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

Desenvolvimento de uma Arquitetura Dedicada em Plataforma Reconfigurável para Suporte ao Alinhamento de Seqüências

Fábio Vinícius Pinto e Silva

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Coorientadora
Profª. Drª. Alba Cristina M. de Melo

Brasília
2007

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

Coordenadora: Profª. Drª. Alba Cristina M. de Melo

Banca examinadora composta por:

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador) – CiC/UnB
Prof. Dr. Ricardo Augusto da Luz Reis – Instituto de Informática/UFRGS
Prof. Dr. Pedro de Azevedo Berger – CiC/UnB

CIP – Catalogação Internacional na Publicação

Fábio Vinícius Pinto e Silva.

Desenvolvimento de uma Arquitetura Dedicada em Plataforma Reconfigurável para Suporte ao Alinhamento de Seqüências/ Fábio Vinícius Pinto e Silva. Brasília : UnB, 2007.
17 p. : il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2007.

1. Comparação de seqüências, 2. Alinhamento de seqüências, 3. Programação dinâmica, 4. Smith-Waterman, 5. Arquiteturas sistólicas, 6. Arquiteturas reconfiguráveis, 7. FPGA

CDU 004

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

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

Desenvolvimento de uma Arquitetura Dedicada em Plataforma Reconfigurável para Suporte ao Alinhamento de Seqüências

Fábio Vinícius Pinto e Silva

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
CiC/UnB

Prof. Dr. Ricardo Augusto da Luz Reis
Instituto de Informática/UFRGS

Prof. Dr. Pedro de Azevedo Berger
CiC/UnB

Profª Drª Alba Cristina M. de Melo
Coordenadora do Mestrado em Informática

Brasília, 17 de setembro de 2007

Dedico este trabalho à minha família e aos meus amigos.

Agradecimentos

Ao meu orientador Prof. Dr. Ricardo Jacobi,
pela orientação e confiança depositada na realização deste trabalho.

Ao Dr. Donald E. Knuth, por ter desenvolvido o TEXe
ao Dr. Leslie Lamport, por ter criado o LATEX.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Isaac Newton

Resumo

Encontrar e visualizar semelhanças entre seqüências de DNA permite aprofundar o conhecimento sobre genomas de organismos em Biologia Molecular. Com o número de seqüências disponíveis para consulta em alguns bancos de dados crescendo exponencialmente, surge um desafio para a ciência da computação. É o de construir sistemas de informática com desempenho suficiente para permitir comparar seqüências genômicas em tempo hábil para a pesquisa e com um custo viável. Freqüentemente são usadas soluções heurísticas, devido ao grande tempo computacional necessário para o uso de soluções exatas. Soluções exatas atualmente apresentam complexidade de tempo quadrática em computadores convencionais, dificultando seu uso prático para seqüências de comprimento como as de aplicações reais.

O principal objetivo deste trabalho é viabilizar o uso de algoritmos exatos para comparação de seqüências genômicas, acelerando a obtenção de seus resultados. É proposto um arranjo sistólico de elementos de processamento em hardware reconfigurável. Assim, é explorado o paralelismo potencial do algoritmo de programação dinâmica de Smith-Waterman, reduzindo sua complexidade de tempo de quadrática para linear. É proposta uma solução para minimizar o problema de gargalo de comunicação, esperado por uma implementação "ingênua" da solução. Além do sistema proposto, a prototipação realizada em FPGA é descrita, incluindo uma análise do desempenho obtido.

Palavras–Chave: Comparação de seqüências, Alinhamento de seqüências, Programação dinâmica, Smith-Waterman, Arquiteturas sistólicas, Arquiteturas reconfiguráveis, FPGA.

Abstract

To find and to visualize similarities between DNA sequences allow to deepen the knowledgement on genomas of organisms in Molecular Biology. With the number of available sequences for consultation in some data bases growing exponentially , a challenge for the computer science appears. It is to construct computing systems with enough performance to allow to compare genomics sequences in skillful time for the research and at a viable cost. Frequently heuristical solutions are used, due to the great computational time necessary to the use of exact solutions. Exact solutions currently presents quadratic time complexity in conventionals computers, making difficult its practical use for sequences of length as of real applications.

