenha projetado o sistema para atingir um *speed up* de até 35 vezes, os resultados práticos revelaram que a combinação *hardware-software* era até 5 vezes mais lenta que a versão totalmente em *software*. Os resultados obtidos evidenciaram, à época, a dificuldade em desenvolver uma arquitetura de alta performance para o algoritmo BLAST. A contribuição desse trabalho foi apresentar um estudo detalhado do tempo de processamento consumido por cada etapa do algoritmo BLAST, além de ratificar através de seus experimentos, que o grande gargalo de desempenho em uma abordagem FPGA dos antigos modelos, como utilizado por Muriki, reside na entrada e saída de dados.

A comparação entre *hardware* e *software* tem sempre um elemento fortemente presente: o desempenho. Nesse sentido, o trabalho mais notável envolvendo o algoritmo BLAST é o sistema de aceleração DeCypher (CORPORATION, 2011), que consiste numa nova família de FPGAs otimizadas para bioinformática. A solução, desenvolvida e lançada comercialmente pela TimeLogic, baseia-se em cartões PCI, que são conectados a servidores e baseados em tecnologia FPGA (LUETHY; HOOVER, 2004). A solução Tera-BLAST, que utiliza todo esse arcabouço, apresenta resultados de até 400 vezes de *speed up*, mas apesar desses resultados impressionantes, não são detalhados o número de *chips*, recurso de I/O, tipo de arquitetura e nem são descritos as condições que os testes foram realizados.

Outro trabalho que apresenta resultados expressivos foi proposto por Kasap e Benkrid

(KASAP; BENKRID, 2010), que propuseram um arquitetura de alta performance baseada em FPGAs, que alinham sequências de até 1000 bases, com resultado de *speed up* médio de 52 vezes. Na abordagem de Kasap e Benkrid, o algoritmo BLAST é dividido em etapas, e para cada uma dessas etapa é reservado um determinado número de recursos na FPGA para garantir a execução das tarefas dessa etapa. A arquitetura é totalmente parametrizável, podendo assumir, para sequências de até 1000 bases, qualquer parâmetro para ajuste do algoritmo BLAST. Toda a arquitetura trabalha de forma paralela, e foi implementada em uma placa FPGA, exceto a etapa mais onerosa do algoritmo, o pré-processamento e divisão de sequências, que é realizada em *software* em um computador hospedeiro. Cabe também ao computador que roda a versão do *software*, eleger um perfil de configuração para a placa FPGA de acordo com os parâmetros informados. Os testes de desempenho foram realizadas para um tipo específico de base de dados *Swiss-Prot*, pesquisando em um banco de dados armazenado em memória.

Dentre todos os trabalhos que utilizam FPGAs para implementação do algoritmo BLAST, o que menos realiza alterações na estrutura do algoritmo original, e consequentemente o que possui resultados de alinhamento mais próximos deste algoritmo é o Mercury BLASTn (BUHLER et al., 2007). Essa proposta implementa o algoritmo BLAST em lógica reconfigurável, e utiliza várias FPGAs trabalhando em paralelo. O Mercury trás resultados expressivos, pois em 99 % dos casos testados a resposta encontrada pelo Mercury é idêntica à encontrada por *software*, um valor excelente quando considerado um método heurístico. Quanto ao desempenho, a implementação Mercury para o algoritmo BLAST consegue um *speed up* médio de até 11 vezes, quando comparado ao *software*.

Embora alguns exemplos de implementação apresentem-se com resultados importantes, a grande variação de versões existentes para o algoritmo BLAST dificultam a comparação fidedigna entre cada uma das implementações. O funcionamento do algoritmo BLAST e suas versões serão detalhadas no Capítulo 2.

2.4 Considerações Finais do Capítulo

Neste capítulo foi apresentado o algorítimo BLAST e suas variações. Foi apresentado o funcionamento do algorítimo, sua formalização e complexidade matemática, além da descrição de cada etapa e apresentação em pseudo-código da versão para a qual foi desenvolvida a arquitetura. Neste capítulo, visou-se detalhar a importância de cada um dos parâmetros do BLAST, relacionando e familiarizando os termos utilizados, e mostrando o impacto de cada parâmetro escolhido em termos do espaço de busca. No capítulo a seguir, é apresentada a arquitetura

proposta para executar o algorítimo BLAST para o alinhamento de sequências de DNA, como descrito nesse Capítulo.

Capítulo 3

ARQUITETURA PROPOSTA

A ARQUITETURA proposta visa implementar o algorítimo BLAST, para encontrar o melhor alinhamento entre duas sequências de DNA. Dadas duas sequências a serem alinhadas, uma sequência alvo de tamanho m, representada por s; e uma sequência buscada de tamanho n, representada por t; a resposta desejada é a posição relativa onde as duas sequências se relacionam.

Na arquitetura proposta, as sequências a serem alinhadas são codificadas em representação binária e armazenadas em dois registradores. Cada registrador terá o tamanho exato da sequência que for armazenar, sendo tamanho m para o registrador que armazena a sequência s e tamanho n para o registrador que armazena a sequência t. A arquitetura de *hardware* proposta, vista na Figura 3, possui como entradas um sinal de *clk*, que é responsável pelo sincronismo do sistema; a sequência alvo, s, que é armazenada em um registrador de m posições; e a sequência buscada, t, que é armazenada em um registrador de n posições. Essas informações são inseridas nos registradores através de carga paralela. Na saída da arquitetura proposta, são apresentadas as possíveis soluções de alinhamento no formato (i, j, d), onde i é a posição inicial do alinhamento na sequência s; j a posição inicial do alinhamento na sequência t; e d o número de bases idênticas seguidas nos par de sequências (deslocamento do alinhamento encontrado). Todos os resultados são armazenados em uma memória de dados de 3 vias: i, j e d.

O algoritmo BLAST necessita que os seus parâmetros w, T e S sejam definidos, e estes também são informados a arquitetura. Além deles, na arquitetura paralela proposta existe o parâmetro p que determina o número de estruturas paralelas que serão utilizadas para a computação dos dados. Quanto mais processadores, mais paralelo será o processamento, e a tendência é que o resultado seja gerado mais rapidamente. Como descrito no Capítulo 2, o princípio do BLAST é dividir as sequências s e t em subsequências menores. O parâmetro w

determina o tamanho das subsequências que são criadas a partir de t. Os parâmetros T e S representam as pontuações mínimas para que um determinado trecho seja considerado como um resultado válido. A análise do impacto da variação de cada parâmetro é objeto do Capítulo 6.

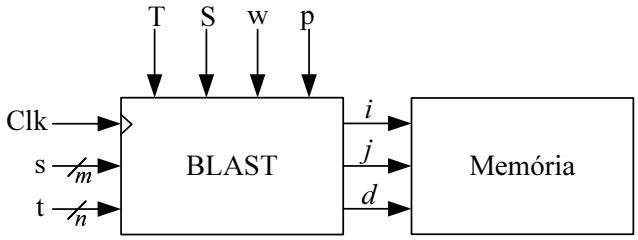


Figura 3: Entrada e saída de dados e parâmetros

Para gerir todos os sinais essenciais ao funcionamento da arquitetura de *hardware* proposta foi implementado um controlador global. Esse controlador é uma máquina de estados que tem a função de prover os sinais necessários para inicializar e sincronizar os componentes, realizar a carga dos registradores com os valores das sequências a serem alinhadas e testar as variáveis de forma a conduzir as etapas do algorítimo BLAST. Como descrito no Capítulo anterior, o algorítimo BLAST é dividido em três etapas: semeadura, extensão e avaliação. A etapa de avaliação, que classifica estatisticamente cada um dos alinhamentos encontrados, não foi implementada em *hardware*, podendo ser realizada em *software* sobre todo o conteúdo da memória. Cada etapa executa um conjunto de operações completamente diferente, sendo que a etapa de avaliação precisa dos dados fornecidos pela extensão, que por sua vez depende dos valores vindos da semeadura. Na arquitetura proposta, cada etapa tem seus componentes dedicados, e para cada etapa há um controlador de etapa. Os controladores de cada etapa e o controlador global trabalham como escravos e mestre, respectivamente. O controlador global é também responsável por todas as operações lógico-aritméticas da arquitetura.

A função dos controladores de cada etapa é prover os sinais necessários para que os componentes executem as tarefas necessárias dentro do escopo definido pelo algoritmo BLAST para cada etapa específica. Por exemplo, o controlador de semeadura é responsável por todos os sinais inerentes à etapa de semeadura, assim como o controlador de extensão é responsável por todos os sinais relativos à etapa de extensão. Por trabalharem como escravos do controlador global, os controladores de etapa precisam informar ao controlador global o início, andamento e término de cada tarefa que eles executam. Essa comunicação é feita através de *flags* de requisição e interrupções.

Respeitando a divisão de etapas proposta pelo algoritmo BLAST, na Seção 3.1 é apresentada a macro-arquitetura, através de diagrama de blocos, onde serão detalhados os componentes comuns a todas as etapas. Após, na Capítulo 4 detalha-se o *hardware* responsável pela semeadura, enquanto no Capítulo 5 é abordado a etapa de extensão.

3.1 Macro Arquitetura

Dadas duas sequências a serem alinhadas, s e t, o conteúdo dessas sequências é codificado em representação binária e armazenado nos registradores. Após ser realizada a carga dos registradores com os dados dessas sequências, o controlador global fornece aos controladores de etapa os sinais necessários para que se inicie o processamento dos dados, percorrendo as etapas do algorítimo BLAST: semeadura e extensão, como apresentadas no Capítulo 2.

No tocante ao algorítimo BLAST, observa-se que a terceira etapa, avaliação, depende dos dados da segunda etapa, de extensão, que por sua vez depende dos dados encontrados na primeira etapa, a semeadura. Todavia, na etapa de semeadura varre-se o espaço de busca com o objetivo de encontrar todas as sementes contidas, para uma determinada sensibilidade. Essa tarefa consome diversos ciclos, e as sementes encontradas ao longo desse período são armazenadas, pois serão utilizadas pela próxima etapa, a extensão. A divisão de etapas e a comunicação entre elas, e o controlador global é observada na Figura 4. Há um barramento de dados e de controle que realiza a troca de dados entre a semeadura e a extensão. Esse barramento é controlado por um árbitro, que também é gerido pelo controlador global.

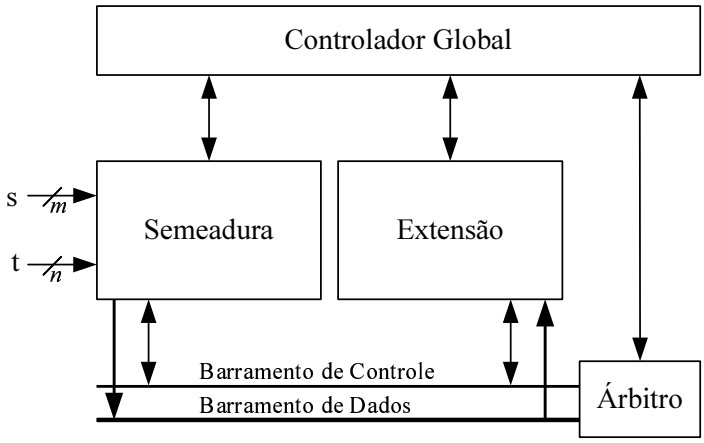


Figura 4: Macro-Arquitetura: Divisão pelas etapas do algoritmo

Existem duas abordagens para a elaboração de um *hardware* que executa tarefas divididas em etapas. A primeira abordagem é realizar o algoritmo, etapa por etapa, executando a primeira etapa, e armazenando os resultados, e ao fim desta, iniciando segunda etapa. Essa

abordagem possibilita o gerenciamento mais simples das tarefas, e consequentemente um *hardware* menos elaborado, principalmente no tocante ao roteamento e controle. Nessa abordagem, o gargalo de desempenho surge quando uma etapa demande um alto tempo de processamento, ocasionando uma ociosidade nas etapas subsequentes. Supondo por exemplo, um sistema de duas etapas, E_1 e E_2. Na abordagem simplificada, a primeira etapa E_1 executa todas as suas tarefas, e só quando termina todas suas tarefas, envia o resultado para a etapa seguinte: E_2.

Outra abordagem possível é a realização de um *pipeline* entre etapas do algoritmo. Nessa abordagem, embora as divisões de etapas sejam validas, há uma diferença essencial, pois o *pipeline* permite que enquanto uma parte do *hardware* esteja executando uma tarefa, outra parte do mesmo *hardware* execute outra tarefa. Na abordagem utilizando *pipeline*, quando a etapa E_1 dispõe de algum resultado, mesmo que parcial, esse resultado já é enviado para a etapa E_2 que inicia o processamento desse resultado, mesmo que E_1 ainda não tenha finalizado todas as suas tarefas (TANENBAUM, 2007).

A estratégia utilizada nesse trabalho é realizar *pipeline* entre as etapas do algoritmo BLAST. Esta, baseia-se na identificação de uma semente, a qualquer tempo, e a partir dessa identificação, iniciar, para essa semente, a etapa de extensão, independentemente das demais sementes. Quando um dado está disponível, inicia-se a tarefa subsequente que depende desse dado, instanciando um componente exclusivo para essa finalidade. Partindo dessa definição, configura-se o *pipeline* entre as etapas, pois enquanto um componente estará executando a semeadura para uma semente, uma outra parte do *hardware* proposto já estará executando a extensão e uma outra semente.

Além do *pipeline* entre as etapas do algoritmo BLAST, criou-se dentro de cada etapa o paralelismo estrutural das sub-tarefas. Para a semeadura, foram implementados $n - w + 1$ estruturas idênticas para encontrar as sementes. Na extensão, são p estruturas idênticas visando maximizar a pontuação das sementes recebidas. O gerenciamento simultâneo dessas etapas justifica a necessidade do controlador global, para conduzir essas combinações, além de um árbitro para realizar o roteamento. Dentro das etapas, o paralelismo estrutural implementado justifica a necessidade de um controlador para cada etapa. Esse paralelismo estrutural é visto na Figura 5.

Em geral, o controlador é um ponto muito impactado em uma arquitetura paralela ou com *pipeline*. Quando opta-se por uma arquitetura nessas configurações, em detrimento de uma sequencial, objetiva-se o ganho em *speedup*, ao custo de um aumento de área de *hardware*. Ao invés de tarefas grandes e complexas, o trabalho é dividido em tarefas menores, e para

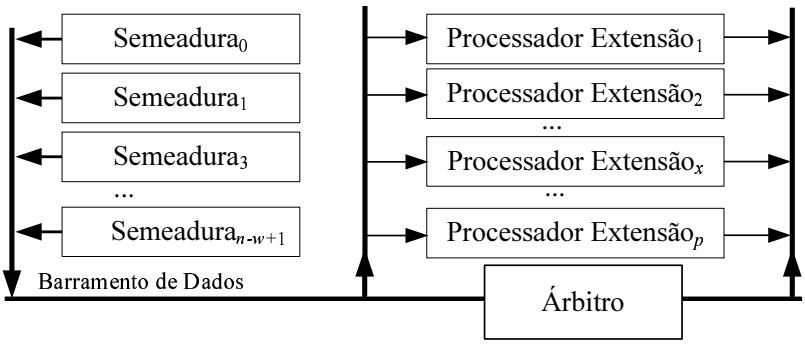


Figura 5: Macro-Arquitetura: Paralelismo estrutural das etapas de semeadura e extensão

as mais impactantes em termos de tempo de execução, são criados vários blocos funcionais simples, duplicados, que executam essa mesma tarefa ao mesmo tempo. Com o aumento do número de blocos funcionais, o controle passa a ser mais complexo, não só pelo aumento de sinais, mas pela necessidade de executar novas funções, como sincronização e chaveamento.

O algoritmo BLAST é divido em três etapas principais: semeadura, extensão e avaliação. Portanto existem três controladores de etapa, e um controlador global. A este controlador cabe gerir o *pipeline*, todas as funções que dependem de uma interseção entre etapas distintas e todas as operações lógico-aritméticas (soma e incremento, subtração e comparação). Os demais controladores, exercem o gerenciamento de suas etapas, reportando-se ao controlador global. São exemplo das funções que cabem ao controlador global:

- Inicialização dos componentes,
- Carga dos registradores s e t,
- Armazenamento em memória dos resultados,
- Informar resultados parciais e final,
- Operações lógico-aritméticas,
- Contadores do Sistema.

Na etapa de semeadura, o controlador deve garantir a divisão lógica das palavras. Dessa forma ele é o responsável pelos registradores s e t. O algorítimo da semeadura é apresentado no Capítulo 2, e o detalhamento do *hardware* responsável por essa etapa está no Capítulo 4. As principais funções desse controlador são:

- Controle do deslocamento dos registradores s e t,

- Controle da lógica de comparação, e chaveamento entre conteúdo dos registradores s e t, de acordo com deslocamento e parâmetros definidos na entrada do algoritmo.

De forma análoga, a etapa de extensão tem um controlador, que provê os sinais necessários para a extensão das etiquetas. Optou-se por deixar esse controlador responsável apenas pelos processadores de extensão, enquanto o controle dos barramentos, gerenciamento das filas e o roteamento das etiquetas foi atribuído ao controlador global e ao árbitro. Com isso, o controlador da etapa de extensão, é o responsável, por exemplo, por prover os sinais para:

- Extensão das sementes,
- Gerenciamento dos p processadores de extensão.

A interação entre as etapas se dá através do controlador global e do barramento de dados e controle, que é gerido pelo árbitro. Na Seção 3.1.1 é detalhado o funcionamento do árbitro, e exposta a lógica de prioridade utilizada para roteamento entre semeadura e extensão. Na Seção 3.1.2 é apresentado o controlador global, descrita a função de cada um de seus sinais e o diagrama de transição de estados relacionado.

3.1.1 Árbitro

Por se tratar de uma proposta para uma arquitetura paralela, pressupõe-se que quanto mais blocos funcionais idênticos trabalhando juntos, melhor a resposta em desempenho. Porém, é preciso estudar a relação de área de *hardware* $\times$ desempenho, comparando-se o aumento de área e da lógica de controle contra os benefícios de tempo de resposta, antes de paralelizar alguma estrutura. Nesse trabalho, foi determinado o número de cada um dos componentes principais que melhor responde a essa relação custo/benefício. Foi determinado o número de blocos paralelos para a semeadura e o número de blocos paralelos para a extensão. No Capítulo 6 são justificados, através de resultados de síntese e simulação, o motivo da escolha desses valores.

Para a extensão, foi possível verificar que um número fixo de processadores de extensão, entre 2 e 5, atende na maior parte dos casos, independente do tamanho de w. Quando se ultrapassa o número de 5 processadores, o tempo ocioso de alguns processadores aumenta abruptamente e consequentemente, a melhora do tempo de execução é irrelevante, quando comparada ao acréscimo de área de *hardware*. Logo, optou-se por paralelizar os processadores de extensão, mas com um número fixo determinado de forma empírica: $2 < p < 5$.

A partir da determinação do número de processadores de extensão, há a possibilidade do número de sementes geradas, de até $(n - w + 1)$ por ciclos, ser diferente da quantidade de processadores de extensão existentes. Por esse motivo, fez-se necessário desenvolver um componente para controlar o acesso das sementes encontradas e balancear o tráfego entre os processadores de extensão, de modo a distribuir a carga de forma otimizada entre os processadores de extensão. Esse componente, é o árbitro que está descrito a seguir.

A inserção de múltiplos processadores que podem ser compartilhados por mais de uma linha de dados traz consigo, além da divisão da carga de trabalho, a necessidade de controle de acesso dos barramentos de entrada e saída, assunto amplamente abordado nos protocolos de comunicações (TANENBAUM, 2000).

Para a tarefa de divisão da carga de trabalho, importou-se da teoria de sistemas operacionais, os conceitos necessários para a construção do árbitro, que é também um escalonador.

Versa a teoria de sistemas operacionais que, numa arquitetura paralela, é factível 2 ou mais processos estarem aptos a utilizar um processador para ser executado. Nesse instante, o sistema operacional deve decidir qual dos processos aptos, armazenados em uma fila, será escolhido para rodar primeiro. Essa tarefa e a tomada de decisão é feita pelo escalonador de processos através da implementação de alguns algoritmos de seleção, denominados algoritmos de escalonamento. O escalonador é a entidade do sistema operacional responsável por selecionar um processo apto a executar no processador e dividir o tempo do processador de forma justa entre os processos que estão aptos (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

A Figura 5, mostra que são $(n-w+1)$ blocos que executam a semeadura paralelamente. Na saída de cada um desses blocos há uma fila, que armazena as sementes encontradas ao longo do processo até que haja um processador disponível. Essas $(n-w+1)$ filas, foram construídas no padrão *fifo* (*first in first out*). Cada semente armazenada, é um processo apto aguardando o processador para ser executado, e quando selecionadas, precisam ser roteadas para um dos processadores de extensão através do barramento de dados.