The main objective of this work is to make possible the use of exact algorithms for comparison of genomics sequences, by speeding up the attainment of its results. A systolic arrangement of elements of processing in reconfigurable hardware is proposed. This way, the potential parallelism of the algorithm of dynamic programming of Smith-Waterman is explored, reducing its time complexity from quadratic to linear. Is also proposed a solution to minimize the problem of communication bottleneck, waited in a "naive" implementation. Besides the proposed system, the prototipation made in FPGA is described, including an analysis of the performance gotten.

Keywords: Sequence comparison, Sequence alignment, Dynamic Programming,

Smith-Waterman, Sistolic Architectures, Reconfigurable Architectures, FPGA.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASIC	*Application Specific Integrated Circuit* - Circuito integrado específico para aplicação
CPU	*Central Processing Unit* - Unidade central de processamento
CUPS	*Cell updates per second* - Atualizações de células por segundo
FPGA	*Field Programmable Gate Array* - Matriz de portas programáveis em campo
GPU	*Graphics Processing Unit* - Unidade gráfica de processamento
SISD	*Single Instruction Single Data* - Instrução única, dado único
DNA	*Deoxyribonucleic Acid* - Ácido desoxiribonucleico
RNA	*Ribonucleic Acid* - Ácido ribonucleico
PCI	*Pheripheral Components Interconnect* - Interconexão de componentes periféricos
GPP	*General Purpose Procesor* - Processador de uso geral
PLD	*Programmable Logic Device* - Dispositivo de lógica programável
CPLD	*Complex Programmable Logic Device* - Dispositivo complexo de lógica programável
SIMD	*Single Instruction Multiple Data* - Instrução única, dado múltiplo
MMX	*Multimedia Extensions* - Extensões multimídia
MMX2	*Multimedia Extensions* - Extensões multimídia 2
SSE	*Streaming SIMD Extensions* - Extensões SIMD para streaming

SSE2 *Streaming SIMD Extensions 2* - Extensões SIMD para streaming

SSE3 *Streaming SIMD Extensions 3* - Extensões SIMD para streaming

SSE4 *Streaming SIMD Extensions 4* - Extensões SIMD para streaming

RAM *Randomic Access Memory* - Memória de acesso randômico

Lista de Símbolos

$S_{i,j}$ célula da matriz da linha i, coluna j

1 Introdução

1.1 Motivação

A principal motivação para este trabalho é ajudar a viabilizar o uso de algoritmos exatos para comparação e alinhamento de seqüências, particularmente seqüências de DNA. A comparação de seqüências de DNA é, atualmente, uma importante ferramenta para a derivação de novos conhecimentos em biologia molecular. No entanto, devido a complexidade de tempo dos algoritmos exatos existentes ser no mínimo $O(m \times n)$ (m e n o tamanho das seqüências) (GOTOH, 1982), (SMITH; WATERMAN, 1981),(MYERS; W.; MILLER, 1988), apresentar soluções ótimas para o problema de comparação e alinhamento de DNA tem se tornado um desafio para a ciência da computação.

Por exemplo, o cromossomo humano X possui 154 milhões de pares de bases e o cromossomo Y possui 57 milhões. Usando o algoritmo SSEARCH34 (Smith-Waterman clássico) incluso na suíte FASTA (FASTA, 2006), um processador Intel Xeon 2.6 GHz, em um processamento serial, levaria 5,3 anos para comparar dois cromossomos X e Y (HARPER, 2007). Além disso, bancos de dados com seqüências vindas de organismos vivos - chamados bancos de dados biológicos - têm apresentado crescimento até exponencial no número de dados disponíveis desde à última década. Exemplos são o GenBank (figura 1.1) (GENBANK, 2006) e o UniProtKB/TrEMBL *Protein Database* (figura 1.2) ((EBI), 2007).

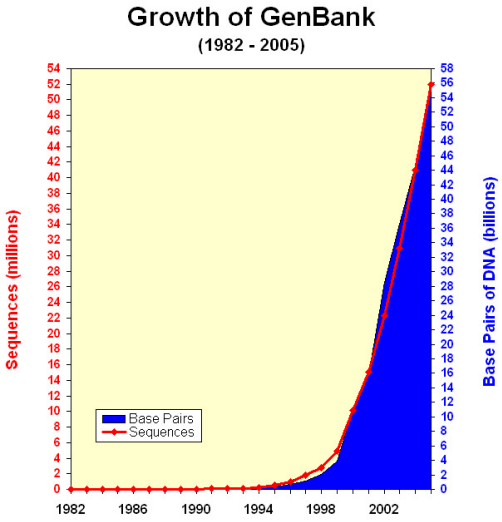

FIGURA 1.1: Crescimento do banco de dados biológico Genbank

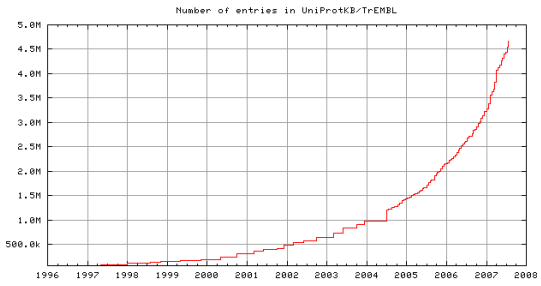

FIGURA 1.2: Crescimento do banco de dados biológico UniProtKB/TrEMBL

Assim, tratando-se de seqüências de DNA, a complexidade de tempo dos algoritmos ótimos de alinhamento tem inviabilizado seu uso em situações práticas. Para permitir a busca por soluções exatas nessas situações, desenvolvedores de hardware e software têm sido constantemente desafiados a produzirem sistemas de computação com desempenho mais eficiente. Ao invés dos algoritmos exatos, soluções probabilísticas, ou heurísticas, têm sido utilizadas, pela sua menor complexidade de tempo. Apesar de algumas dessas abordagens heurísticas, implementadas por exemplo em softwares de alinhamento de seqüências como o BLAST (BLAST, 2006) e o FASTA (FASTA, 2006), terem uma sensibilidade considerada boa, as melhores respostas podem não ser encontradas.

O algoritmo exato de alinhamento de seqüências de Smith-Waterman (SMITH; WATERMAN, 1981) apresenta resultado exato, mas quando implementado seqüencialmente, por exemplo por *software* num computador SISD, apresenta complexidade de tempo quadrática. Este algoritmo usa técnicas de programação dinâmica para evidenciar as posições de maior similaridade entre duas seqüências, segundo uma métrica adotada. No entanto, quando lidamos com seqüências reais de DNA temos dificuldades de tempo para computar os melhores alinhamentos por um algoritmo de complexidade quadrática, devido ao número de caracteres (bases) que cada seqüência pode ter.

Assim, se essa complexidade de tempo puder ser reduzida, a utilização de algoritmos exatos nesse campo poderá ser viabilizada na prática. Uma das alternativas para a redução da complexidade de tempo é o uso de técnicas de paralelismo. Através de uma abordagem paralela em *hardware* dedicado, utilizando diversos elementos de processamento em um único *chip*, a complexidade de tempo do algoritmo pode ser reduzida de quadrática para linear (LIPTON; LOPRESTI, 1985). Neste projeto é considerada a implementação deste *hardware*, dedicado ao alinhamento de seqüências, em uma arquitetura reconfigurável, dando maior efetividade de custo e tempo ao projeto quando comparado a soluções utilizando ASICs (*Application Specific Integrated Circuits*).

Considerando o cálculo da matriz de similaridade em *hardware*, fica evidente que se todas as posições desta fossem comunicadas ao meio externo por um barramento como o PCI, haveria um gargalo de comunicação na saída de dados (MENG; CHAUDHARY, 2007) (BYUN, 2005). Não há capacidade de transferência suficiente para transportar os dados gerados pelo *chip* na mesma velocidade de produção possível. Por exemplo, se o FPGA estiver com a mesma freqüência de operação do barramento PCI, poderiam ser lidos somente 64 *bits* de dados por ciclo. Esta é uma taxa de saída pequena para dar vazão à matriz de similaridade, comparada ao desempenho possível nas demais partes do projeto.