O árbitro destaca-se por realizar três importantes funções simultaneamente. Para os processadores que realizam a extensão, executa a função de escalonador. Há também a interface entre semeadura e extensão, com os barramentos dados, exercendo a função de controlador de barramentos. Para as filas localizadas na saída de cada um dos $(n-w+1)$ blocos que executam a semeadura paralelamente, o árbitro monitora a ocupação essa filas, na função de gerenciador de filas.

Na função de gerenciador de fila, o árbitro trabalha determinando qual fila tem a priori-

Tabela 11: Lógica de desempate entre filas

Requisição da Fila	Resposta do Balanceador	Operação Resultante
XXX1	0001	*POP* na fila 1
XX1X	0010	*POP* na fila 2
X1XX	0100	*POP* na fila 3
1XXX	1000	*POP* na fila 4

dade para enviar sua etiqueta para o processador de extensão disponível. O ganho obtido com esse gerenciador de fila permite que o número de processadores seja independente do numero de filas e módulos de comparação.

Como descrito acima, foi necessário inserir uma memória do tipo *fifo* (*first in first out*) para armazenar as sementes até que essas fossem estendidas pelos processadores. A largura da *fifo* é determinada pelo tamanho das palavras, enquanto sua profundidade é derivada do número de processadores de extensão incluídos, de modo que quando há um tempo médio necessário para os processadores liberarem os dados, há necessidade de uma fila mais profunda. A menos que a *fifo* esteja cheia, a etapa de semeadura não vai interferir no processo de extensão, pois as sementes serão gradativamente armazenadas na *fifo*.

Entre a saída das filas no padrão *fifo* e os processadores de extensão, há o árbitro que monitora as filas e toda vez que um processador está disponível, direciona a semente contida na fila prioritária para ser estendida. Esse monitoramento é feito através de *flags* oriundos dos processadores e das filas. Para escalonar os processadores, o algoritmo utilizado é o *round-robin*, que direciona trabalho aos processadores de forma circular, a cada rodada. Caso o processador da rodada ainda esteja ocupado, o próximo processador é escolhido.

A lógica de controle para escolher a fila prioritária, é sempre escolher a fila mais cheia, isto é, com mais posições ocupadas. Caso duas ou mais filas estejam igualmente ocupadas, o critério de desempate é a fila de menor número, como é visto na Tabela 11.

O princípio de funcionamento da função do gerenciador de filas do árbitro é monitorar os sinais de *flag* de cada fila identificando qual deve ser a prioritária, por estar mais cheia. Caso mais de uma fila esteja cheia e não haja processadores de extensão disponíveis para todas elas, o árbitro gera uma interrupção para o controlador global, que irá desabilitar o *clk* da etapa de semeadura até que as filas tenham posições disponíveis para receber novas sementes. A Figura 6 mostra a lógica combinacional para conceder do barramento à uma fila, a partir da requisição do processador da rodada, repassado pelo árbitro. O árbitro determina o *id* da fila a ser usada, baseado nas posições das filas. Esse *id* fornecido pelo árbitro é comparado com o *id* de cada fila, e quando há a requisição, a lógica habilita o *buffer tri-state* somente da fila escolhida.

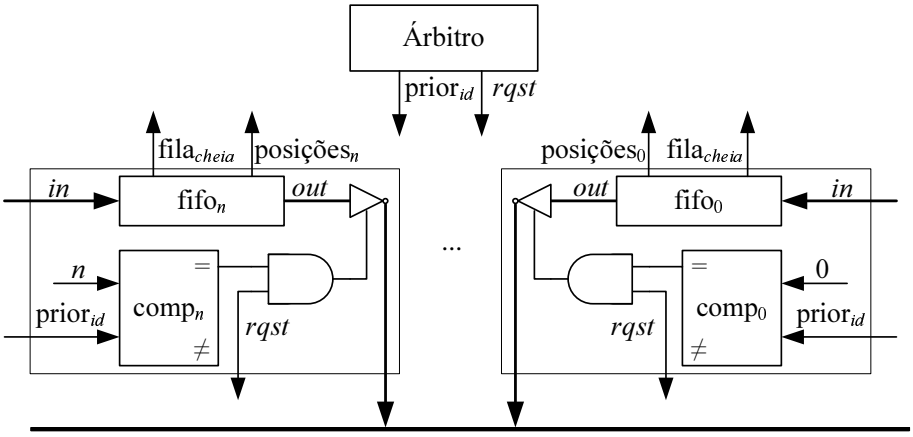


Figura 6: Controle de barramento e prioridade

A escolha da fila a ser utilizada só acontece quando existe um processador livre, solicitando através de um sinal de requisição uma nova semente para então processá-la. No caso de todos os processadores de extensão estarem livres ao mesmo tempo, o árbitro, na função de escalonador, prioriza o processador de menor número, em detrimento dos demais. A escolha do processador de extensão que será utilizado, quando mais de um estiver requisitando dados, é como descrito, feito através do algoritmo *round-robin*.

Todas as memórias contidas na semeadura, são do padrão de fila *fifo*, *first in first out*. Para o gerenciamento dessas filas, importou-se um bloco IP (*Intellectual Property*), artifício utilizado sempre que um componente ou bloco é devidamente difundido, como é o caso dos módulos de memória, sem que seja acrescentada nenhuma especificidade para o projeto em questão.

3.1.2 Controlador Global

A escolha de projeto foi elaborar uma arquitetura distribuída, com controladores distribuídos, trabalhando orquestrados por um controlador global. A metodologia foi agrupar os componentes que cada controlador iria gerir a partir das etapas do algoritmo BLAST, que são três. Porém, na implementação de um sistema paralelo, existe a necessidade de algumas tarefas, como sincronização e chaveamento, que também precisam ser gerenciadas por algum controlador. Nesse caso, o responsável por essas funções de interseção é o controlador global.

Para controlar os sinais necessários, o controlador global foi criado no modelo de máquina de estados, genérica em função do parâmetro p, que é o número de processadores de extensão. As portas de entrada desse controlador recebem sinais de tamanho p e de 1 bit. A Figura 7 apresenta a via de dados desse controlador.

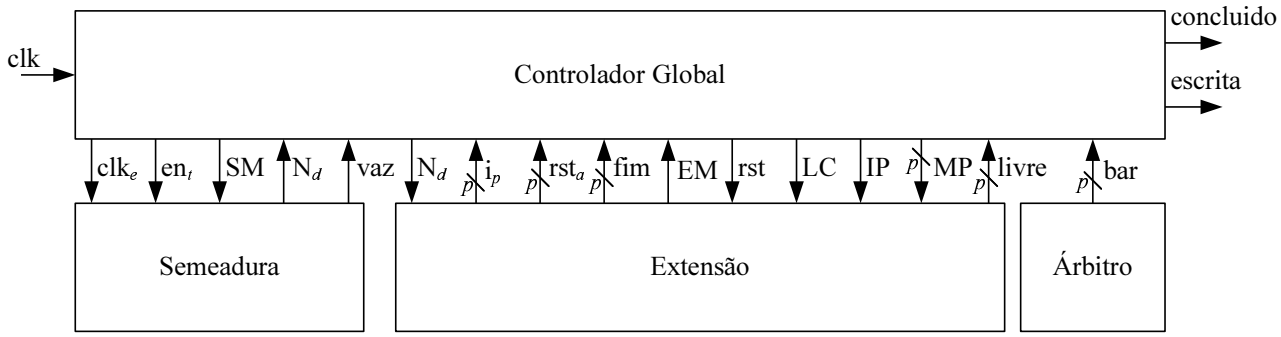


Figura 7: Via de dados controlador global

Os sinais de entrada e saída do controlador global são descritos a seguir, com suas respectivas funções. Os sinais de entrada desse controlador global são:

- *clk*: Sinal de 1 bit, responsável pelo sincronismo do circuito.
- I_p: Sinal de p bits, recebe a informação de qual processadores de extensão é o prioritário, gerada pelo árbitro.
- *Livre*: Sinal de p bits, monitora os sinais de saída dos processadores que indica o estado livre ou ocupado.
- *FIM*: Sinal de p bits, indica, através de nível lógico 1 quando um processador termina a extensão.
- Rst_a: Sinal de p bits, cópia dos sinais de Rst enviados aos controladores dos processadores de extensão no ciclo anterior, gerada pelo árbitro. Para garantir que apenas o processador da rodada será reiniciado.
- *EM*: Sinal de p bits, informação oriunda dos processadores de extensão informando que não foi possível receber os dados enviados, sendo necessário aguardar mais um ciclos de *clk*.
- *vaz*: Sinal de 1 bit, monitora se as filas estão cheias ou vazias. Quando todas as filas estão vazias, assume nível lógico 1.
- N_d: Sinal de 1 bit, monitora a quantidade de operações de deslocamento realizadas no registradores s. Quando atinge esse valor atinge m, o sinal é ativado em 1.
- *Bar*: Sinal de 1 bit, monitora o sinal N_d e os barramentos entre processadores. Quando sistema ocioso, sinal é ativado em 1.

Como apresentado na Figura 7, os sinais de entrada do controlador global, são oriundos dos controladores de etapa e do árbitro. Eles excitam a máquina de estado para que se produza os seguintes sinais de saída:

- *clke*: Sinal de 1 bit, habilita ou desabilita o *clk* dos demais componentes.
- *SM*: Sinal de 1 bit, seleciona entre entrada paralela e entrada serial dos registradores s e t.
- en_t: Sinal de 1 bit, trabalha como *enable* do registrador t.
- *IP*: Sinal de 1 bit, inicializa o processador de extensão.
- *LC*: Sinal de 1 bit, seta a carga paralela dos contadores.
- Reset: Sinal de 1 bit, responsável pelo *Rst* enviado aos processadores de extensão.
- *escrita*: Sinal de 1 bit, habilita a gravação do resultado do processador extensão em memória.
- *MP*: Sinal de p bits, sinaliza aos processadores através de nível lógico 1, para iniciar o processamento de uma etiqueta.

A partir dos sinais de entrada e saída descritos acima, o controlador global, gerencia a arquitetura proposta, através da seguinte sequência de estados apresentada na Figura 8.

- S0: Estado de inicialização do sistema, através dos sinais de saída *clke*, *IP* e *LC*.
- S1: Realiza a carga paralela de todos os registradores, através do sinal de saída *SM*.
- S2: Registradores carregados, são designados para receber dados da entrada serial, através dos sinais de saída *SM* e En_t. Avalia se as filas estão vazias ou se já existe alguma semente armazenada em uma delas.
- S3: Aguarda até que haja uma semente armazenada em uma das filas.
- S4: Informa ao sistema que há uma semente aguardando e qual será o processador que irá recebe-la através do sinal *MP*. Testa o sinal de entrada *EM*.
- S5: Testa os sinais de entrada *FIM*, para verificar se algum processador de extensão terminou sua tarefa.

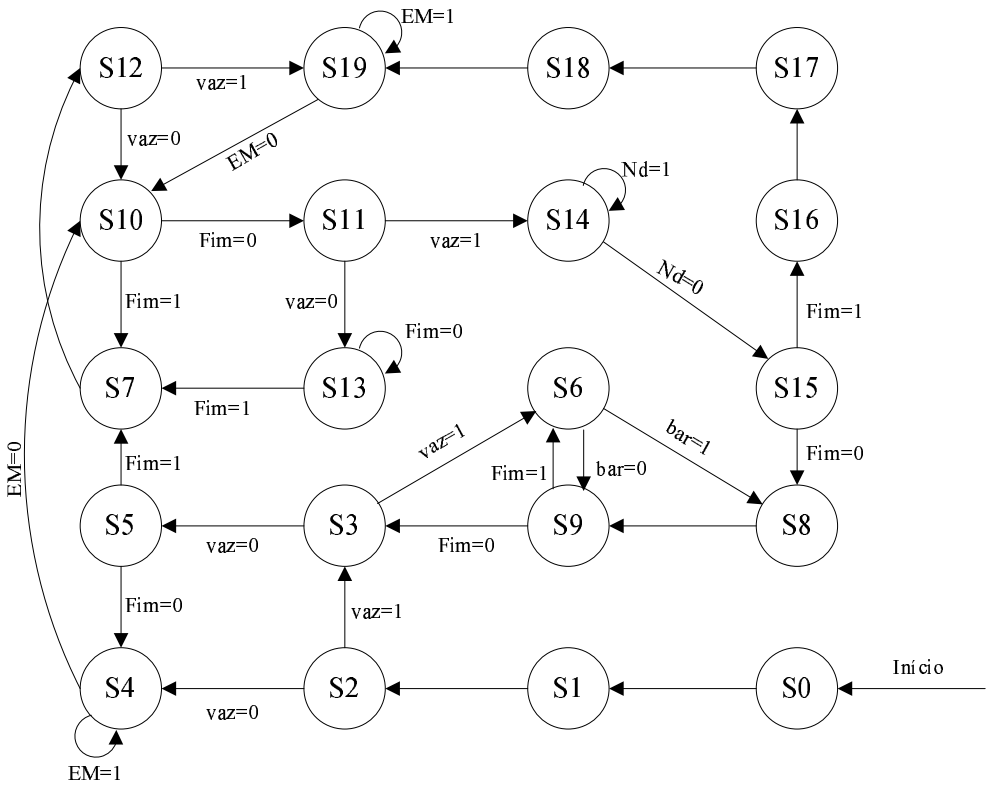


Figura 8: Diagrama de Transição de Estados

- S6: Testa o sinal de entrada *Bar*, que é a ociosidade do barramento.
- S7: Grava o valor encontrado nos processadores de extensão em memória, através do sinal de saída *escrita*. Provê os sinais necessário para dar início ao ciclos de escrita.
- S8: Informa através do sinal de saída *concluído*, que o sistema terminou todas as tarefas.
- S9: Testa os sinais de entrada *FIM*, para verificar se mais algum processador de extensão terminou sua tarefa.
- S10: Testa os sinais de entrada *FIM*, para verificar se algum processador de extensão terminou sua tarefa.
- S11: Aguarda até que haja uma semente armazenada em uma das filas.
- S12: Finaliza o ciclos de escrita em memória e envia o sinal de *Rst* ao processador de extensão. Testa o sinal de entrada *vaz*.
- S13: Testa os sinais de entrada *FIM*, para verificar se mais algum processador de extensão terminou sua tarefa.
- S14: Testa o sinal de entrada N_d.

- S15: Testa os sinais de entrada *FIM*, para verificar se mais algum processador de extensão terminou sua tarefa.
- S16: Aguarda um transição de *clk* para que a ultima semente esteja disponível para gravação em memória.
- S17: Grava os valores remanescentes nos processadores de extensão em memória, através do sinal de saída *escrita*. Prove os sinais necessário para dar início ao ciclos de escrita.
- S18: Finaliza o ciclos de escrita em memória e envia o sinal de *Rst* ao processador de extensão.
- S19: Informa que há uma semente aguardando e qual será o processador que irá recebe-la através do sinal *MP*. Testa o sinal de entrada *EM*.

3.2 Considerações Finais do Capítulo

Neste capítulo foi apresentado a macro-arquitetura proposta, a divisão do algoritmo em etapas e detalhou-se os componentes utilizados para interface e controle dessas etapas. Para os componentes criados, foi descrito a necessidade e o funcionamento de cada um deles. A premissa de projeto foi obedecer ao máximo a divisão das etapas do algorítimo, mas por conta do paralelismo implementado, algumas funções de etapas diferentes se sobrepõem, enquanto outras funções, dentro da mesma etapa, precisaram ser separadas. Algumas funções serão melhor entendidas em conjunto com os algorítimos apresentados no Capítulo 2, e algumas das escolhas de projeto mencionadas, como o número de processadores, está justificada no Capítulo 6, onde são apresentados os resultados de simulação e síntese. No próximo capítulo é detalhado o *hardware* que executa a semeadura.

Capítulo 4

SEMEADURA

AO propor uma arquitetura de *hardware* que implementa o algoritmo BLAST, optou-se por obedecer a divisão de etapas do algoritmo: semeadura, extensão e avaliação. Para cada uma dessas etapas, há componentes dedicados e paralelos para realizar cada função do BLAST, conforme apresentado pela macro-arquitetura no Capítulo 3. A seguir, é descrito a arquitetura de *hardware* criada para realizar a etapa de semeadura.

No alinhamento de duas sequências de DNA, os valores dessas sequências são, antes de tudo, codificados em representação binária. Essas sequências são armazenadas em dois registradores, denominados t e s, sendo que o primeiro armazena a sequência a ser procurada, denominada também sequência buscada ou *query*; enquanto o segundo armazena a sequência onde essa primeira sequência será procurada, chamada de sequência alvo ou *subject*. Essa nomenclatura é oriunda da teoria de sistemas de informação, quando realizada uma pesquisa de um valor específico, *query*, em um determinado banco de dados, o alvo. A adoção dessa nomenclatura faz-se relevante pois está em conformidade com a bibliografia usual.

No Capítulo 2, é apresentado o funcionamento do algorítimo BLAST, para condições genéricas. Em termos de *hardware* cada função ou operação do BLAST, resulta em um grupo de componentes. Por exemplo, o BLAST trabalha subdividindo s e t, de comprimentos m e n, em palavras menores, de tamanho w, parâmetro esse que pode ser definido. Desse modo, ao invés de se realizar a comparação das duas sequências inteiras o algoritmo BLAST utiliza essas palavras menores criadas a partir da subdivisão das sequências s e t para otimizar a busca. Por ser um método heurístico, o BLAST consequentemente utiliza as técnicas de heurística para justificar algumas escolhas. O algoritmo parte do princípio que, por exemplo, para que duas sequências tenham alto grau de similaridade, elas devem ter vários pequenos trechos idênticos. Esses trechos idênticos devem ter no mínimo o tamanho w, e para encontrar esses trechos idênticos pode-se, por exemplo, inserir w portas lógicas *não ou exclusivas* de 2 entradas cada,

uma em s e outra em t, entre os trechos a serem analisados. Dessa forma, o nível logico 1 significa que os trechos de s e t, de tamanho w são idênticos.

Uma sequência de DNA pode ter qualquer tamanho, e é representada por um alfabeto codificado em 4 letras possíveis: [A, C, G, T]. O cálculo do número de bits necessários para representar um determinado universo é dado por $n_b = 2^x$, onde x é a quantidade de símbolos presentes no alfabeto e n_b é o número de bits necessários para representar esse mesmo alfabeto. A forma de representação binária possibilita que muitas operações do algoritmo sejam feitas através de lógica combinacional simples. O alfabeto para representar sequências de DNA necessita de 2 bits (4 bases) para ser representado no formato YZ, sendo Y o bit mais significativo e Z o bit menos significativo. Nesse trabalho optou-se por comparar a parte mais significativas das sequências, Y, em um *hardware* separado da parte menos significativa das sequências, Z. Com isso, embora tenha-se duplicado a área de *hardware*, a complexidade das operações foi extremamente reduzida. Nesse Capítulo, serão detalhadas essas escolhas realizadas para o projeto da arquitetura proposta, e também são apresentadas as soluções de *hardware* elaboradas para cada função do algoritmo BLAST.

A primeira ação do BLAST é dividir as sequências s e t em palavras menores de tamanho w. Em *hardware*, a subdivisão supracitada foi idealizada como um mapeamento do conteúdo dos registradores s e t, através da inserção de índices i e j, respectivamente, que mostram em qual posição de cada registrador está a palavra de tamanho w que será utilizada. Com isso, o algoritmo utiliza esses índices como ponteiros, e também como chave para a lógica de roteamento entre os registradores e os comparadores. Supondo a sequência t=TAGC, os índices j serão: t_0 para base T, t_1 para base A, t_2 para base G e t_3 para base C. Ainda para esse exemplo, para dividir essa sequência em palavras de tamanho $w = 3$, os índices são incrementados, à partir de $j = 0$, de forma que as palavras mapeadas serão $palavra_0$ = TAG, ponteiro na posição 2 e $palavra_1$ = AGC, ponteiro na posição 3.

Pelo fato das palavras terem menor comprimento que t, essa comparação tende a ser mais rápida e o algoritmo consegue encontrar através de comparação das sequências quais trechos de tamanho w são minimamente semelhantes para iniciar a tentativa de alinhamento. O significado biológico desse processo, como descrito no Capítulo 2, é que o algoritmo procura nas palavras de tamanho w, regiões de alta similaridade para que possa iniciar a pesquisa a partir daquela região que atingiu uma pontuação mínima estabelecida, T. Para as sequências de DNA, as palavras de tamanho w comparadas precisam ser idênticas para serem consideradas uma semente, sendo assim estendida e avaliada nas etapas subsequentes. Portanto, um par de

segmento (s, t) só é considerado uma semente se eles forem idênticos no trecho de tamanho w analisado.

A macro-arquitetura da etapa de semeadura pode ser vista na Figura 9, onde é mostrado a comunicação entre o controlador de etapa, os registradores s e t e os contadores que, juntamente com os parâmetros w, men são utilizados para a implementar a lógica de roteamento.

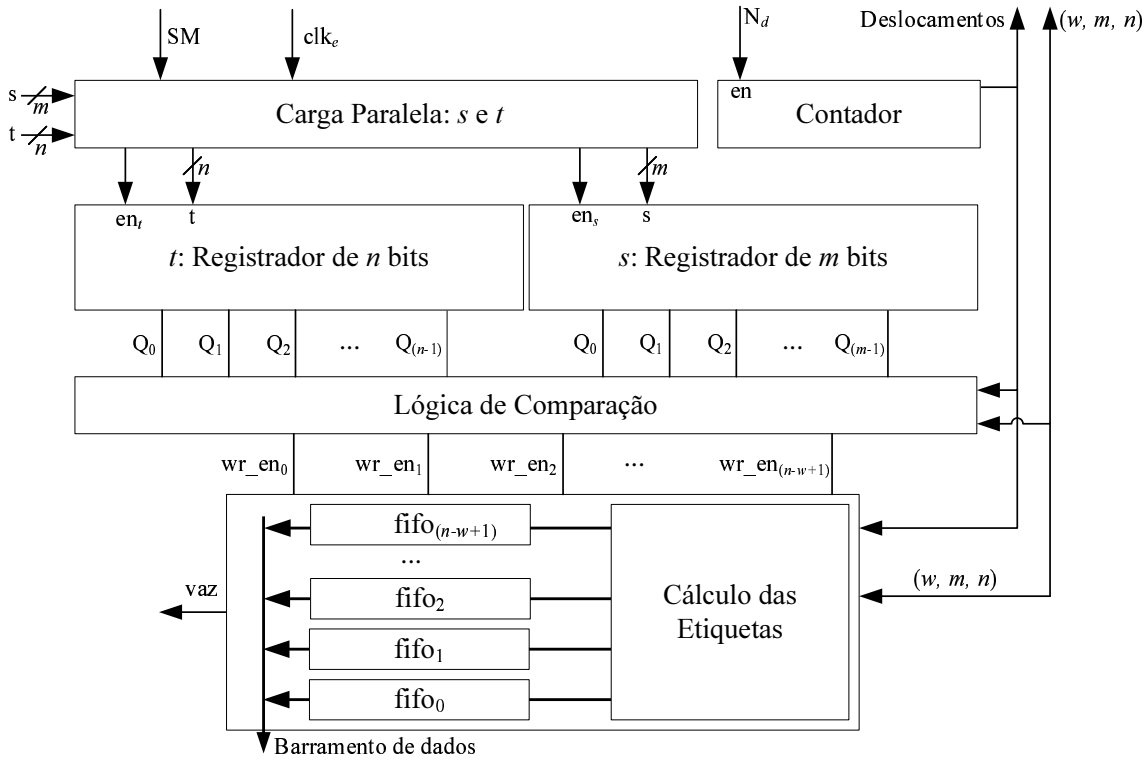


Figura 9: Diagrama funcional da Semeadura

Na Figura 9 são apresentados os componentes para a identificação e comparação das palavras, sendo que mais adiante, na Seção 4.2 é detalhado o como é feito o mapeamento dessas palavras dentro dos registradores que armazenam as sequências a serem alinhadas e a respectiva comparação. É possível ver na Figura 9 que após a adequação dos padrões de entrada, os registradores contém as sequências, na forma de vetores binários que estão ligados à uma lógica de comparação. A lógica de comparação realiza o roteamento utilizando parâmetros w, m e n e pelo número de deslocamentos realizados pelo registrador. Dessa forma, sempre que um trecho de tamanho w é idêntico em s e t, os valores dos índices que representam essa igualdade são armazenados em uma memória do tipo *fifo*, *first in first out*.

Antes de explicar como o *hardware* realiza essas comparações, e o armazenamento dos índices em memória, faz-se necessário demonstrar o método de pré-divisão das sequências a serem alinhadas, e detalhar como o *pipeline* e paralelismo estrutural implementado inter e

intra-etapas interfere na busca do resultado do alinhamento. Na Seção 4.1 são apresentadas as premissas e a metodologia para elaborar a arquitetura proposta em *hardware*, partindo da adequação das entradas, pela codificação e divisão das tarefas.

4.1 Premissas

Como descrito, o cálculo do número de bits necessários para representar um determinado universo é dado por $x = 2_b^n$, onde x é a quantidade de símbolos presentes no alfabeto e n_b é o número de bits necessários para representar esse mesmo alfabeto.

A partir do cálculo de n_b visualiza-se, que para codificar uma sequência de DNA, com seu alfabeto de 4 letras, em bits, serão necessários 2 bits para cada letra. Para utilizar um alfabeto maior, de 26 letras por exemplo, serão precisos de 5 bits para representá-lo. Visto isso, aliado a complexidade matemática abordada no Capítulo 2, verifica-se que para o alinhamento de duas sequências de DNA de tamanho m e n, o espaço de busca será duas matrizes de tamanho $[2, m]$ e $[2, n]$ respectivamente.

Uma matriz $[2, m]$ é equivalente a dois vetores de tamanho m, o mesmo sendo válido para as matrizes $[2, n]$. Logo, uma sequência de DNA de tamanho m, quando convertida em binário, dará origem a uma matriz $[2, m]$ que equivale a dois vetores de comprimento m, sendo um vetor para os bits mais significativos *MSB*, *most significant bit*, e outro vetor para os bits menos significativos *LSB*, *least significant bit*. Porém, em termos computacionais, é mais simples realizar operações em vetores do que em matrizes. Partindo desse princípio, a arquitetura foi elaborada para analisar e alinhar separadamente os vetores *MSB* e *LSB*, ao invés de trabalhar com as matrizes.

Enfatizando, portanto, que para o alfabeto de DNA, onde são necessários 2 bits numa representação no formato YZ, sendo Y o bit mais significativo e Z o bit menos significativo. São tratados, nesse projeto, de forma separada a parte mais significativas das sequências, Y; da parte menos significativa das sequências, Z.

Visando simplificar a estrutura de controle e otimizar o desempenho da comparação, a abordagem vetorial é mais adequada, com 2 registradores com m ou n posições de 1 bit para armazenar cada sequência, em detrimento de utilizar a forma matricial, com 1 registrador com m ou n posições de 2 bits para cada sequência. Nessa abordagem vetorial, todas as estruturas são duplicadas, demandando mais área, porém simplifica-se a lógica de comparação e a quantidade de sinais de controle necessários.

A partir dessa premissa, existem dois registradores com m posições de 1 bit para armazenar a sequência alvo s, sendo que um registrador vai armazenar o vetor dos bits mais significativos, *MSB*; e outro registrador para armazenar o vetor referente aos bits menos significativos, *LSB*. Da mesma forma existem 2 registradores com n posições de 1 bit para armazenar a sequência buscada t, um para o *MSB* e outro para o *LSB*. Para o alfabeto codificado em 4 letras possíveis: [A, C, G, T], a Tabela 12 mostra qual valor binário será recebido em cada registrador.

Tabela 12: Divisão dos símbolos entre *MSB* e *LSB*

MSB	LSB	Simbolo
0	0	A
0	1	C
1	0	G
1	1	T

Supondo como exemplo as sequências s= AGTC e t= GGAC. Representando as sequências em binário: $s = [00, 10, 11, 01]$ e $t = [10, 10, 00, 01]$. Dividindo-as de acordo com a Tabela 12, o vetor dos *MSB* em $s = [0\ ,1\ ,1\ ,0\ ]$ e o vetor dos *LSB* em $s = [\ 0,\ 0,\ 1,\ 1]$. De forma análoga, o vetor dos *MSB* em $t = [1\ ,1\ ,0\ ,1\ ]$ e o vetor dos *LSB* em $t = [\ 0,\ 0,\ 0,\ 1]$. Esses vetores estão armazenados nos respectivos registradores e serão processados de forma independente no *hardware* devido, conforme apresentado na Figura 10.

Os blocos de *hardware* são idênticos, mas trabalham de forma completamente independente, até encontrar uma posição de parada, situação que o sinal *Fim* é ativado, informando ao outro bloco para finalizar a atividade. Essa posição de parada corresponde à uma interrupção externa, e é tratada prioritariamente no controlador global, conforme apresentado no Capítulo 3.

O objetivo de dividir o processamento da parte alta e parte baixa das sequências foi, além de simplificar as operações, prover escalabilidade ao *hardware* proposto, uma vez que para sequências de DNA num alfabeto de 4 letras, são utilizados 2 blocos idênticos ao apresentado, um para o vetor *MSB* e outro para o vetor *LSB*. Dessa forma, no caso de alinhamento de um alfabeto de 10 letras, haveriam 4 blocos idênticos; para 32 letras, 5 blocos idênticos e assim sucessivamente, de acordo com cálculo demonstrado. Ao longo deste capítulo, será detalhado sempre o *hardware* utilizado para o vetor relativo ao *MSB*, sabendo que o utilizado para o *LSB* é idêntico em funcionamento e estrutura.

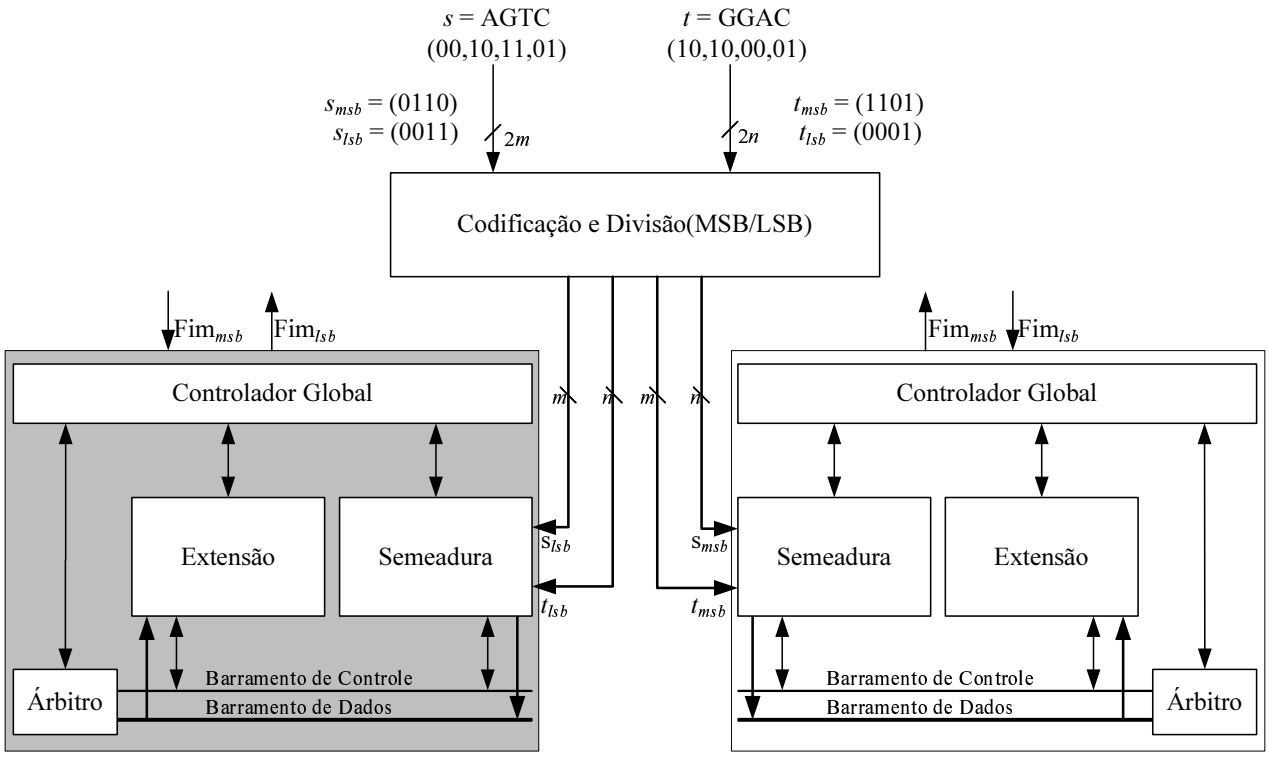


Figura 10: Codificação e adequação das sequências

4.2 Busca das sementes

Na etapa de semeadura, ocorrem diversas comparações entre o conteúdo dos registradores. Essas comparações são realizadas por lógica combinacional, baseadas em portas lógicas de 2 a w entradas, que representam pouco consumo de área. Essas estruturas de comparação foram duplicadas, de modo que a identificação de uma semente tenha duração máxima de um ciclo de *clk*. Conforme apresentado no Capítulo 3, o paralelismo estrutural implementado visa aumentar o desempenho, balanceando, sempre que possível com o consumo de recursos de *hardware*. Foi uma preocupação deste trabalho não criar uma estrutura altamente paralela, que demandasse em área de *hardware* valores extremamente dispendiosos. Desse estudo, objeto do Capítulo 6 foram criadas $(n - w + 1)$ estruturas duplicadas para a semeadura, e esse número foi escolhido por ser exatamente o número máximo de palavras de tamanho w que se pode criar a partir da sequência t. Dessa forma, foi possível atrelar o número de ciclos da semeadura ao tamanho da sequência s, pois para a sequência t, há o paralelismo, atingindo mais um objetivo proposto pelo trabalho, que é criar relação entre os requisitos de *hardware* e os parâmetros do algoritmo BLAST.

A lógica de comparação é o principal responsável pela etapa de semeadura do algoritmo BLAST. Nela o *hardware* identifica as sementes, que são pequenas regiões de alta similaridade

que serão inspecionadas nas etapas posteriores. Como descrito no Capítulo 2, a etapa de semeadura vai dividir a t, originalmente de tamanho n, em palavras de tamanho w, criando $(n - w + 1)$ palavras. Cada uma dessas palavras, se submete a comparação na sequência s. Para que as $(n - w + 1)$ palavras que sejam comparadas com todas as posições de s, o total de comparações é $n_c = (m - w + 1) \times (n - w + 1)$, onde n_c é o número de comparações necessárias.

A subdivisão das sequências em palavras menores e independentes permite que seja elaborada uma estrutura de comparação paralela. Com esse intuito, a arquitetura foi idealizada para que a comparação seja realizada entre o registrador que armazena s e as palavras de tamanho w correspondentes. Com isso, quanto maior o tamanho dos vetores armazenados em s e t, de tamanhos m e n, maior a quantidade de comparações. Outro parâmetro que vai influenciar nessa quantidade é o tamanho w das palavras.

Como apresentado, quando há um grande número de operações (n_c) a realizar, é possível realizá-las de forma paralela, isto é, todas ao mesmo tempo, duplicando o *hardware* necessário para cada função; ou realizá-la de forma sequencial, através de operação de deslocamento circular nos registradores, economizando em *hardware*, mas perdendo em tempo de execução. Por existirem dois registradores s e t, as quatro abordagens possíveis para realizar todas as comparações apresentadas no cálculo de n_c são:

- Apenas uma lógica de comparação entre s e t, testando todas as posições através de operações de deslocamento dos registradores. Dessa forma, a semeadura demandaria $(m) \times (n)$ ciclos.

- Lógica de comparação paralela para s e t, criando $(m - w + 1) \times (n - w + 1)$ blocos. Dessa forma, a semeadura demandaria apenas um ciclo.

- Lógica de comparação paralela somente para s, criando $(m-w+1)$ blocos; testando as posições através de operações de deslocamento em t. Dessa forma a semeadura demandaria n ciclos.

- Lógica de comparação paralelas somente para t, criando $(n-w+1)$ blocos; testando as posições através de operações de deslocamento em s. Dessa forma a semeadura demandaria m ciclos.

A etapa de semeadura consome pouco tempo de processamento quando comparada as demais etapas do algoritmo BLAST. As informações de tempo estão contidas no Capítulo 6, que

apresenta os resultados. Portanto, optou-se por paralelizar somente a lógica de comparação de t, realizando as comparações através de operação de deslocamento em s. A partir disso haverá $(n - w + 1)$ blocos como os da Figura 11, e será necessário aguardar m ciclos para que a toda etapa esteja concluída.

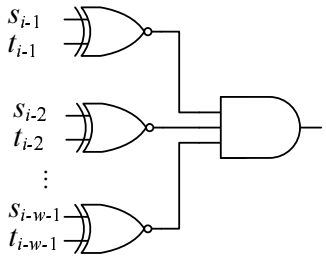


Figura 11: Comparação de um trecho w entre s e t

Como as palavras criadas, são apenas uma subdivisão lógica do registrador de t, se o *hardware* for capaz de realizar o roteamento entre o conteúdo dos registradores e a entrada correta de um bloco semelhante ao apresentado na Figura 11, não existe necessidade de componentes adicionais para armazenamento desses dados. A Figura 12 exemplifica como é feita essa subdivisão lógica. O custo está em instanciar diversos blocos para realizar a comparação entre as palavras e a s de forma paralela, além da lógica de roteamento dos sinais.

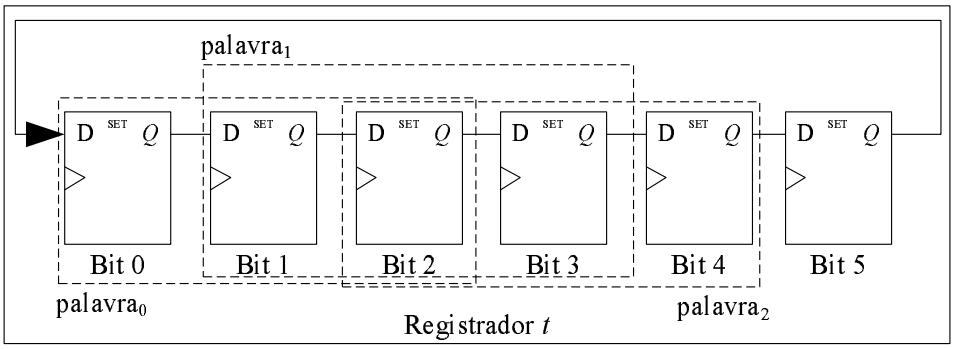


Figura 12: Mapeamento dos registradores para $w = 3$

Na Figura 13, é visto o exemplo da comparação entre uma palavra de $w = 3$ e a sequência armazenada no registrador s. São $(n - w + 1)$ palavras, que são comparadas com os w bits mais significativos do registrador s. A cada ciclo, o conteúdo de s é deslocado uma posição. Com isso, atualiza-se o contador de deslocamentos, e são feitas as comparações entre essa posição de s e todas as palavras. A cada ciclo, w bits de s são comparados com todas as $(n - w + 1)$ palavras criadas.

O circuito lógico apresentado compara todos os bits de uma vez retornando valor lógico 1 se e somente se encontrar igualdade entre a palavra e a sequência armazenada no registrador. A

lógica de comparação é composta por n portas lógicas *não ou exclusivo* de duas entradas e uma por uma porta *AND* de w entradas, sendo w o tamanho da palavra, como descrito na Figura 11. A partir do resultado obtido na lógica de comparação, o *hardware* irá considerar como uma semente, cada bloco com nível lógico 1. Como a escolha de projeto foi por paralelizar somente a lógica de comparação de t, realizando as comparações através de operação de deslocamento em s, o bloco comparador será utilizado novamente para uma nova comparação após a operação de deslocamento em s. Por isso, foi preciso desenvolver um método de armazenar as posições das comparações que resultaram em nível lógico 1, pois é necessário armazenar os índices das sementes encontradas para recuperá-las nas próximas etapas do algorítimo.

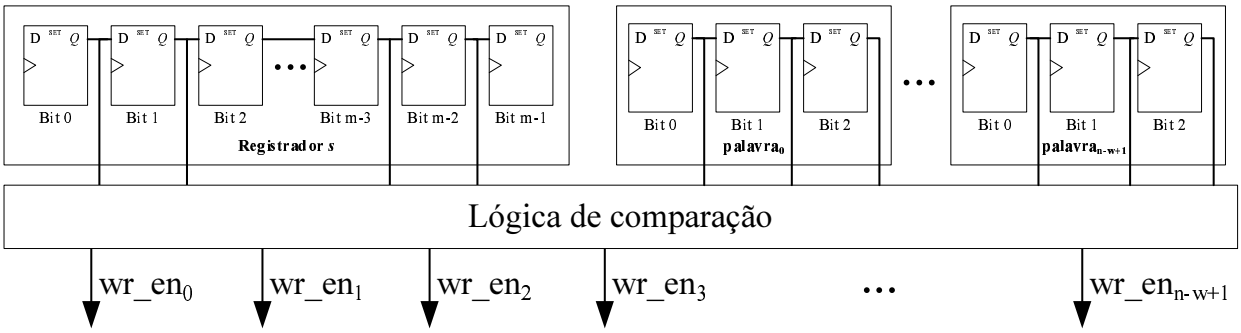


Figura 13: Comparação $t \times s$ a cada palavra

4.3 Sistema de Etiquetas

A partir do processo de comparação apresentado, o algoritmo utiliza todos os resultados de nível lógico 1, chamados de sementes, para realizar a próxima etapa de busca. Como descrito, são considerados como sementes somente trechos idênticos entre as duas sequências comparadas. Na próxima etapa, a de extensão, apresentada no Capítulo 5, objetiva-se realizar a extensão da semente, em ambas as direções visando maximizar a pontuação quando comparada à s.