Então, é necessária uma proposta para evitar o atraso associado a esse gargalo de comunicação, sempre observando o requisito de obtenção do(s) melhore(s) alinhamento(s), incluindo o ótimo. A principal motivação é, então, permitir o uso de um algoritmo exato de alinhamento de seqüências, através da redução da complexidade de tempo deste por uma implementação paralela em *chip*, analisar os problemas de gargalo de comunicação associados a integração deste *hardware* com um sistema externo e propor soluções.

1.2 Objetivos

O principal objeto de estudo da presente monografia consiste no projeto de um sistema de computação, capaz de executar o algoritmo de alinhamento local de seqüências de Smith-Waterman. A redução da complexidade de tempo se dará pela exploração do paralelismo potencial do algoritmo. Para computar em paralelo, é utilizado um arranjo sistólico de elementos de processamento, implementado em um circuito integrado.
Quanto à saida de dados do *hardware*, se for comunicada toda a matriz de similaridade gerada pelo algoritmo, teremos notoriamente um problema de gargalo de comunicação. Outros autores já apontaram este gargalo como sendo limitante do desempenho de comparadores de seqüências por hardware (MENG; CHAUDHARY, 2007) (BYUN, 2005). Faz parte do escopo deste projeto, encontrar meios de lidar com este gargalo de comunicação, minimizando seu impacto no desempenho global do sistema. Mais especificamente, pretende-se:

- Apresentar os fundamentos da biologia molecular e do alinhamento de seqüências;
- Introduzir o uso de hardware específico e arquiteturas reconfiguráveis para computação de alto-desempenho, em especial para o problema de alinha-

mento de seqüências;

- Realizar uma revisão do estado da arte em alinhamento de seqüências;
- Propor uma abordagem em *hardware* de um algoritmo exato de alinhamento de seqüências, utilizando um arranjo sistólico de elementos de processamento em *chip*;
- Propor maneiras de acoplamento deste *hardware* a um sistema externo, considerando o gargalo de comunicação esperado. Propor técnica (em *hardware* e possivelmente *software*) para reduzir o impacto deste gargalo no desempenho do sistema.

1.3 Organização da Dissertação

No capítulo 2 são apresentados os conceitos fundamentais da biologia molecular, incluindo a relação entre seqüências de DNA, aminoácidos e proteínas. No capítulo 3, é definida a comparação e o alinhamento de seqüências. São apresentados os principais algoritmos de alinhamento baseados em programação dinâmica. É discutida, no capítulo 4, a implementação em *hardware* dedicado de algoritmos de programação dinâmica. No capítulo 5 é feita uma revisão do estado da arte. São descritas implementações publicadas para a aceleração do alinhamento de seqüências, principalmente os projetos que utilizam *hardware* dedicado. No capítulo 6 é descrita a implementação proposta neste trabalho. É feita uma análise sobre os principais problemas encontrados. O sistema proposto é descrito de uma forma geral e cada componente de *hardware* é descrito detalhadamente. São mostrados, no capítulo 7, os resultados alcançados com a prototipação do sistema. O desempenho global calculado é comparado com o de outras soluções. Finalmente, no capítulo 8, temos conclusões sobre o projeto desenvolvido.

2 Biologia Molecular

A Biologia Molecular é uma ciência que tem como objeto de estudo as interações bioquímicas celulares relacionadas à estrutura e função do material genético e nos seus produtos de expressão, as proteínas (BIOLOGIA..., 2006). O termo molecular é utilizado porque no estudo feito observa-se as interações em nível de moléculas. A área tem sobreposição com outras áreas da biologia e da química, especialmente genética e bioquímica. A Biologia Molecular estuda também as interações entre vários sistemas da célula, partindo da relação entre o DNA, o RNA e a síntese de proteínas, e o modo como essas interações são reguladas.