Como descrito na seção anterior, a etapa de semeadura não retorna todos os resultados simultaneamente, trazendo a cada ciclos de *clk* um conjunto de até $(n - w + 1)$ sementes encontradas. Portanto a cada ciclo de *clock* a informação nos módulos de comparação será sobrescrita devido ao deslocamento feito no registrador que armazena s. Foi necessário desenvolver uma maneira de armazenar a informação de cada semente encontrada antes que a mesma seja sobrescrita. Foi elaborado um sistema de etiquetas que guarda a posição relativa dos registradores para todas as comparações feitas na lógica de comparação.

Exposto que, para realizar a extensão, é preciso recuperar as posições dos registradores quando foi encontrado uma semente, faz-se necessário saber quantas operações de deslocamento

o registrador que armazena s realizou quando foi encontrado o valor lógico 1 na lógica de comparação. Essa posição é controlada pelo controlador de etapa, através de um contador que itera a contagem a cada pulso de *clk*. É a partir dessa informação que as etiquetas serão geradas.

Para cada comparação da etapa anterior que resultar em valor lógico 1 (*hit*) é atribuída uma etiqueta, que funcionará como um ponteiro, que contem a posição dos registradores s e t, onde foi encontrada essa igualdade. Esses valores já estão disponíveis no próprio mapeamento das palavras quando foi realizada a subdivisão lógica do registrador que armazena t e estão armazenados em contadores. Portanto, a etiqueta, que funciona como ponteiro, terá dois campos, a posição relativa em s e a posição relativa em t, e será guardada em uma memória local presente dentro do própria lógica de comparação. Essa memória trabalha como uma fila, *first in first out*, com o sinal de escrita habilitado pelo valor lógico 1 oriundo do bloco de comparação.

No total, são $(n - w + 1)$ palavras sendo comparadas ao mesmo tempo na lógica de comparação. O resultado da lógica de comparação funciona como habilitador de escrita em $(n - w + 1)$ filas criadas para armazenar as sementes geradas. Cada palavra compara, a cada ciclo, o seu conteúdo com os w bits mais significativos de s. A $palavra_0$, por exemplo, sempre escreve na $fila_0$, assim como a $palavra_1$ sempre escreve na $fila_1$ e assim sucessivamente. Dessa forma é possível que uma fila tenha muitas sementes armazenadas, enquanto outra fila não tem nenhuma semente armazenada.

Todas as sementes terão inicialmente o tamanho w, logo esse valor não precisa ser armazenado em uma etiqueta. Com as informações de posição armazenadas em etiquetas, guardadas em filas, é possível recuperar essas informações a qualquer tempo, e estender as sementes em ambas as direções buscando maximizar a pontuação quando as próximas etapas estiverem aptas a receber tal informação.

Cada uma das etiquetas geradas, até $(n-w+1)$ por ciclos, é tratada por processadores de extensão, que tem a função de, a partir da etiqueta recebida, recuperar a posição dos registradores; buscar o conteúdo das próximas posições; comparar esses conteúdos; atribuir uma pontuação para a comparação; atualizar a etiqueta ou descartá-la em caso de diminuição da pontuação. Esse processo é o mesmo apresentado no Capítulo 2, onde foi descrito o algoritmo de extensão, e será descrito em *hardware* no Capítulo 5.

Tabela 13: Controle da Etapa de Semeadura

Teste Lógico	Habilita *en*=1 em:
Se N_d = m-1, $en_s = 0$; senão en_c= 1	Contador de Deslocamentos
Se N_c > m-1, $en_s = 0$; senão en_s=1	Registrador s

4.4 Controle da etapa de semeadura

O controle da etapa de semeadura consiste em prover os sinais necessários para o deslocamento dos registradores, o chaveamento da comparação e garantir que as etiquetas estejam armazenadas nas filas quando encontrada uma semente. A arquitetura de *hardware* proposta necessita de m ciclos para encontrar todas as sementes possíveis. Portanto, existe um contator de ciclos e um contador de deslocamentos do registrador. Esses valores são utilizados como chave na lógica de roteamento para comparação das sementes, e todas as vezes que se precisa recuperar um conteúdo do registrador s, pois devido ao deslocamento é necessário um índice para encontrar as posições desejadas.

Para gerenciar esta etapa e o deslocamento do registrador s, existe um circuito baseado em comparadores que recebe como sinais de entrada o contador de deslocamento, e fornece como sinais de saída en_c e en_s, que são os sinais que vão habilitar e desabilitar, em conjunto com o controle geral, o carregamento ou deslocamento do registrador s.

Os contadores armazenam o número de deslocamentos realizado pelo registrador, e é usado para calcular os índices. Existe uma variável que indica a quantidade de operações de deslocamento foram realizadas no registrados s. Na inicialização do sistema, o sinal de saída en_c assume valor lógico 1. Quando o contador indica que foram realizadas $(m-1)$ operações de deslocamento no registrado s o sinal de saída en_c assume valor 0. Quando o sinal de saída en_c recebe 0, ele desabilita o *clk* do contador de deslocamento, de forma que o índice não seja mais incrementado.

Outro sinal de saída gerado pelo controle da etapa de semeadura é o sinal en_s. Ele é assume valor lógico 1 na inicialização do sistema e recebe valor 0 quando o contador de ciclos se torna maior que $(m-1)$. O sinal de saída en_s em valor lógico 0 desabilita o *clock* do registrador s e indica ao sistema de controle geral que todas as sementes já foram encontradas. A Tabela 13 ilustra como funciona o controle da etapa de semeadura baseada na contagem de deslocamento e ciclos, enquanto o Algoritmo 10 mostra como essa informação é utilizada na geração de etiquetas, onde N_d representa o número de deslocamentos e N_c o número de ciclos, i_s é o índice de s e j_t é o índice de t.

O *clock* do registrador t é de responsabilidade do controlador global, através do sinal

de saída t_{en}. Apesar do restante do circuito de comparação ser totalmente combinacional, conforme apresentado na Figura 13, na criação das etiquetas, o valor armazenado no contador de ciclos é utilizado para calcular o índice a ser armazenado.

Algoritmo 10 Controle baseado em valor do contador de deslocamentos

1: **Para** $i = 0 \rightarrow (n - w)$ **Faça**
2: $\quad i_s \leftarrow (m - 1 - N_d)$
3: $\quad j_t \leftarrow (i + w - 1)$
4: **Fim Para**

Com as etiquetas armazenadas nas memórias do padrão *first in first out*, e após m operações de deslocamento no registrador s, a etapa de semeadura é concluída. A informação de término da etapa vai até o controlador global, que irá tratar especificamente da etapa de extensão.

4.5 Simulações

Esta Seção apresenta janelas de simulação da arquitetura que visam demonstrar a atuação dos principais sinais de controle para a execução do algoritmo. O simulador utilizado foi o *ModelSim*, na linguagem VHDL. A Figura 14 mostra o início do processamento. Quando recebe o sinal *Iniciar* o controlador geral entra no estado de inicialização do *hardware*. Envia o sinal de *reset* ao contador que conta o número de deslocamentos do registrador s; carrega os registradores com as sequências s e t, respectivamente, controlando através dos sinais s_{enable} e t_{enable} os descolamento desses registradores.

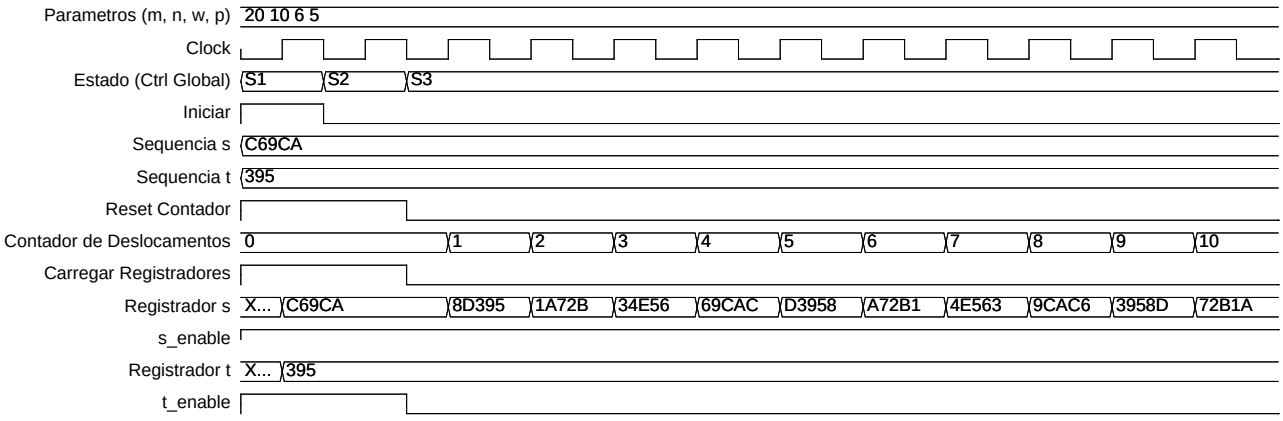


Figura 14: Inicialização dos Componentes

A partir dos parâmetros do algoritmo m, n, w, e do descolamento do registrador s a lógica de comparação faz o roteamento e busca as sementes nas sequências. Criam-se $(n - w + 1)$ comparadores para buscar as sementes comparando as palavras de tamanho w, criadas a partir

de t, e os w bits da sequência s, que são apresentados a cada ciclo através do deslocamento do registrador. Na Figura 15 verifica-se a comparação, ciclo-ciclo, entre os w bits mais significativos de s e todas as palavras de w bits criadas a partir de t. Além desses resultados intermediários, são apresentadas as palavras e os w bits mais significativos do registrador s.

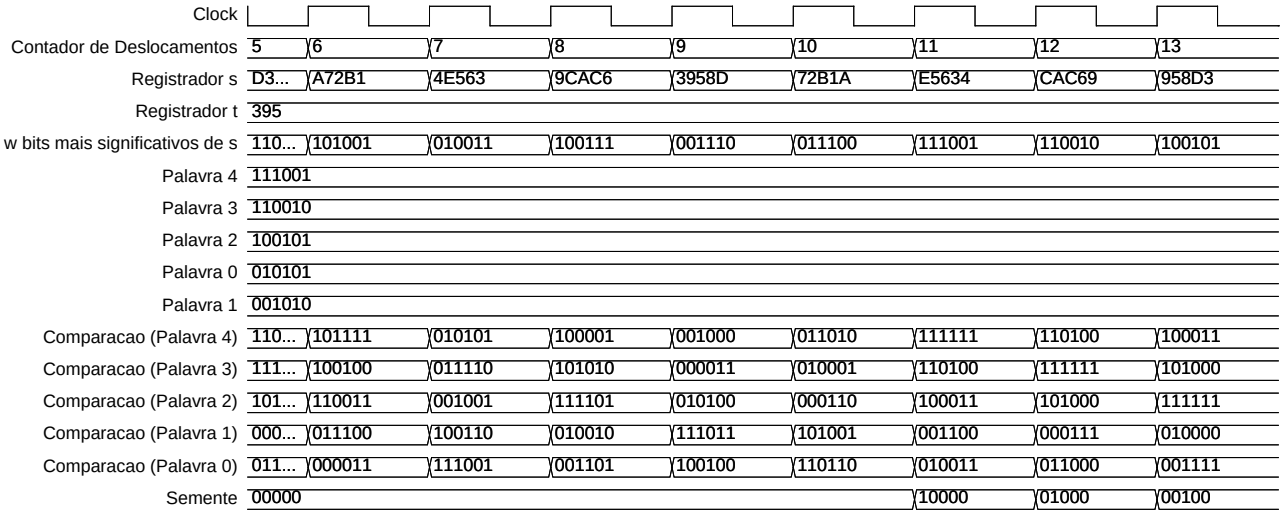


Figura 15: Detalhamento da lógica de comparação

O sinal *Semente* só será ativo, quando todos os w bits da comparação forem ativos, como pode-se verificar na Figura 16 que mostra também, que para cada resultado, está associado um valor de etiqueta. Esse valor de etiqueta é calculado a partir do número de deslocamento, e do índice da comparação. Por exemplo, para a $palavra_4$, o índice será 4.

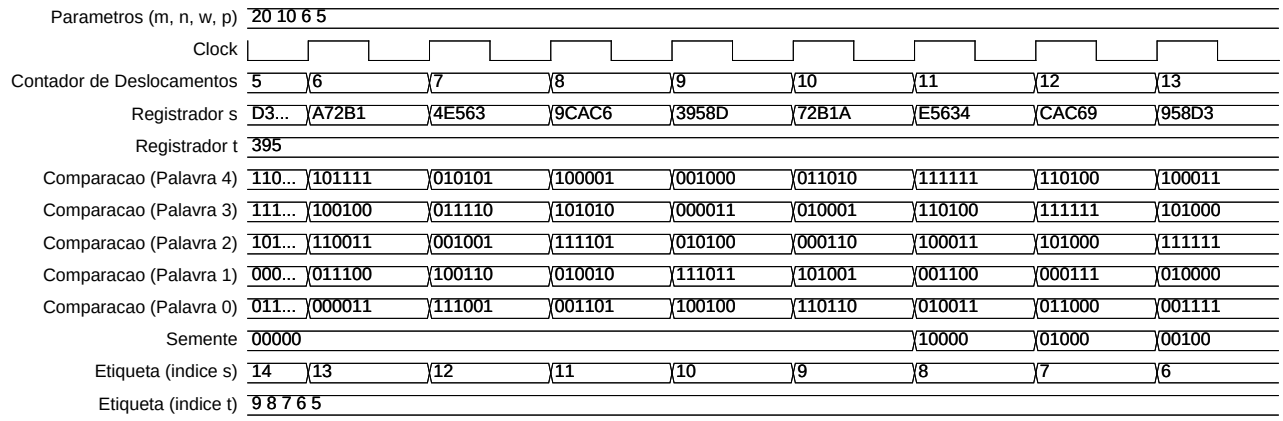


Figura 16: Comparação entre palavras e os w MSB de s

Como o cálculo das etiquetas é feito dinamicamente, a cada ciclo, é preciso armazenar as etiquetas que serão utilizadas posteriormente. Cada par de etiquetas é armazenado nas $(n - w + 1)$ *fifos*, de acordo com o índice. Por exemplo, as sementes encontradas na *Palavra 3*, serão armazenadas na *fifo* de índice 3. O sinal *semente* é utilizado como sinal de *push* de

uma fila, onde o par de etiquetas é o valor de entrada e saída. Na Figura 17 pode-se observar as sementes encontradas dando os sinais de *push* para a inserção de etiquetas nas filas. Essa Figura mostra que a partir da gravação de um etiqueta, o controlador global muda de estado, para iniciar a extensão. Assim como descrito, não é necessário que todas as etiquetas estejam prontas para se iniciar a extensão. Os detalhes da extensão são abordados no próximo capítulo.

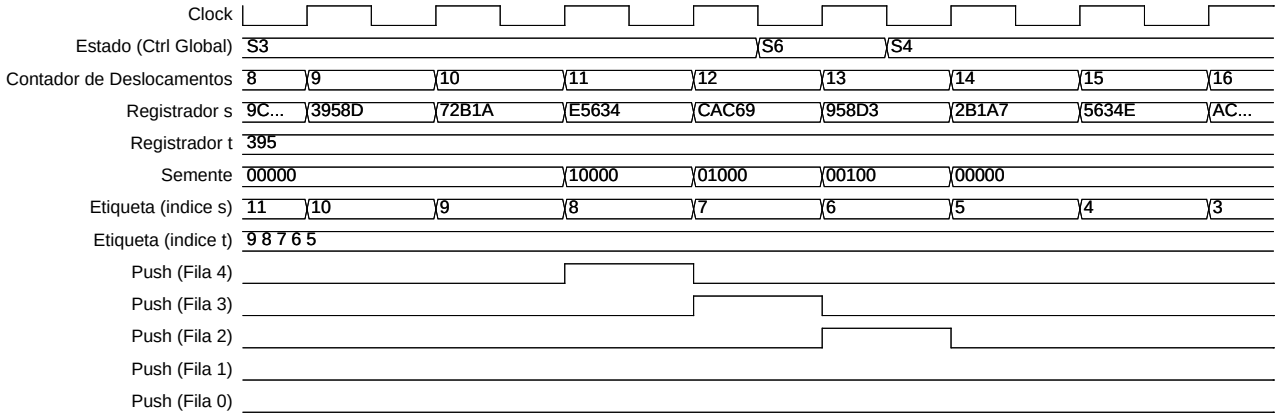


Figura 17: Sementes realizam operação de *push* nas filas

No exemplo simulado, existem 5 filas do padrão *first in first out*. Quando ocorre uma requisição de algum processador, o árbitro seleciona uma das filas para utilizar o barramento, conforme apresentado no Capítulo 3. A Figura 18 mostra a concessão do barramento à fila escolhida, através dos critérios do árbitro; e os pares processador/filas, escolhidos para ocupar o barramento. É possível observar que para múltiplas requisições, o árbitro processa uma de cada vez, e a medida que o barramento é liberado, outras filas e processadores vão recebendo a concessão.

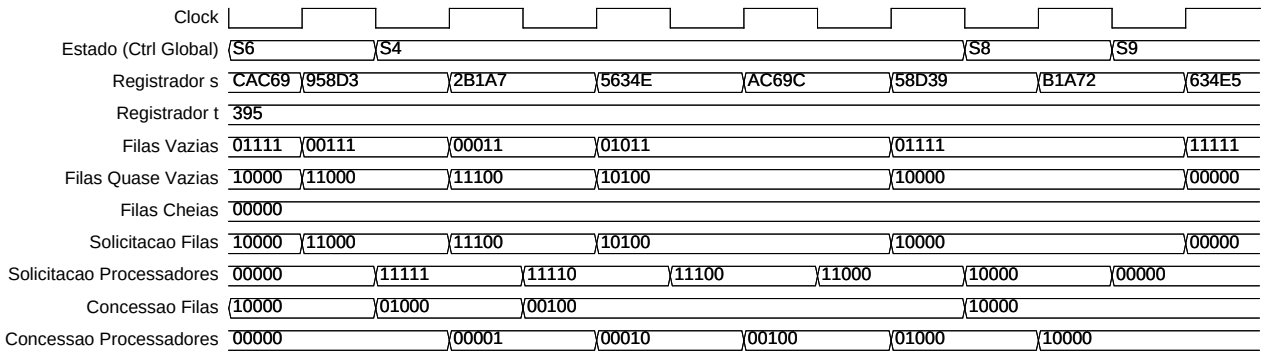


Figura 18: Escolha da fila prioritária.

Após realizar m operações de deslocamento no registrador s, todas as possibilidades de sementes foram testadas, e estão ou armazenadas nas filas, ou já estão na extensão. Portanto,

quando o contador atinge a contagem de m, o controlador global apenas desabilita o descolamento do registrador s, através do sinal s_{enable}, como apresentado na Figura 19. Esse sinal também serve para inibir a lógica de comparação, evitando novas sementes.

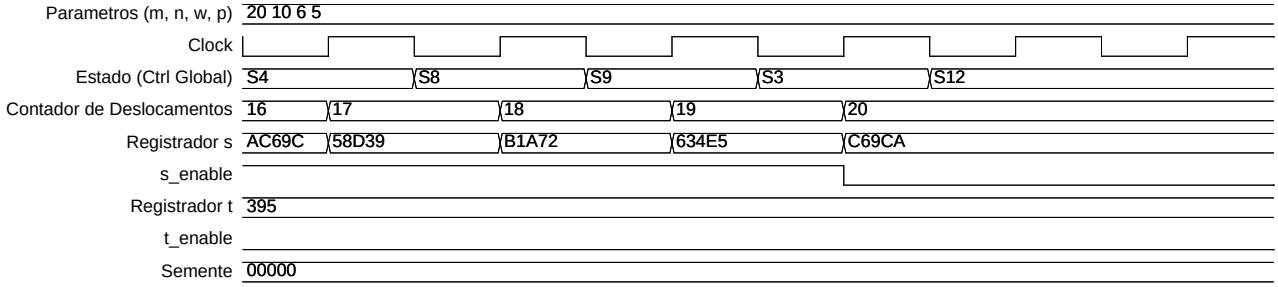


Figura 19: Fim da Semeadura

4.6 Considerações Finais do Capítulo

Neste capítulo foi apresentado a arquitetura da semeadura. Foi detalhado como se dá a adequação das entradas, a divisão das sequências, a busca das sementes, o processo de comparação e a inserção nas filas. Algumas das escolhas de projeto mencionadas, como o número de estruturas paralelas para a comparação está justificada no Capítulo 6, onde são apresentados os resultados de simulação e síntese. No próximo capítulo é detalhado o *hardware* que executa a extensão.

Capítulo 5

EXTENSÃO

A ETAPA de extensão é responsável por tentar aumentar a pontuação de cada semente encontrada na etapa de semeadura. As sementes encontradas nessa etapa anterior, estão armazenadas em memórias, e podem ser lidas através do barramento de dados. Como descrito no capítulo anterior, há um número grande de filas compartilhando esse barramento. Cabe ao árbitro, descrito no Capítulo 2, a função de controlar esse barramento de dados.

A partir da semente lida, a etapa de extensão realiza uma série de avaliações para determinar se os índices armazenados nas etiquetas representam o melhor alinhamento possível, ou se ainda podem ser melhorados. Quem realiza essas avaliações é o componente denominado processador de extensão. Por se tratar de uma proposta de arquitetura paralela, o *hardware* descrito pode trabalhar com até p processadores de extensão, sendo que p é um valor definido pelo usuário, de acordo com a disponibilidade de recursos de *hardware* da placa em que será implementada o BLAST.

O conceito de paralelismo é extremamente abrangente para a extensão. Para cada um dos p processadores, há acoplado um controlador, de forma que, apesar de realizar a mesma tarefa, os processadores trabalham de forma independente. Como demonstrado na Figura 20, esses p processadores calculam os resultados e escrevem na memória que armazena os resultados, no formato $(i,\ j,\ d)$.

Neste capítulo será detalhado o *hardware* criado para executar a extensão. Na Seção 5.1 é detalhado o funcionamento do processador de extensão e na Seção 5.2 é apresentado o controlador de etapa.

5.1 Processador de extensão

O algoritmo BLAST visa alinhar 2 sequências de DNA. Estando as duas sequências codificadas e armazenadas em registradores s e t, a primeira etapa do algoritmo é dividir a sequência t

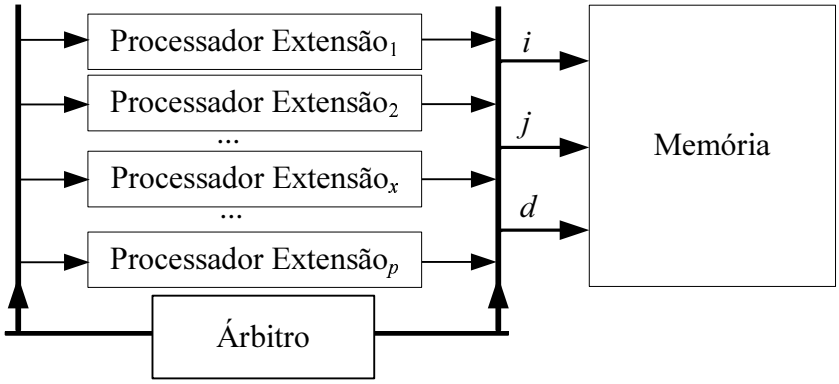


Figura 20: Processadores de extensão paralelos escrevendo em memória

em palavras menores de tamanho w. Após realizar essa divisão, o algoritmo busca pequenos trechos idênticos, comparando pedaços de w bits entre s e t. Para cada um desses trechos idênticos encontrados, chamados de sementes, é gerado uma etiqueta, que contém a posição dessa semente nas sequências s e t. É a partir dessa etiquetas que é realizada a etapa de extensão.

Cada etiqueta é armazenada no formato $(i,\ j,\ d)$. O índice i corresponde a posição na sequência s onde foi encontrada a igualdade. O índice j corresponde a posição na sequência t onde foi encontrada a igualdade. O índice d corresponde ao número de bits de igualdade que cada semente tem, sendo que w é o valor inicial.

Como descrito no Capítulo 2, realizar a extensão consiste em recuperar o valor das etiquetas, e analisar se aumentando o valor de w, as sequências continuam sendo semelhantes. Essa comparação se dá tanto para a esquerda, quanto para a direita. Por exemplo, supondo que a etiqueta $(10, 5, 4)$ é recebida pelo processador de extensão. Isso indica que a partir das posições s_{10} e t_5, os próximos 4 bits serão idênticos, resultando nos trechos (s_{10}, s_9, s_8, s_7) e (t_5, t_4, t_3, t_2).

Para realizar a extensão à esquerda, o processador irá recuperar o conteúdo próxima posição das duas sequências: s_{11} e t_6, compará-los, e caso o valor seja positivo, atualizar os índices. O mesmo deve ser feito para o outro lado da sequência, ou seja, para a extensão à direita, serão recuperados as posições s_6 e t_1.

Para sequências muito grandes, a extensão em um sentido, direita ou esquerda, pode levar muito mais tempo do que em outro sentido. Visando otimizar a utilização de recursos nesse cenário, os processadores de extensão tem componentes específicos para estender a semente à direita e à esquerda trabalhando paralelamente, porém se comunicando. A função de cada componente é detalhada na Figura 21, onde é apresentada a arquitetura de um processador de

extensão.

Como descrito, um processador, recebe a etiqueta vinda de umas das filas. Essa etiqueta é carregada em contadores. A saída desses contadores é monitorada, gerando as interrupções através de lógica combinacional. No fim do processo, o valor dos contadores é armazenado em uma memória de saída. Toda a lógica do processador de extensão é baseada em comparações e somas, cujos resultados incrementam os contadores de $(i,\ j,\ d)$, e consequentemente modificam os índices que são utilizados para recuperar novas posições, como visto na Figura 21.

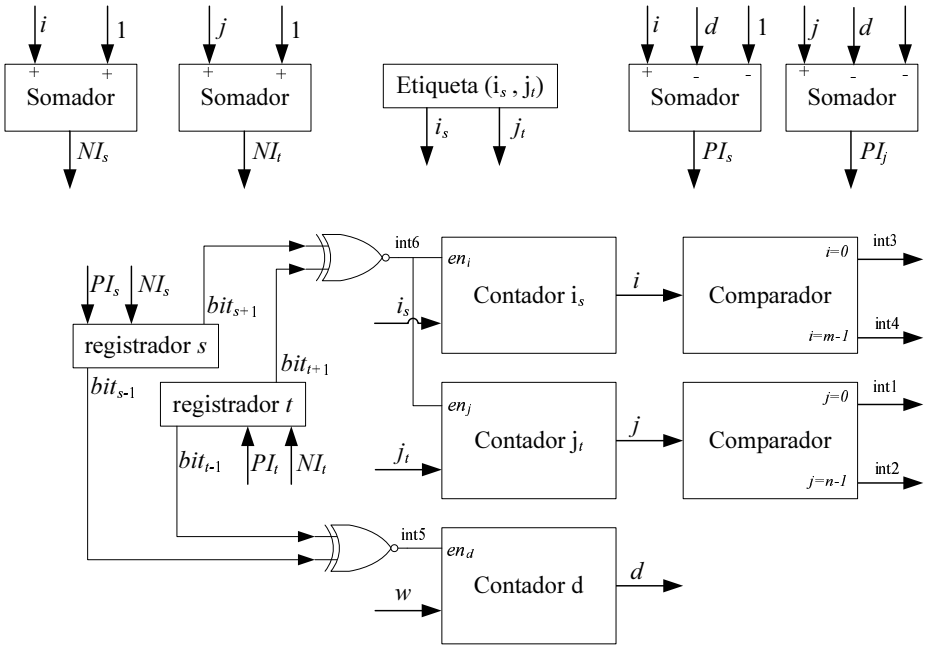


Figura 21: Arquitetura do processador de extensão

Ao receber uma etiqueta, o processador carrega seus contadores com os valores contidos nessa etiqueta. Após isso, testa o valor dos contadores através de um circuito lógico-aritmético, que calcula o índice das posições que precisam ser recuperadas nos registradores. Após calcular esses índices, o processador solicita, através do barramento de controle, o bit contido em cada uma dessas posições. O funcionamento é idêntico a uma memória, onde é calculado um endereço, e o dado contido naquele endereço é retornado.

Após recuperar os bits de s e j, o processador testa se há igualdade ou não, utilizando o circuito combinacional mostrado na Figura 21. Com esse resultado, o processador decide se continua ou se para a extensão. Em caso de igualdade, o processador deve incrementar o contador e atualizar os índices. Quando encontra uma diferença, o processador armazena o valor presente dos contadores, i, j e d, em memória e informa ao controlador que está livre. A extensão para a esquerda, trabalha com os contadores i e j. De forma análoga, a extensão

Tabela 14: Lista de interrupções do processador

Interrupção	Situação
$Int(1)$	j_t atingiu o limite inferior:(0, ou $w-1$)
$Int(2)$	j_t atingiu o limite superior:$(n-1)$
$Int(3)$	i_s atingiu o limite inferior:(0, ou $w-1$)
$Int(4)$	i_s atingiu o limite superior:$(m-1)$
$Int(5)$	s_{i-1} diferente de t_{j-1}
$Int(6)$	s_{i+1} diferente de t_{j+1}

para a direita, procura similaridades a direita, incrementando o contador d e atualizando os índices em caso de igualdade.

Independentemente do lado que a extensão estiver sendo feita, direita ou esquerda, é necessário que o processador saiba quando deve parar de realizar a extensão. Numa sequência de 18 bits, quando o processador atingir a posição 18, ele deve receber um sinal indicando o fim da repetição. De forma análoga, na extensão para o outro lado, quando estiver na posição 0, ele deve reconhecer que não há mais o que fazer com aquela sequência. Para informar ao processador e aos outros componentes que essas posições foram atingidas, elaborou-se um mecanismo de interrupções.

As operações realizadas pelo processador têm limites superiores e inferiores, que indicam quando a posição mais significativa é atingida, e quando a posição menos significativa é atingida. Dessa forma, quando a extensão atinge a primeira ou a última posição possível de um registrador, é gerado um sinal de interrupção. As interrupções dos processadores são apresentadas na Tabela 14. Além das posições limites, ocorre também uma interrupção quando os bits carregados são diferentes. Cada interrupção tem significado local, apenas para o processador p em questão e o seu respectivo controlador de etapa. O objetivo desses sinais de interrupção é informar ao controlador de etapa que aquele processador não pôde mais realizar a extensão em um dos sentidos, ou em ambos os sentidos. Dessa forma, o controlador de etapa daquele processador, identificará qual próxima ação deverá realizar.

Dentro do processador de extensão, os índices e as etiquetas são atualizadas, utilizando estruturas de comparação semelhantes a da etapa de semeadura e contadores para armazenar a pontuação. Quando da opção por separar a parte alta e a parte baixa das sequências, o objetivo era também simplificar o *hardware*, o que realmente aconteceu, pois as estruturas para comparar um bit se resumem, na maioria das vezes, a portas lógicas.

Quando atingir uma pontuação que não puder ser melhorada, os valores dos contadores serão armazenados em uma memória que constitui a saída da extensão para avaliação da

etapa subsequente. Com isso, está concluído o ciclos de *hardware* que executou o algoritmo BLAST, pois a etapa de avaliação, que classifica estatisticamente cada um dos alinhamentos encontrados, não foi implementada em *hardware*, podendo ser realizada em *software* sobre todo o conteúdo da memória. Para otimizar a escrita dos resultados em memória, foi construída uma memória de p vias, de forma que todos os processadores possam escrever na memória ao mesmo tempo.

Para cada um dos p processadores, há um controlador de etapa específico. Na Seção 5.2, é detalhado o controlador de etapa, e na Seção 5.3 são apresentados resultados de simulação que validam o modelo do processador e seu respectivo controlador.

5.2 Controlador de etapa de extensão

O objetivo desse projeto é implementar uma arquitetura paralela para executar o algoritmo BLAST. Como descrito no Capítulo 3, a extensão foi paralelizada em p estruturas que vão conter, cada uma, um processador de extensão e um controlador de etapa.

Controlar a etapa de extensão é basicamente gerenciar as tarefas dos processadores de extensão, pois os barramentos são controlados pelo árbitro e a escrita em memória é gerida pelo controlador global. O compromisso do controlador da etapa de extensão é realizar a interface com os demais controladores e com o árbitro, para sequenciar as tarefas de modo a não gerar conflitos no *hardware*.

Outra tarefa importante do controlador da etapa de extensão é a comunicação entre o *hardware* responsável pelo processamento dos bits menos significativos e o dos bits mais significativos das sequências. Conforme apresentado na Seção 4.1, existe um *hardware* específico para a parte alta e baixa, respectivamente *MSB* e *LSB*. Os controladores da etapa de extensão de cada um desses blocos se comunicam através das interrupções, de modo que quando é encontrada uma diferença na comparação, em qualquer um dos dois blocos, seja *MSB* ou *LSB*, o processamento daquela etiqueta é interrompido em ambos os blocos. Desse forma foi garantido o perfeito sincronismo entre os blocos, e a integridade da saída dos processadores, que é armazenada em memória. A Figura 22 apresenta a via de dados de um controlador da etapa de extensão com os sinais de entrada e saída que são descritos a seguir, bem como a interação entre processador de extensão, árbitro e o controlador de etapa.

Com a possibilidade de se utilizar vários processadores de extensão, cada um deles será gerenciado por um controlador individual, e todos eles se reportam ao controlador global, através de seus sinais de controle. No controlador global, esses sinais individuais são recebidos

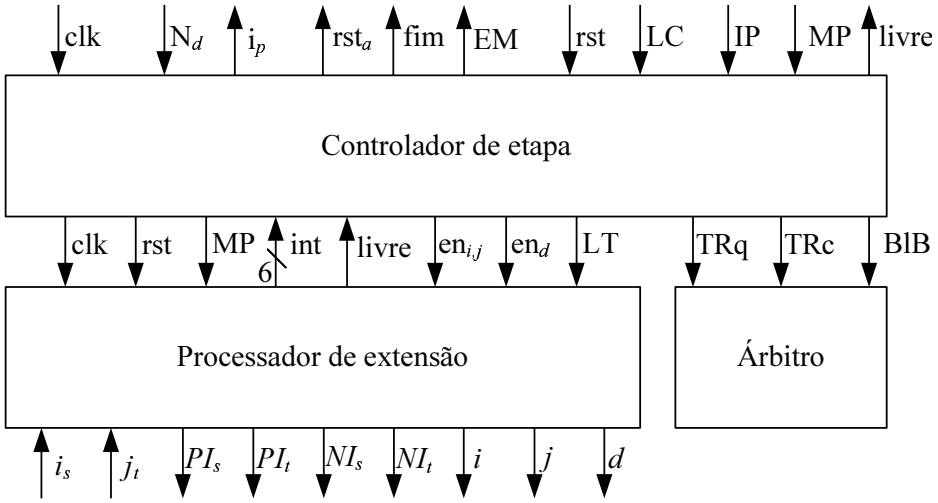


Figura 22: Via de dados controlador da etapa de extensão

no formato de um vetor de bits, sendo que cada bit do vetor corresponde a um conjunto processador/controlador de extensão. Os sinais de entrada de um controlador da etapa de extensão são os seguintes:

- *clk*: Sinal de 1 bit, responsável pelo sincronismo do sistema;
- *rst*: Sinal de 1 bit, que reinicia o processador de extensão;
- *init*: Sinal de 1 bit, que quando recebido, indica que o processador deve iniciar o processamento de uma etiqueta;
- *Int*: Vetor de 6 bits, que contém a solicitação de interrupção gerada por outros componentes do processador, sendo cada bit um tipo de interrupção;
- N_d: Indica quantas operações de deslocamento foram realizadas no registrador s. Utilizada para calcular os índices para divisão das palavras;
- *Pos*: Ponteiro que indica a primeira posição do registrador s. Serve como valor de referência para calcular os índices;
- i_s *e* j_t: Recebe o valor contido na etiqueta a ser estendida. Indicam a posição relativa em s e t respectivamente;
- *d*: Recebe o valor contido na etiqueta a ser estendida. Indica a quantidade de posições existe no resultado encontrado;
- *MP*: Sinal de 1 bit, indica que o processador recebeu a concessão do árbitro.

O controlador da etapa de extensão, utiliza os sinais de entrada descritos para prover os sinais de controle necessários ao pleno funcionamento dos processadores de extensão. Nesse controlador, além dos sinais binários de controle, também são calculados variáveis que trabalham como os índices do sistema de comparação que precisa recuperar o conteúdo dos registradores. Abaixo, são descritos os sinais de saída do controlador da etapa de extensão

- PI_s *e* PI_t: Indicam em qual posições dos registradores s e t, respectivamente, estão os bits a serem comparados para a extensão à esquerda.
- NI_s *e* NI_t: Indica em qual posição dos registradores s e t, respectivamente, estão os próximos bits a serem comparados, para a extensão à direita.
- *Livre*: Sinal de 1 bit, indica que o processador em questão está disponível.
- $en_{i,j}$: Sinal de 1 bit, responsável por habilitar e desabilitar o *clk* dos contadores que processam as etiquetas.
- en_d: Sinal de 1 bit, responsável por habilitar e desabilitar o *clk* do contadores que incrementa a contagem de posições idênticas entre as duas sequências.
- *LT*: Sinal de 1 bit, enviado aos contadores do processador para que sejam carregados com os valores contidos em i_s, j_t e d.
- *TRq*: Sinal de 1 bit, é ativo em 1 quando o processador solicita ao árbitro uma nova etiqueta.
- *TRc*: Sinal de 1 bit, é ativo em 1 para confirmar ao árbitro que a etiqueta foi recebida.
- *FIM*: Sinal de 1 bit, é ativo em 1 para informar ao controlador global o término da extensão de uma etiqueta.
- *BlB*: Sinal de 1 bit, é ativo em 1 para bloquear o barramento de transmissão de etiquetas, pois o mesmo está sendo utilizado por esse processador de extensão.
- *EM*: Sinal de 1 bit, que indica ao controlador global que o processador de extensão precisa de mais ciclos para carregar os valores fornecidos.

Algumas das entradas e saídas dos controladores são no formato de números naturais. Elas são utilizadas para realizar o cálculo da posição dos bits a serem comparados nos respectivos registradores. Isso é necessário, pois quando ocorre a operação de deslocamento em s, a

posição de um bit no registrador muda. Consequentemente, para atualizar os índices PI_s, PI_t, NI_s e NI_t, que trabalham como ponteiros no mapeamento dos dados, o controlador precisa das informações para atualizar esses ponteiros.

O controlador da etapa de extensão também foi criado como uma máquina de estados. Dessa forma, os sinais de interrupção, em conjunto com os sinais de entrada são monitorados, para prover os sinais de saída apresentados, na sequência de transição mostrada na Figura 23. As interrupções *Int(1)*, *Int(3)* e *Int(5)* são comparadas em uma porta lógica OU, e a saída dessa porta lógica, *ant*, é monitorada pela máquina da estados. As interrupções *Int(2)*, *Int(4)* e *Int(6)* também são comparadas numa porta lógica OU, e a saída dessa porta lógica, *prox*, é monitorada pela máquina de estados.

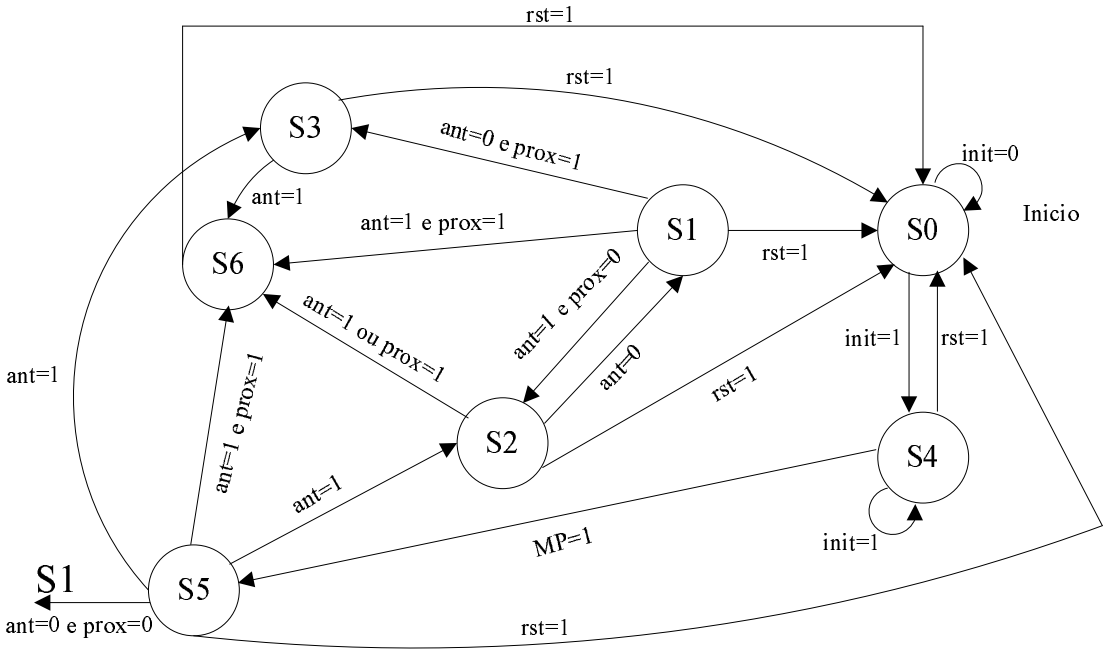


Figura 23: Diagrama de transição de estados do controlador da etapa de extensão

- S0: Inicializa o processador. Testa o sinal de entrada *init*.
- S1: Estado onde ocorre a extensão tanto para direita quanto para a esquerda, pois os sinais de saída $en_{i,j}$ e en_d estão ativos em 1. Testa os sinais de entrada *ant*, *prox* e *rst*.
- S2: Estado onde somente ocorre a extensão para direita, estando portanto somente o sinal de saída en_d ativo em 1. Testa os sinais de entrada *ant*, *prox* e *rst*.
- S3: Estado onde somente ocorre a extensão para esquerda, estando portanto somente o sinal de saída $en_{i,j}$ ativo em 1. Testa os sinais de entrada *ant*, *prox* e *rst*.