2.1 DNA, aminoácidos e proteínas

As características físicas e comportamentais de um ser vivo são determinadas (ou herdadas) através da informação codificada em um par de moléculas chamado DNA. O DNA faz parte do grupo dos ácidos nucléicos (apresentam propriedades ácidas e estão presentes no núcleo das células) e é composto por um par de longas moléculas, enroladas em torno de seu próprio eixo e ligadas por pontes de hidrogênio (figura 2.1). Cada uma das moléculas (ou fita) é uma seqüência de nucleotídeos. Cada nucleotídeo é composto de um açúcar (pentose), um fosfato e uma base nitrogenada heterocíclica. As bases nitrogenadas de um nucleotídeo podem ser: adenina (A), citosina (C), guanina (G) ou timina (T) (figura 2.1).

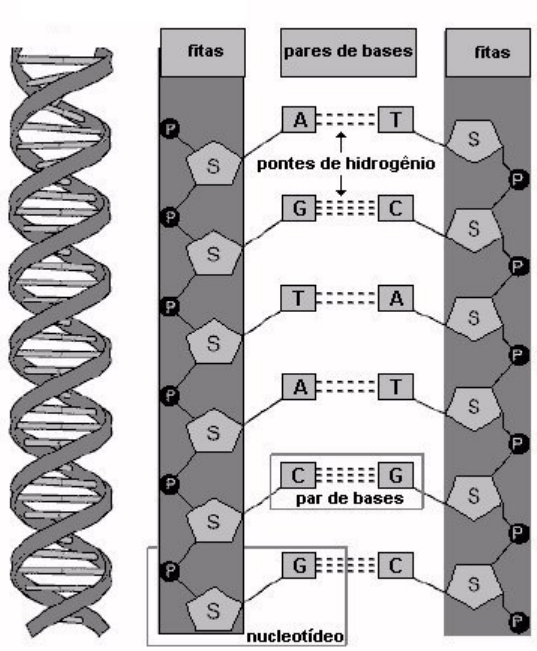

FIGURA 2.1: Estrutura do DNA

Os pares de bases sempre são ou C e G ou A e T. A determinação da seqüência de bases que compõe um DNA é uma tarefa essencial que permite análises em biologia molecular.

Cada grupo de três nucleotídeos, chamado códon, codifica a síntese de um aminoácido diferente (figura 2.2).

Primeira Posição	Segunda Posição U	Segunda Posição C	Segunda Posição A	Segunda Posição G	Terceira Posição
U	UUU Phe	UCU Ser	UAU Tyr	UGU Cys	U
	UUC Phe	UCC Ser	UAC Tyr	UGC Cys	C
	UUA Leu	UCA Ser	UAA STOP	UGA STOP	A
	UUG Leu	UCG Ser	UAG STOP	UGG Cys	G
C	CUU Leu	CCU Pro	CAU His	CGU Arg	U
	CUC Leu	CCC Pro	CAC His	CGC Arg	C
	CUA Leu	CCA Pro	CAA Gln	CGA Arg	A
	CUG Leu	CCG Pro	CAG Gln	CGG Arg	G
A	AUU Ile	ACU Trh	AAU Asp	AGU Ser	U
	AUC Ile	ACC Trh	AAC Asp	AGC Ser	C
	AUA Ile	ACA Trh	AAA Lys	AGA Arg	A
	AUG Met	ACG Trh	AAG Lys	AGG Arg	G
G	GUU Val	GCU Ala	GAU Asp	GGU Gly	U
	GUC Val	GCC Ala	GAC Asp	GGC Gly	C
	GUA Val	GCA Ala	GAA Glu	GGA Gly	A
	GUG Val	GCG Ala	GAG Glu	GGG Gly	G

FIGURA 2.2: Códons e codificação de aminoácidos

A junção de aminoácidos, por meio de ligações peptídicas, forma uma proteína. As proteínas são moléculas essenciais para a vida, exercendo várias funções como

catalisadoras de reações, funções mecânicas ou estruturais, funções no sistema imunológico, entre outras.

Os genes, triplas de nucleotídeos que que codificam os aminoácidos e por conseguinte as proteínas, são considerados as unidades da hereditariedade. A informação registrada nos genes, controla o desenvolvimento do organismo.