- S4: Estado que envia ao controlador global e ao árbitro a requisição do processador por uma nova etiqueta, pois o mesmo está ocioso. Esta requisição é feita através dos sinais de saída *BlB* e *TRq*. Testa os sinais de entrada *init*, *MP* e *rst*.

- S5: Estado para carregar as etiquetas nos contadores através do sinal de saída *LT*. O sinal de saída *TRc* é ativo em 1 para informar que a etiqueta foi recebida com sucesso, e o sinal *BlB* é desabilitado, para que o barramento seja utilizado para outra finalidade. Testa os sinais de entrada *ant*, *prox* e *rst*.

- S6: Estado que informa que a etiqueta não pode mais ser estendida. Aguarda o sinal de entrada *rst*.

5.3 Simulações

Esta Seção apresenta as janelas de simulação da extensão. As principais tarefas são apresentadas para demonstrar a execução do algoritmo, a interação entre os processadores de extensão e o árbitro, e o cálculo dos resultados de cada alinhamento. A Figura 24 apresenta o *hardware* aguardando alguma semente ser encontrada pela semeadura, para então iniciar a extensão. Quando, a partir do ciclo 11, uma semente é identificada ocorre uma operação de *push* na fila, gerada pela lógica combinacional. Os valores das etiquetas desse ciclo são armazenados para posterior extensão. Nesta figura, é possível verificar que a medida que as etiquetas são inseridas nas filas, o árbitro atualiza a fila prioritária, baseado nos critérios apresentados no Capítulo 3.

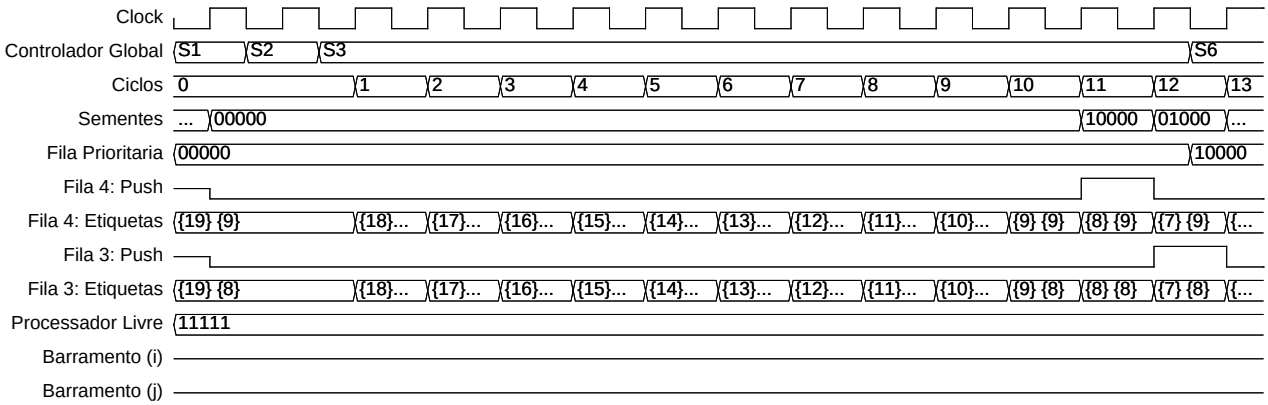


Figura 24: Extensão aguardando sementes

Quando há uma etiqueta armazenada em fila, o controlador global muda de estado, transmitindo a informação aos processadores de extensão. Na primeira vez, como todos os processadores estão ociosos, todos eles solicitam a etiqueta, e o árbitro escolhe através do algoritmo *round-robin*, a qual processador dará a concessão. Na Figura 25 é apresentado os

momentos em que as requisições chegam ao árbitro, e a concessão enviada pelo árbitro aos processadores, informando qual foi o escolhido. Nessa figura também observa-se o fechamento da rotina de extensão, que se inicia com a identificação da semente no ciclo 12, a requisição do processador no ciclo 13, a concessão pelo árbitro no ciclo 14, o barramento de dados entregando as etiquetas ao processador, que logo informa através do sinal *livre* que já não está mais ocioso. Na figura também é possível verificar que o processador escolhido envia uma confirmação ao árbitro, e que só após essa confirmação o árbitro libera o barramento.

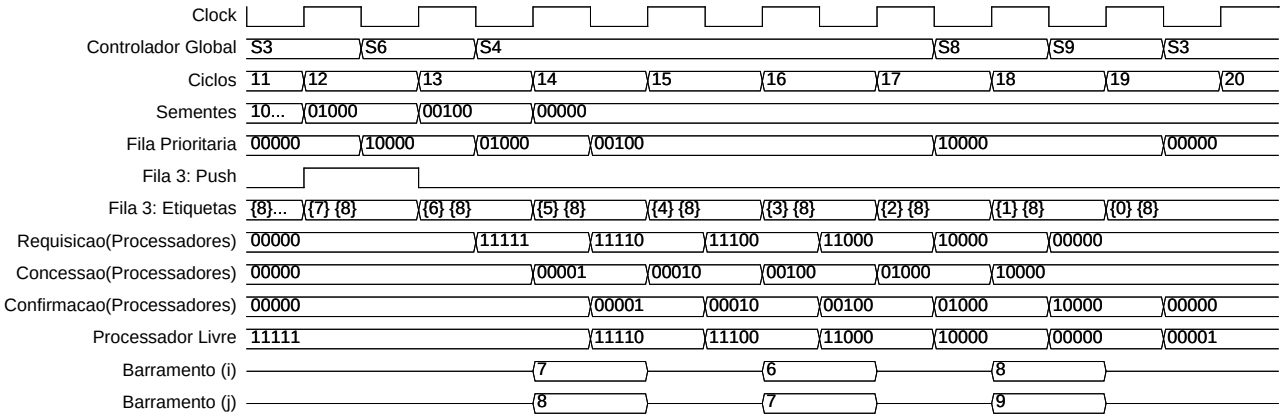


Figura 25: Escolha do processador de extensão

Na Figura 26 é possível verificar cada processador de extensão aguardando a concessão do árbitro para iniciar o trabalho, onde é visto o estado de cada um de seus respectivos controladores e a atualização das interrupções que são geradas a todo tempo, por se tratarem de circuitos combinacionais.

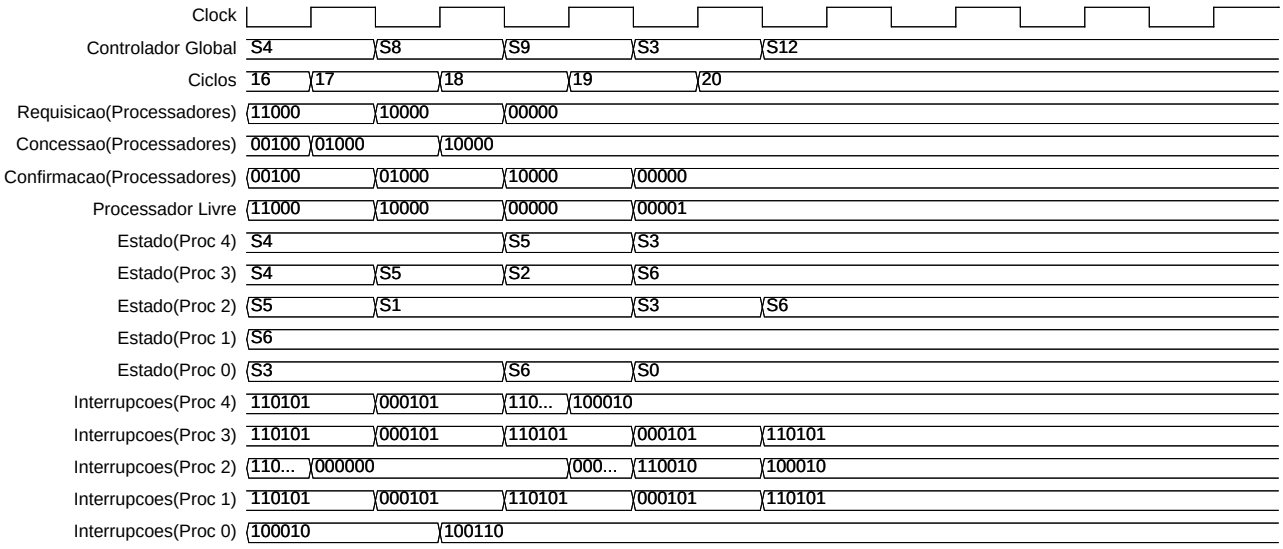


Figura 26: Estado dos processadores de extensão

Após receber a etiqueta, o processador de extensão realizará as operações na tentativa de melhorar a pontuação, conforme descrito na Seção 5.1. Dando continuidade ao exemplo acima, na Figura 27 o processador de extensão 0 recebe a concessão do árbitro, recebe as etiquetas através do barramento de dados, e começa a tentar maximizar a pontuação. Como o registrador s sofreu alguns deslocamentos, os índices que trabalham como ponteiros refletem a nova posição em s dos bits que o processador precisar comparar. Cada um dos p processadores tem seu conjunto de interrupções e índices independente. É possível verificar na figura que usa o processador 0 como exemplo, que a cada ciclo, os índices são atualizados, e os bits dessas posições recebidos e comparados. Na Figura 27 também observa-se que quando é encontrada uma diferença nos bits da extensão para a esquerda, no ciclo 16, a interrupção faz com que o controlador mude de estado e o processador continua estendendo essa etiqueta apenas para a direita. No fim, o valor dos contadores na forma $(7, 8, 9)$ é gravado na memória, representado o resultado encontrado para um alinhamento$(s,\ t)$ e seu deslocamento.

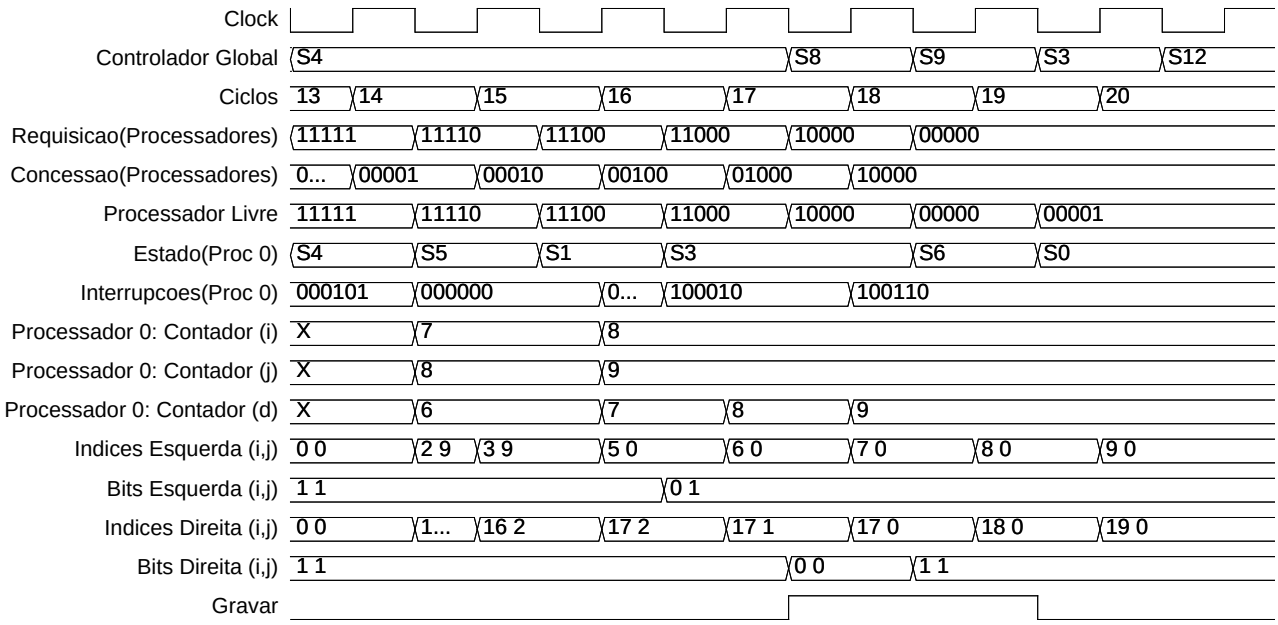


Figura 27: Índices atualizados e término da extensão

A medida que os processadores vão sendo selecionados pelo árbitro para realizar a extensão das etiquetas, as filas vão esvaziando. Quando todas as filas estiverem vazias, todos os processadores estiverem ociosos e caso o registrador s já tenha realizado m operações de deslocamento, o controlador global ativa o sinal *concluído*, e desabilita o *clock* de todo o *hardware*, terminando assim o alinhamento das sequências. Quando o sinal *concluído* é acionado, significa que as possíveis soluções de alinhamento, estarão armazenados em memória.

Como descrito nos Capítulos 2 e 3, a etapa de avaliação, que complementa o algoritmo BLAST realizando uma análise estatística dos alinhamentos encontrados não foi implementada em *hardware*.

5.4 Considerações Finais do Capítulo

Neste capítulo foi apresentado a arquitetura da extensão. Foi detalhado como ocorre o processamento das etiquetas, dentro de cada um dos p processadores de extensão. Para cada processador instanciado, está acoplado um controlador de etapa, que também foi detalhado nesse capítulo.

Apresentou-se ainda, neste capítulo, resultados de simulação, para validar o modelo de arquitetura proposta. No Capítulo 6 são demonstrados resultados obtidos através da síntese da arquitetura exposta nos Capítulos 3, 4 e 5, e são elencados e demonstrados os objetivos propostos e atingidos por esse trabalho.

Capítulo 6

IMPLEMENTAÇÃO E RESULTADOS

ESTE capítulo avalia os resultados e o desempenho da arquitetura proposta para o alinhamento de sequências de DNA. O foco deste capítulo é apresentar a importância de cada parâmetro do algorítimo BLAST, e o seu consequente impacto em termos de área de *hardware*, frequência de operação e tempo de resposta.

A importância de se identificar o impacto de cada parâmetro para o alinhamento de DNA utilizando o algoritmo BLAST é criar uma relação entre a sensibilidade do algoritmo, tempo de resposta e área de *hardware* consumida. Como grande parte dos trabalhos utilizando o algoritmo BLAST tem como forma de implementação o *software*, é impossível construir essa análise de área consumida contra o desempenho do algoritmo. Estabelecer a relação dos parâmetros de *hardware*, área e frequência, com os parâmetros do algoritmo BLAST é essencial para implementar uma versão otimizada em desempenho e área.

Na Seção 6.1 é descrita a metodologia utilizada e os parâmetros gerais de comparação. Na Seção 6.2 são comentados alguns dos resultados de simulação encontrados. Na Seção 6.3 estão os resultados de síntese, e as observações feitas com base nos mesmos.

6.1 Metodologia

Usualmente, a ferramenta BLAST, na sua versão em *software* é utilizada para alinhar múltiplas sequências de DNA ou de proteínas. A partir de uma sequência buscada, ela é submetida a um banco de dados, onde estão milhares de sequências. O resultado é qual sequência do banco de dados (ou quais) tem o maior número de alinhamentos com a sequencia submetida. Para esse par de sequências, também são demonstrados os possíveis alinhamentos. Na grande maioria dos trabalho existentes, o objetivo é melhorar a forma de pesquisa do BLAST em um banco de dados.

Neste trabalho, optou-se por realizar a versão mais primitiva do BLAST: alinhamento de um par de sequências de DNA. Qualquer outra versão do algoritmo BLAST é criada a partir de modificações nessa versão. Tomando como base o algoritmo apresentado no Capítulo 2, foi elaborada a arquitetura de *hardware* descrita nos Capítulos 3, 4 e 5. Ao construir uma arquitetura de *hardware* capaz de realizar o alinhamento de um par de sequências utilizando o algoritmo BLAST e conseguir relacionar os parâmetros do algoritmo com os requisitos de *hardware* é aberta a possibilidade de construir em *hardware* outra versão do algoritmo proposto.

As propostas de implementação do algoritmo BLAST em *hardware* são poucas, pois o custo de programação e processamento em *cluster* de computadores é cada vez mais barato, o que justifica, numa primeira análise, a opção pelo *software*. De modo geral, as abordagens em *hardware* para a comparação de sequências genéticas são baseados em autômatos finitos. O problema da solução baseada em autômatos finitos reside na escalabilidade. Como as comparações são feitas dentro dos estados do autômato, quando é necessário modificar o tamanho das sequências, é necessário modificar o autômato e consequentemente o *harware* construído.

A premissa para o projeto da arquitetura apresentada foi implementar um *hardware* escalável, para justamente possibilitar uma futura adequação a qualquer outra versão do BLAST. A arquitetura proposta nos Capítulos 3, 4 e 5 é uma arquitetura totalmente escalável, escrita em código parametrizado.

O objetivo de dividir o processamento da parte alta e parte baixa das sequências foi, além de simplificar as operações, prover escalabilidade ao *hardware* proposto, uma vez que para sequências de DNA num alfabeto de 4 letras, são utilizados 2 blocos idênticos, um para o vetor *MSB* e outro para o vetor *LSB*. Dessa forma, no caso de alinhamento de um alfabeto de 10 letras, haveriam 4 blocos idênticos; para 32 letras, 5 blocos idênticos e assim sucessivamente, de acordo com cálculo demonstrado no Capítulo 4, Seção 4.1.

Um primeiro resultado, apresentado logo no Capítulo 3, Seção 3.1, é que foi possível atrelar o número de ciclos da semeadura ao tamanho da sequência s. São necesários m ciclos para concluir a semeadura. Essa clara descrição vai de encontro e atinge um objetivo proposto pelo trabalho, que é criar relação entre os requisitos de *hardware*, os parâmetros e o desempenho do algoritmo BLAST.

Além disso, a abordagem de separar *MSB* e *LSB* permite que, com outras modificações no critério de pontuação e comparação seja construída uma arquitetura para alinhamento de sequências de DNA e proteínas. No próximo capítulo, onde são sugeridos os trabalhos futuros, essas modificações são detalhadas.

Com a metodologia de construir uma arquitetura de *hardware* escalável e paralela, o projeto descrito nos Capítulos 3, 4 e 5 foi levado à ferramentas de simulação e síntese para validação. Na Seção 6.2 são comentados as observações sobre os resultados de simulação, que foram apresentados nos Capítulos 4 e 5, nas Seções 4.5 e 5.3 respectivamente.

6.2 Resultados de Simulação

Visando validar o modelo apresentado nos capítulos anteriores, um código VHDL parametrizado foi escrito e simulado no *software* ModelSim XE III 6.4 (MODELSIM, 2011). Os resultados de simulação, foram apresentados nos Capítulos 4 e 5, e atingiram os resultados esperados, consistentes com o detalhado no Capítulo 2 que descreve o algoritmo BLAST. Dadas duas sequências de DNA s e t de tamanhos m e n, a arquitetura simulada proposta encontra os possíveis alinhamentos entre elas. Como descrito, por se tratar de um método heurístico, o BLAST não tem o compromisso de retornar sempre o melhor alinhamento entre duas sequências, mas sim, trazer um conjunto de possíveis soluções para o problema.

No *software* de simulação foi possível testar o comportamento utilizando qualquer tamanho de m e n. Na ferramenta de síntese, havia uma limitação quanto ao tamanho das sequências, que será detalhada na Seção 6.3.

Ainda na simulação, a primeira etapa do projeto foi encontrar, para cada etapa, o número ideal de estruturas paralelas. O número de estruturas paralelas na semeadura, como descrito no Capítulo 4, depende de requisitos de área, e portanto precisa da síntese para ser justificado.

Para a extensão, a simulação, para o conjunto de dados utilizados, demonstrou que a melhoria de desempenho acontece em média, para valores de p entre $2 < p < 5$. Para um número de processadores de extensão maior que 5, a melhoria é desempenho é muito pequena, e para o conjunto de dados utilizados, não justificável.

Na Seção 6.3 são demonstrados os resultados obtidos para área, frequência de operação e tempo de resposta, para os testes realizados através da ferramenta de síntese.

6.3 Resultados de Síntese

Após a escrita do código VHDL parametrizado, o mesmo foi validado, conforme as janelas de simulação apresentadas nos Capítulos 4 e 5. Nesta seção será mostrada a aplicação do algoritmo BLAST para o alinhamento de sequências de DNA construído, em um processador MicroBlaze™ (XILINX, 2005) para avaliar o tempo de processamento em software embarcado.

Essa avaliação serviu para realizar as medidas de desempenho que uma solução em *hardware*, como a proposta nesta dissertação, necessita, e assim concluir o objetivo do projeto que é estabelecer relação entre os parâmetros do algoritmo e os requisitos de *hardware*.

O MicroBlaze™ (XILINX, 2005) é um processador, otimizado para implementação em dispositivos FPGA, da empresa *Xilinx*. Este é um *soft-processor*, ou seja, é um processador construído em linguagem de descrição de *hardware*. A sua arquitetura é de 32 bits, com um conjunto de instruções reduzidas (RISC - *Reduced Instruction Set Computing*), podendo ser sintetizado e implementado em FPGA.

Para implementar o processador MicroBlaze™ em uma FPGA é necessário utilizar a ferramenta XPS da *Xilinx* (*Xilinx Platform Studio*) (XILINX, 2008). Com esta ferramenta é possível configurar o processador com os periféricos necessários e desenvolver um *software* que será embarcado no processador. Este *software* pode ser desenvolvido em linguagem C.

Foi desenvolvido um *software* em linguagem C que recebe duas sequências, e codifica esses valores em binário. Esses valores binários são enviados à arquitetura proposta, sintetizada em FPGA e acoplada ao MicroBlaze™ como um co-processador.

A FPGA utilizada foi a *Virtex* 5 (XILINX, 2011), modelo xc5vfx70t-ff1136. Ela pertence a família de FPGAs para arquiteturas digitais de alto desempenho, com 11200 *slices* contendo 4 LuTs cada; num total de 11200 *slices*, 44800 LuTs e 44800 registradores.

O kit de desenvolvimento utilizado, o *Xilinx ML505-V5LX70T Evaluation Plataform - Virtex 5*, possui, além da FPGA, um conjunto de interfaces (USB, RS-232 e VGA) e características adicionais como mostrador de cristal liquido, chaves e leds.

A implementação da arquitetura foi realizada com a ferramenta *Xilinx ISE* (XILINX, 2012). Para a avaliação da área ocupada foi considerado o número de *slices* ocupados, o número de LuTs e registradores utilizados. A ferramenta *Xilinx ISE* também informa a frequência máxima de operação da arquitetura sintetizada e mapeada no dispositivo. Os valores de área e frequência estão apresentados à seguir, de acordo com cada o teste realizado.

Foram realizados inúmeros testes pra identificar a relação e o impacto dos parâmetros do BLAST nos resultados de tempo e área consumida. De forma empírica, variando m, n, w e p, pôde-se chegar aos resultados apresentados nas Seções 6.3.1 e 6.3.2. Após apresentar os resultados relativos aos parâmetros do algoritmo, na Seção 6.3.3 são apresentados resultados de comparação entre a implementação em *hardware* e *software*.

6.3.1 Tamanho das sequências

O tamanho das sequências s e t é dado por m e n respectivamente. Apesar de m e n não terem relação direta com o número de sementes, esses parâmetros determinam o número de estruturas paralelas que serão criadas para a semeadura. Como apresentado no Capítulo 3, são criados $(n - w + 1)$ estruturas paralelas para realizar a semeadura, com lógica comparadora e filas do padrão *fifo*. Portanto o tamanho n da sequência t, e a sensibilidade w determinam o total de palavras criadas e consequentemente têm um impacto maior na área consumida. Para realizar os testes demonstrados nas Tabelas 16 e 15, os valores utilizados foram $p = 2$ e $w = 3$, variando-se m e n. A Figura 28 trás os resultados de área para as variações de de m e n, descritos no gráfico no formato m, n.

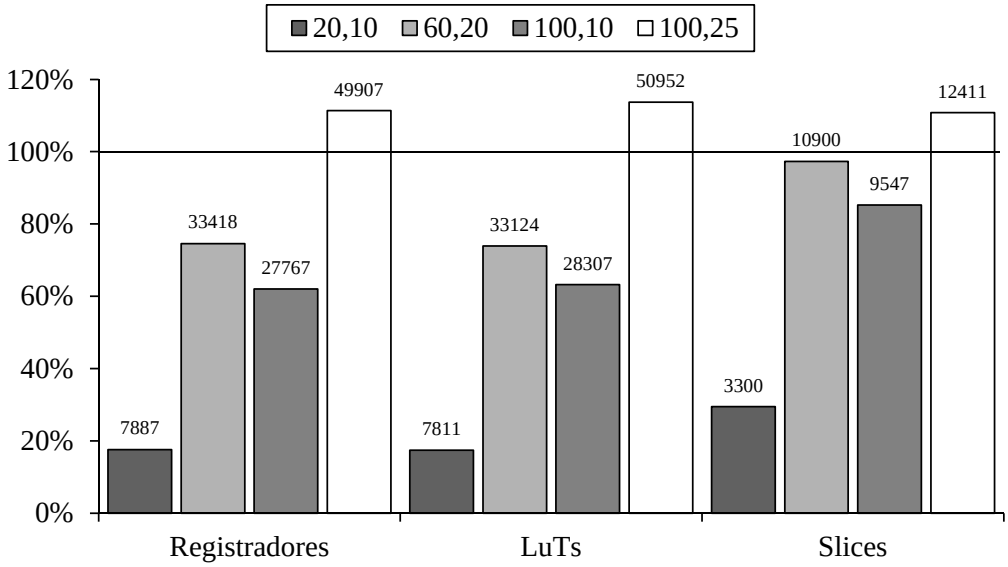
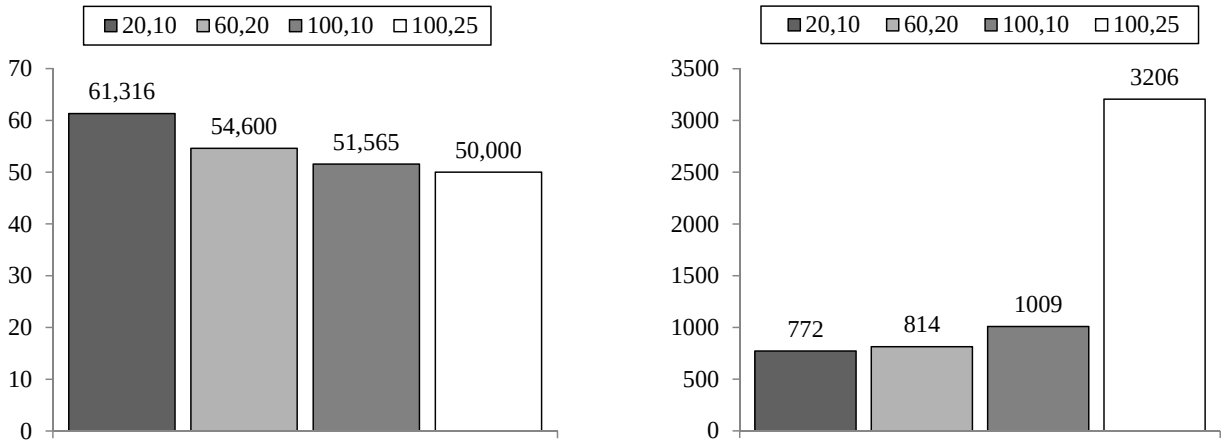


Figura 28: Resultados de área para variações em m e n

Com relação ao tempo de processamento, quanto maiores as sequências, maior o tempo demandado para analisar todos os alinhamentos. Apesar de ser uma média, é mais provável que uma sequência maior tenha um número maior de sementes. Na Figura 29 verifica-se como a variação de m e n impacta na máxima frequência de operação da placa, e consequentemente no número de ciclos, que é encontrado a partir da média de sucessivas comparações.

Os valores utilizados para construção dos gráficos e os resultados consolidados são apresentados nas Tabelas 16 e 15, onde é possível observar que para a placa utilizada, os valores de m não devem ultrapassar 60, enquanto os valores de n não devem ultrapassar 20.

É possível constatar que o parâmetro n é mais impactante em termos de área do que o parâmetro m, o que torna o tamanho n da sequência t, um parâmetro limitador para a



(a) Frequência (MHz) (b) Média de Ciclos

Figura 29: Resultados de tempo para variações em m e n.

Tabela 15: Resultados de área para $10 \leq m, n \leq 100$

m,n	Registradores	%	LuTs	%	Slices	%
20,10	7887	18	7811	17	3300	29
60,20	33418	75	33124	74	10900	97
100,10	27767	62	28307	63	9547	85
100,25	49907	111	50952	114	12411	111

execução do algoritmo. Para o alinhamento de sequências de DNA, é comum ser $m >> n$, o que torna os resultados obtidos satisfatórios com relação ao tamanho das sequências.

Tabela 16: Resultados de tempo para $10 \leq m, n \leq 100$

m,n	Frequência(MHz)	Ciclos	Tempo(ms)
20,10	61,316	772	12,591
60,20	54,600	814	14,908
100,10	51,565	1009	19,568
100,25	50,000	3206	64,120

Por se tratar de um método heurístico, o BLAST trabalha com alguns critérios visando diminuir o espaço de busca. A sensibilidade do algoritmo é o parâmetro que mostra qual o tamanho mínimo das similaridades que um determinado algoritmo consegue encontrar. No Capítulo 2 é descrito que o parâmetro w representa a sensibilidade do BLAST. Na Figura 30 são apresentados os resultados de área ocupada para uma variação de $3 < w < 7$. Para realização dos testes abaixo, os valores utilizados foram $m = 20$, $n = 10$, $p = 2$, variando o parâmetro w.

Na Tabela 17 pode-se perceber que a variação de w não se torna um fator limitante

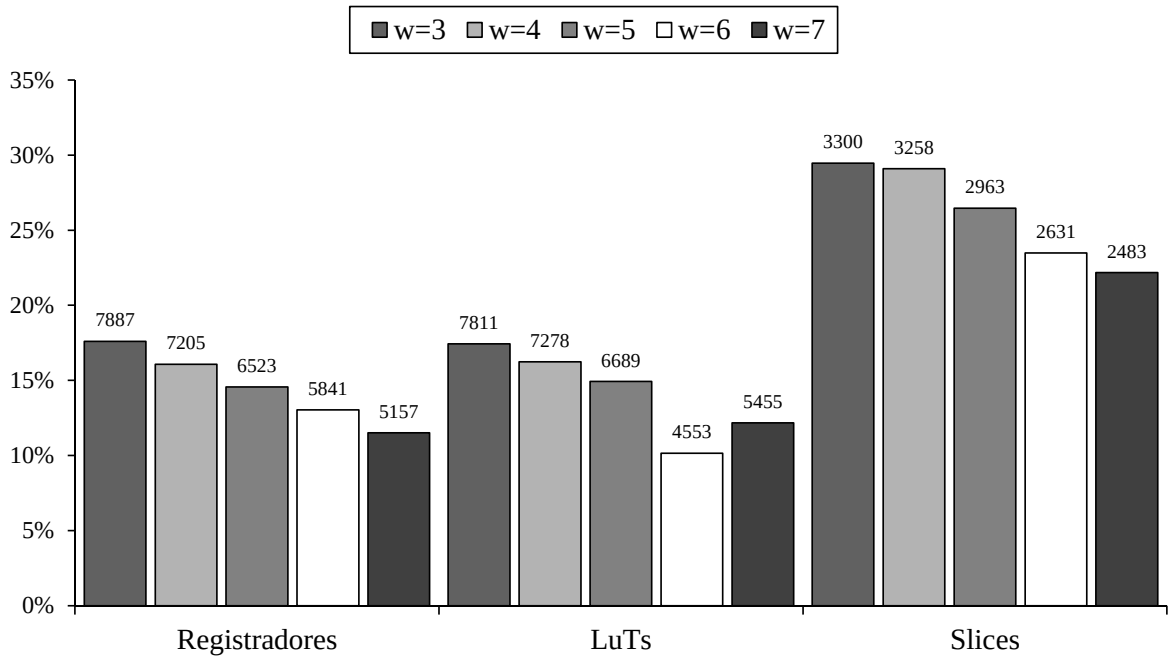


Figura 30: Resultados de área para variações em w

com relação à área de *hardware* ocupada. Portanto, o aumento do valor de w pode ser uma alternativa quando o número de registradores ou *Slices* da placa estiver ligeiramente acima dos 100 %.

Tabela 17: Resultados de área para $3 \leq w \leq 7$

w	Registradores	%	LuTs	%	Slices	%
3	7887	18	7811	17	3300	29
4	7205	16	7278	16	3258	29
5	6523	15	6689	15	2963	26
6	5841	13	4553	10	2631	23
7	5157	12	5455	12	2483	22

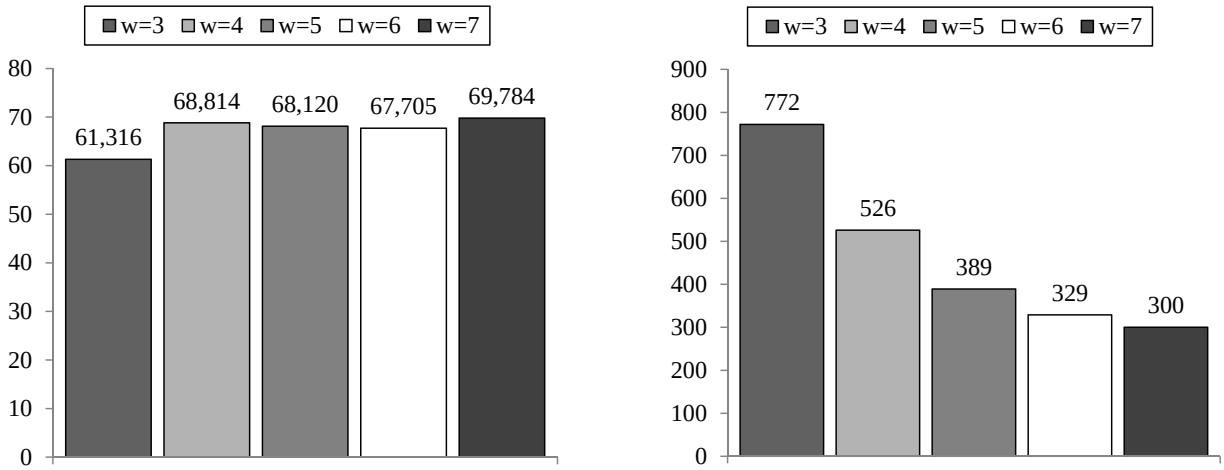
A variação de w é capaz de deixar o algoritmo muito mais lento, ou muito mais rápido, pois altera o tamanho das palavras criadas, aumentando a quantidade de trechos descartados pelo algoritmo. Essa melhoria de desempenho, tem como custo a perda da sensibilidade. Na Tabela 18 são consolidados os resultados de tempo em função da variação do parâmetro w.

Como descrito no Capítulo 3, além de alterar o tamanho das palavras criadas, uma modificação no parâmetro w muda o número de estruturas de paralelas designadas para a semeadura. Na semeadura são $n - w + 1$ estruturas paralelas, e um aumento de w não só diminui o tempo de processamento como diminui também a área de *hardware*. Na Figura 31, é mostrado que a média de ciclos cai substancialmente para um aumento de w. Quanto à

Tabela 18: Resultados de tempo para $3 \leq w \leq 7$

w	Frequência(MHz)	Ciclos	Tempo(ms)
3	61,316	772	12,591
4	68,814	526	7,644
5	68,120	389	5,711
6	67,705	329	4,859
7	69,784	300	4,299

frequência de operação, à medida que o valor de w se aproxima do valor de n, a tendência é que o valor da frequência máxima de operação da placa se estabilize.



(a) Frequência (MHz) (b) Média de Ciclos

Figura 31: Resultados de tempo para variações em w.

Além do tamanho da sequência e da sensibilidade do algoritmo, o resultado do BLAST irá sempre variar de acordo com o conteúdo das sequências submetidas à comparação. Para uma mesma sensibilidade e tamanhos, pares de sequências com conteúdo diferentes apresentam resultados de área iguais, uma vez que o *hardware* é construído baseado em parâmetros do algoritmo; todavia, os resultados de desempenho podem ser bastante distintos, estando diretamente ligado à característica das sequências. A Tabela 19 mostra a variação do número de ciclos em função da variação da quantidade de sementes presentes nas sequências de entrada. Para os valores testados, a variação no tempo de resposta atingiu até 35% .

A determinação da variação do número de ciclos em função da ocorrência ou não de extensão das sementes encontradas, é realizada através dos contadores internos presentes nos processadores de extensão, pois é impossível dissociar o número de extensões realizadas do número de sementes existentes nas sequências. Baseado nos contadores da arquitetura, entre 20 e 70 ciclos são necessários para realizar a extensão de uma semente, para os valores de $p = 2$

Tabela 19: Tempo de resposta variando-se o número de sementes entre 1 e 7

Sementes	Ciclos	Tempo(ms)
1	573	9,345
2	581	9,476
3	601	9,802
4	674	10,992
5	679	11,074
6	702	11,449
7	774	12,623

e $w = 3$, $m = 20$ e $n = 10$.

6.3.2 Número de processadores

O algoritmo BLAST realiza comparação de duas sequências de tamanhos m e n, utilizando um parâmetro w. Na arquitetura paralela proposta, foi incluído o parâmetro p, que reflete o número de processadores de extensão paralelos. A justificativa para a arquitetura dispor de um parâmetro exclusivo para determinar o paralelismo da etapa de extensão é o fato da extensão ser o gargalo de desempenho do algoritmo BLAST para o alinhamento de sequências de DNA. Segundo Altschul (ALTSCHUL et al., 1997) a etapa de extensão pode consumir até 90 % do tempo de execução do algoritmo.

Visando identificar a relação de melhoria do desempenho e confrontá-la com o aumento de área para a inserção de mais processadores, realizaram-se testes com $w = 3$, $m = 20$ e $n = 10$, variando o parâmetro p. Na Tabela 20 observa-se o impacto em área da inserção de mais ou menos processadores.

Tabela 20: Resultados de área para $1 \leq p \leq 5$

p	Registradores	%	LuTs	%	Slices	%
1	6435	14	6622	15	3109	28
2	7887	18	7811	17	3300	29
3	7887	18	7804	17	3438	31
4	8004	18	7989	18	3672	33
5	8052	18	7999	18	3900	35

É possível observar que, o acréscimo de processadores também não é um fator impactante em área consumida. Exceto se os recurso da FPGA estiverem próximos do 100 % , o acréscimo de processadores de extensão não irá prejudicar nesse quesito. Na Figura 32 é possível observar o gráfico do aumento de registradores, *Slices* e *LuTs* de acordo com o valor de p.

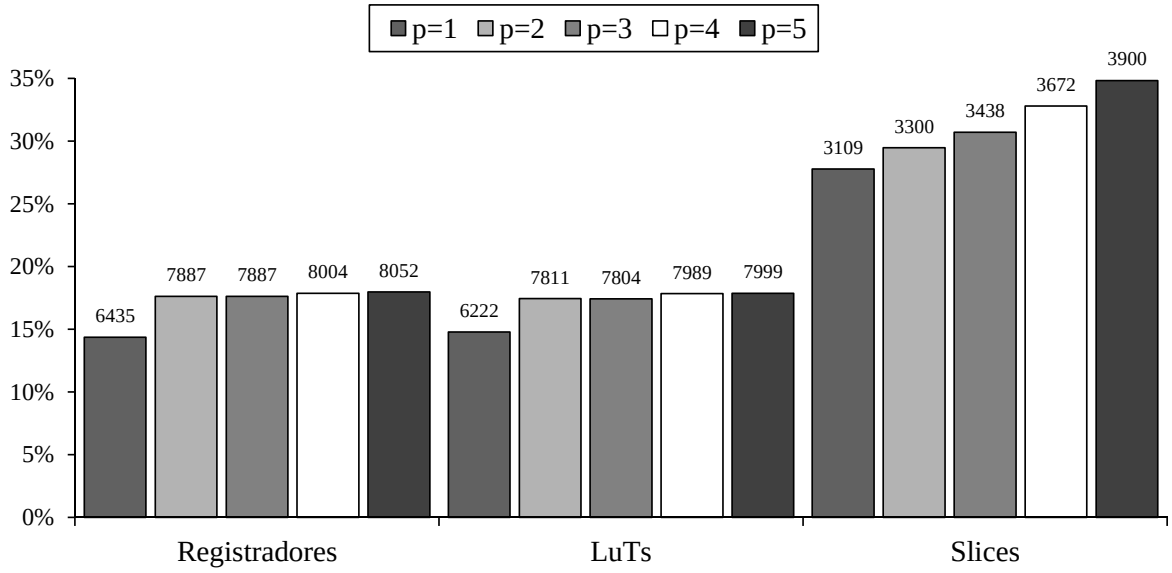


Figura 32: Resultados de área para variações em p

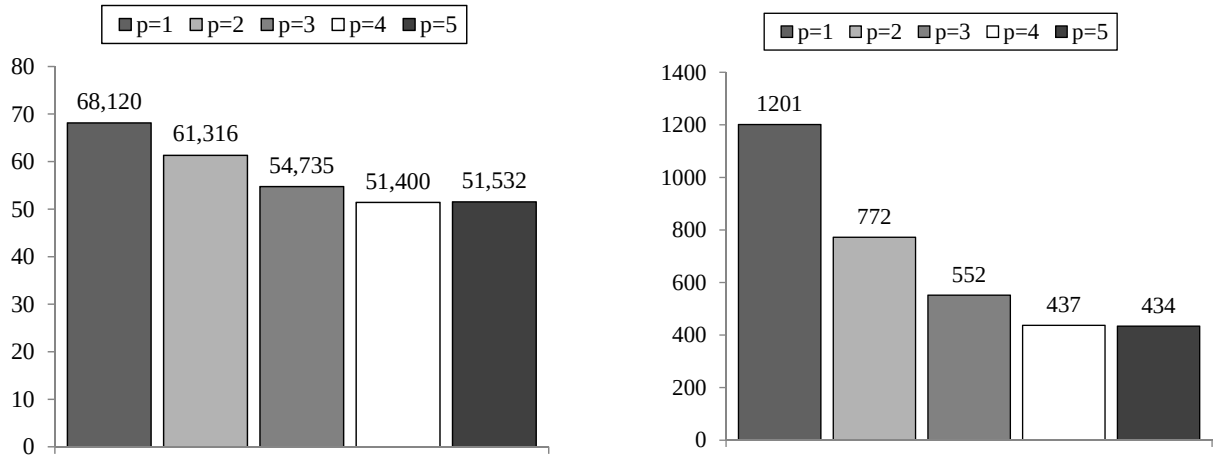
Uma vez que não é limitador em termos de área, é importante determinar o ganho em desempenho ao se inserir p processadores de extensão. Para um mesmo conjunto de dados, foram realizados os testes apresentados na Tabela 21, que visam demonstrar a diminuição da média de ciclos à medida que são inseridos processadores de extensão.

Tabela 21: Resultados de tempo para $1 \leq p \leq 5$

p	Frequência(MHz)	Ciclos	Tempo(ms)
1	68,120	1201	17,631
2	61,316	772	12,591
3	54,735	552	10,085
4	51,400	437	8,502
5	51,532	434	8,422

Através dos testes realizados, foi possível verificar, conforme descrito no Capítulo 4, que a arquitetura proposta apresenta limitação quanto ao número máximo de processadores de extensão. Embora não seja limitante em área consumida, o aumento de processadores de extensão diminui drasticamente a frequência de operação do *hardware* proposto. A placa utilizada tem um limite mínimo de 50 MHz, valor que limita a 5 processadores por placa. A partir desse valor, é necessário a inclusão de divisor de frequência (*flip-flop*). Na Figura 33 é observa-se o gráfico para os valores de p.

O ganho obtido com a implementação paralela, através do parâmetro p, foi elevado. Na figura, verifica-se que para $p = 5$ o tempo de processamento foi metade do tempo de processamento para $p = 1$. À partir de $p = 4$, a média de ciclos necessários para o algoritmo



(a) Frequência (MHz) (b) Média de Ciclos

Figura 33: Resultados de tempo para variações em p.

BLAST retornar os resultados tende a estabilidade. Sendo assim, para arquitetura proposta, o valor máximo, diante das limitações de frequência apresentadas é de $p = 5$.

6.3.3 Comparação dos resultados

A proposta de implementação em *hardware* do algoritmo BLAST, suscita a comparação entre o desempenho de cada abordagem utilizada e uma outra abordagem através de *software*. Para realizar tal comparação, executou-se no MicroBlaze o código em ANSI/C++ que realiza as mesmas funções implementadas pelo *hardware* proposto. Os mesmos conjuntos de sequências que foram submetidos aos testes em *hardware* foram submetidos novamente aos testes em *software*, com os resultados apresentados na Tabela 22. Para os testes a frequência de operação da MicroBlaze é de 50,000 MHz.

Tabela 22: Resultados de *software* para $10 \leq m, n \leq 100$

m,n	Ciclos	Tempo(ms)
20,10	32411	528,589
60,20	54393	996,208
100,10	54919	1065,044
100,25	255454	5109,080

Na Figura 34, há a comparação entre os resultados em *hardware*, utilizando o co-processador e *software*, utilizando apenas a MicroBlaze; variando-se o tamanho das sequências. O *speed up* médio encontrado é de 60 vezes, sendo que para o conjunto de dados testados, o ganho da implementação em *hardware* varia de 41 à 79 vezes quando comparada a aplicação em *software*.

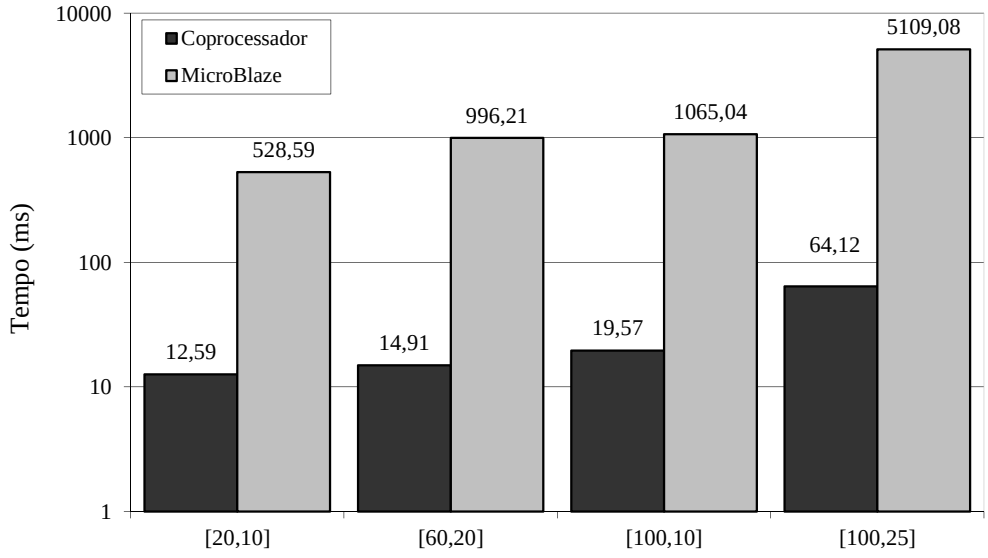


Figura 34: Comparação entre implementações em *hardware* e *software*

6.4 Considerações Finais do Capítulo

Neste capítulo foram apresentado os resultados de testes de síntese, e comentados os resultados obtidos por síntese e simulação. Foram apresentados e ilustrados o quanto cada parâmetro do algoritmo BLAST impacta em termos de área e desempenho da arquitetura sintetizada em FPGA.

Com os resultados obtidos, e as observações feitas, classificam-se os resultados como extremamente satisfatórios, pois contemplam o objetivo de estabelecer uma relação entre os parâmetros do algoritmo e os requisitos de um sistema embutido.

Verificou-se através dos resultados de simulação e síntese que embora o tempo de resposta do algoritmo BLAST esteja fortemente relacionado com os parâmetros (p, w), através dos resultados de síntese foi possível observar que o tamanho das sequências (m, n) também torna-se extremamente importante por ser um fator limitador devido à área de *hardware* disponível. Com essa avaliação, é possível determinar que a melhor configuração para a arquitetura de *hardware* proposto, na placa disponível, podendo-se variar o valor de w, ajustando a sensibilidade. Também demonstrou-se que a arquitetura proposta é em média 60 vezes mais rápida quando implementada em *hardware*.

Atingidos os objetivos propostos para os testes, o capítulo seguinte finaliza este trabalho, abordando suas principais conclusões, bem como os pontos mais relevantes da presente dissertação. Também serão tratadas direções para possíveis trabalhos futuros.

Capítulo 7

CONCLUSÕES E TRABALHOS FUTUROS

O ALGORITMO BLAST para alinhamento de sequências de DNA foi estudado nesta dissertação visando construir uma arquitetura de *hardware* paralela. Dentre os métodos heurísticos, o algoritmo BLAST se destaca por ser mais rápido e intuitivo, quando comparado aos demais.

Nessa dissertação apresentou-se uma arquitetura paralela de *hardware* implementada em lógica reconfigurável, que executa a versão do algoritmo BLAST para alinhamento de duas sequências de DNA. O simples fato de implementar o algoritmo em *hardware* numa plataforma escalável e de baixo custo é um resultado considerável e que vislumbra algumas possibilidades, ao passo que, a partir do estudo, simulação e implementação da arquitetura são possíveis realizar algumas conclusões acerca do trabalho exposto.

Na Seção 7.1 estão as conclusões sobre o trabalho, e na Seção 7.2 a sugestão de trabalhos futuros.

7.1 Resumo e Conclusão

Esta dissertação trouxe a proposta de uma arquitetura paralela de *hardware* para o alinhamento de sequências de DNA. Esse estudo foi motivado pela carência de plataformas reconfiguráveis que simulem a modelagem em *hardware* e que se proponham a realizar o alinhamento de sequências genéticas, uma vez que o perfil dominante da área de bioinformática é realizar essa comparação através de *software*.

A abordagem e *software* é extremamente vantajosa em alguns aspectos como flexibilidade e custo, mas em geral, perde muito nos quesitos desempenho e segurança. A implementação do algoritmo BLAST em *hardware* trás como primeiro resultado o questionamento sobre a preferencia pelo *software* em detrimento do *hardware*. Através desse trabalho é pos-

sível observar que o volume de dados utilizados para o alinhamento de sequências genéticas não é compatível com a capacidade da plataforma utilizada, pois na FPGA utilizada nesse trabalho, o tamanho máximo de cada sequência deve ser de uma centena de bases, enquanto no processamento real de sequencias genéticas esse valor é da ordem de milhares ou milhões de bases. Portanto, em *software* essa limitação de capacidade é mais facilmente solucionada, pois recursos como o processamento em nuvem, vêm para sanar as carências de processamento.

Embora a capacidade das FPGAS utilizadas sejam um fator limitante, este trabalho apresenta uma arquitetura totalmente escalável, de modo a possibilitar que modificações posteriores sejam realizadas, otimizando assim a capacidade do *hardware* proposto.

Ao se implementar a versão básica do algoritmo BLAST, que realiza o alinhamento de duas sequências de DNA sem a inserção de buracos, a arquitetura proposta implementada atingiu seu principal objetivo. Neste trabalho foi possível relacionar de forma clara os parâmetros do algoritmo BLAST e os requisitos de *hardware*. A partir dessa relação, é possível estimar ou construir versões da arquitetura proposta para um determinado compromisso de área ou de desempenho. Como descrito, as poucas abordagens em *hardware* para o alinhamento de sequências genéticas utilizam técnicas não-escaláveis, e não fazem clara relação entre os parâmetros do BLAST e os requisitos de *hardware*. Por se tratar de um método heurístico, os parâmetros do BLAST tem vital importância para se construir um bom alinhamento entre as sequências (SEGUNDO; NEDJAH; MOURELLE, 2011).

No tocante ao desempenho, os resultados de simulação e síntese demonstram que os objetivos de construir uma arquitetura capaz de executar o alinhamento usando o BLAST de forma paralela foram perfeitamente atingidos. A estratégia de inserir *pipeline* entre as etapas e a possibilidade de utilizar o paralelismo estrutural dentro de cada etapa concedeu ao *hardware* tempos de resposta aceitáveis, ainda que na ausência de parâmetros fidedignos de comparação.

Além dos resultados apresentados, um código VHDL parametrizável e sintetizável foi escrito e simulado no ModelSim XE III 6.4 (MODELSIM, 2011) e depois sintetizado com a ferramenta *Xilinx ISE* (XILINX, 2012). Nele, estão embutidos diversas técnicas de desenvolvimento em *hardware*, onde pode ser aplicado o conhecimento de arquitetura de computadores, algoritmos genéticos, processamento distribuído, interface E/S, protocolos de comunicação, gerenciamento de recursos, lógica digital, teoria de sistemas e diversos outros temas. Dessa forma, ao se propor uma arquitetura de *hardware* paralela para alinhamento de sequências de DNA, muitos outros questionamentos e soluções foram levantados, de modo que, além do resultado satisfatório encontrado, principal objetivo, o trabalho também se refletiu num importante

aprendizado.

A partir dos resultados obtidos, na Seção 7.2, são sugeridos alguns próximos passos para continuidade do trabalho apresentado nessa dissertação.

7.2 Trabalhos Futuros

Nesta seção, são citadas algumas possíveis mudanças na arquitetura proposta, com o intuito de melhorar seu desempenho e transformá-la para realizar as demais versões do algoritmo BLAST.

A primeira sugestão a ser feita é, a partir das limitações impostas pelo *hardware* disponível, testar a arquitetura paralela proposta em outra plataforma com maior capacidade. Dessa forma, as limitações de m, n estariam suavizadas. Como as sequências de DNA reais têm tamanhos de ordem de 1000 bases, a implementação dessa arquitetura em placas de maior capacidade possibilitaria o teste de casos reais de alinhamentos.

Uma outra sugestão é melhorar a arquitetura do árbitro e do processador de extensão, de modo que o tempo gasto para escolha do processador e da fila prioritária não tenham interferência no desempenho final da arquitetura. Dentro das limitações apresentadas, os componentes atuais tem desempenho satisfatório. Porém se cessadas as limitações no tamanho das sequências, o volume de requisições no árbitro e no processador tendem a inviabilizar a continuidade da estrutura atual, baseadas em interrupções e comunicação por *flags*. Como solução, deve ser estudado um componente único que controle a comunicação e faça a extensão das etiquetas de forma paralela.

Por fim, uma possibilidade concreta deixada por esse trabalho é a modificação da lógica de comparação para possibilitar o alinhamento não só de sequências de DNA, como de qualquer sequência genética. Como descrito, cada tipo de sequência possui um alfabeto próprio. Isso não representa um problema para a arquitetura atual, uma vez que a mesma foi projetada de forma escalável, separando a parte alta da parte baixa das sequencias de DNA. Dessa forma, para realizar o alinhamento de sequências de proteínas, ao invés de 2 *hardwares* idênticos, usariam-se 5 blocos iguais. As modificações necessárias seriam voltadas então para a lógica de comparação, além da inserção do uma etapa de pré-processamento. Na lógica de comparação, pois para sequências de DNA o BLAST considera somente trechos idênticos, enquanto para proteínas, ao invés da igualdade é usado o critério da pontuação de um determinado trecho, comparando-o com os valores de corte T e S. A inserção do pré-processamento seria necessário para construir o alfabeto de palavras semelhantes, outra etapa que a versão do BLAST para DNA não realiza.

REFERÊNCIAS

ALTSCHUL, S. et al. Basic local alignment search tool. *J Mol Biol*, v. 3, n. 215, p. 403–410, 1990.

ALTSCHUL, S. F. et al. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, v. 25, n. 17, p. 3389–3402, 1997.

BAFNA, V.; LAWLER, E. L.; PEVZNER, P. A. Approximation algorithms for multiple sequence alignment. *Theoretical Computer Science*, n. 182, p. 233–244, 1997.

BALDI, P.; BRUNAK, S. *Bioinformatics: the machine learning approach*. 2nd. ed. : MIT Press, 2001. ISBN 0-262-02506-X.

BAXEVANIS, D. A.; OUELLETTE, B. F. F. *Bioinformatics: a practical guide to the analysis of genes and proteins*. Terceira. : Wiley-Interscience, 2004. ISBN 0-471-47878-4.

BRITO, R. T. D. *Alinhamento de sequências biológicas*. Dissertação (Mestrado) — IME/USP, 2003.

Mercury blastn: Faster dna sequence comparison using a streaming hardware architecture.

CHARRAS, C.; LECROQ, T. Sequence comparison. Gaspard-Monge Institut. 1998.

CORPORATION, T. *TeraBLAST DeChyper*. 2011. Disponível em: <http://www.timelogic.com>.

DAYHOFF, M. O.; SCHWARTZ, R. M.; ORCUTT, B. C. Atlas of protein sequence and structure. *Natl. Biomed. Res. Found. ,Washington*, v. 5, n. 3, p. 345–358, 1978.

DURBIN, R. et al. *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*. : Cambridge, 1998. ISBN 0-521-62971-3.

GIBAS, C.; JAMBECK, P. *Developing Bioinformatics Computer Skills*. : O'Reilly Media, 2001. ISBN 1-565-92664-1.

GIEGERICH, R. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, v. 8, n. 16, p. 665–677, 2000.

Gene Matching Using JBits, v. 2438. 1168–1171 p.

HENIKOFF, S.; HENIKOFF, J. G. A model for evolutionary change in proteins. *Proc. Natl. Acad. Sci. USA*, v. 89, n. 1, p. 10915–10919, 1992.

FPGA Implementation of Systolic Sequence Alignment. 183–191 p.

Searching Genetic Databases on Splash 2. 185-191 p.

KASAP, S.; BENKRID, K. High performance phylogenetic analysis with maximum parsimony on reconfigurable hardware. *IEEE Transactions on Very Large Scale Integration VLSI Systems*, v. 99, p. 1–13, 2010.

KORF, I.; YANDELL, M.; BEDELL, J. *BLAST*. : O'Reilly Media, 2003. ISBN 0-596-00299-2.

LOYTYNOJA, A.; GOLDMAN, N. Phylogeny-aware gap placement prevents errors in sequence alignment and evolutionary analysis. *Science 20*, European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton CB10 1SD, UK., v. 320, n. 5883, p. 1632–1635, 2008.

LUETHY, R.; HOOVER, C. Hardware and software systems of accelerating common bioinformatics sequence analysis algorithms. *BIOSILICO*, v. 2, n. 1, 2004.

MARKEL, S.; LEON, D. *Sequence Analysis*. Primeira. : O'Reilly Media, 2003. ISBN 0-596-00494-1.

MODELSIM. *High performance and capacity mixed HDL simulation*. 2011. Disponível em: <http://model.com>.

MOUNT, D. W. *Bioinformatics: sequence and genome analysis*. 2nd. ed. : Cold Spring Harbor Laboratory Press, 2004. ISBN 0-879-69712-1.

MOUNT, D. W. Steps used by the blast algorithm. *Cold Spring Harbor Protocols: Molecular Biology*, n. 7, 2007.

MOUNT, D. W. Using blosum in sequence alignments. *Cold Spring Harbor Protocols: Molecular Biology*, n. 6, 2008.

MOUNT, D. W. Using pam matrices in sequence alignments. *Cold Spring Harbor Protocols: Molecular Biology*, 2008.

RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation. 196–198 p.

NEEDLMAN, S.; WUNSH, S. A general method applicable to the search of similarities in amino acid sequence of two protein. *J Mol Biol*, v. 1, n. 48, p. 443–453, 1970.

OEHMEN, C.; NIEPLOCHA, J. Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *IEEE Transactions on Parallel e Distributed Systems*, v. 8, n. 17, p. 740–749, 2006.

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. *Sistemas Operacionais.* 4th. ed. : Bookman, 2010. (Livros Didáticos Informática).

Hyper Customized Processors of Bio-Sequence Database Scanning on FPGAs. 229–237 p.

PEARSON, W. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. *Genomics*, v. 3, n. 11, p. 635–650, 1991.

PEARSON, W. Comparison of methods for searching protein sequence databases. *Protein Science*, v. 6, n. 4, p. 1145–1152, 1995.

PEARSON, W. R. Rapid and sentive sequence comparison with pastp and fasta. *Methods Enzymol*, n. 183, p. 63–98, 1990.

A Run-Time Reconfigurable System of Gene-Sequence Searching. 561–566 p.

RUBIN, E.; PIETROKOVSKI, S. Heuristic methods for sequence alignment. *Advanced Topic in Bioinformatics*, 2003.

SEARLS, D. The language of genes. *Nature*, v. 1, n. 420, p. 211–217, 2002.

SEGUNDO, E.; NEDJAH, N.; MOURELLE, L. de M. A parallel architecture for dna matching. In: XIANG, Y. et al. (Ed.). *Algorithms and Architectures for Parallel Processing.* : Springer Berlin / Heidelberg, 2011, (Lecture Notes in Computer Science, v. 7017). p. 399–407. ISBN 978-3-642-24668-5.

SETUBAL, J.; MEIDANIS, J. *Introduction to Computational Molecular Biology.* : Boston: PWS Publishing Company, 1997. ISBN 0-534-95262-3.

SHPAER, E. G. et al. Sensitivity and selectivity in protein similarity searches: a comparison of smith-waterman in hardware to blast and fasta. *Genomics*, v. 2, n. 38, p. 179–191, 1996.

SMITH, T.; WATERMAN, M. Identification of common molecular subsequences. *J Mol Biol*, v. 1, n. 147, p. 195–197, 1981.

TANENBAUM, A. S. *Sistemas Operacionais: Projeto e Implementação.* 2nd. ed. Porto Alegre: Bookman, 2000. ISBN 8-573-07530-9.

TANENBAUM, A. S. *Organização estruturada de computadores.* 5th. ed. : Person, 2007. ISBN 8-576-05067-6.

WATERMAN, M. S. *Introduction to Computational Biology.* : CRC Press, 1995. ISBN 0-412-99391-0.

WOLF, W. *FPGA-based system design.* : Prentice-Hall, 2004.

XILINX. *MicroBlaze Processor Reference Guide.* USA: Xilinx, 2005.

XILINX. *Embedded System Tools Reference Manual - EDK.* 2008.

XILINX. *Virtex-5 Family Overview.* 2011.

XILINX. *ISE 10.1 Quick Start Tutorial.* 2012.

ZAHA, A.; FERREIRA, H. B.; PASSAGLIA, L. M. P. *Biologia Molecular Básica.* 3th. ed. : Editora Mercado Aberto, 2003.