2.2 Bancos de dados biológicos

Com o avanço da Biologia Molecular, o número de organismos que tiveram seu DNA seqüenciado, ou seja, a seqüência de bases que compõe a dupla fita determinada, aumentou significativamente. Para posterior consulta, as seqüências são inseridas em bancos de dados. Assim, um banco de dados biológico é uma coleção de informações que representam DNAs, proteínas ou outras seqüências biológicas. Com o avanço no seqüenciamento de espécies, alguns bancos de dados biológicos têm apresentado taxa de crescimento muito grande no volume de informações. Como pode ser visto nas figuras 1.1 e 1.2, o GenBank (GENBANK, 2006) e o UniProtKB/TrEMBL ((EBI), 2007) são exemplos de bancos de dados biológicos que têm apresentado crescimento exponencial. No Genbank, em fevereiro de 2006, havia 59,7 bilhões de nucleotídeos em mais de 54,5 milhões de seqüências cadastradas.

2.3 Bioinformática e biologia computacional

Bioinformática e biologia computacional são áreas de estudo em que se usam técnicas de matemática aplicada, informática, estatística e ciência da computação em aplicações de biologia molecular. Embora os dois termos freqüentemente sejam usados sem distinção, cabe a seguinte ressalva: Bioinformática pode ser de-

finido como todo tipo de estudo ou ferramenta usado para produzir ou organizar informações biológicas (COMISSION, 2001). Tem seu foco na produção de novos conhecimentos a partir da grande quantidade de dados disponíveis para a biologia, principalmente no que se refere às seqüências de caracteres armazenadas. Já a Biologia Computacional tem o foco no desenvolvimento de algoritmos e programas computacionais para auxiliar a bioinformática na busca de conhecimento (CARVALHO, 2003).

3 Alinhamento de Seqüências

3.1 Comparação e alinhamento

Comparação de seqüências é a primitiva mais importante em Bioinformática, servindo como base para muitas outras manipulações complexas, e é vastamente utilizada nos projetos de seqüenciamento (CARVALHO, 2003). As similaridades descobertas podem ser conseqüências de relações funcionais, estruturais ou evolucionárias. Frequentemente, uma seqüência recém-descoberta (chamada seqüência de consulta) é comparada com um conjunto de seqüências armazenadas em um banco-de-dados.

A comparação de seqüências pode ser dividida em duas partes:

- **valor da similaridade**: uma métrica que represente o grau de semelhança entre as seqüências. Pode ser entendida como o quão distante uma seqüência está da outra, ou como o custo para transformar uma seqüência na outra;
- **alinhamento**: consiste em escrever as seqüências uma acima da outra, emparelhando-se bases e possivelmemte espaços, tornando claras as correspondências entre os caracteres semelhantes (SMITH; WATERMAN, 1981).

Um exemplo de alinhamento pode ser visto na figura 3.1. O alinhamento de seqüências é um problema de casamento aproximado de padrões, possivelmente introduzindo-se espaços (SETUBAL; MEIDANIS, 1997).

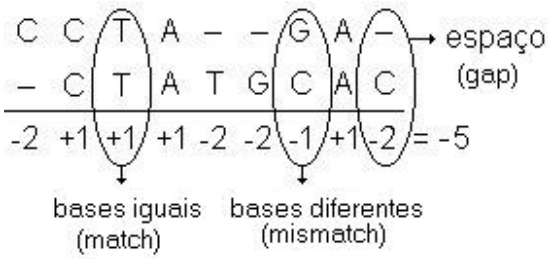

FIGURA 3.1: Exemplo de alinhamento entre duas seqüências

Após escrever as seqüências uma acima da outra, para calcular a pontuação do alinhamento atribui-se um valor para cada coluna. A pontuação total do alinhamento é a soma da pontuação de todas as colunas. O valor da coluna depende dos caracteres emparelhados nesta:

- se as bases forem iguais (*match*), o valor é um bônus (por exemplo +1);
- se as bases forem diferentes (*mismatch*) o valor é uma penalização (por exemplo -1);
- se for uma base e um espaço (*gap*), o valor é uma penalização (por exemplo -2). A possibilidade de alinhamento com espaço permite lidar com situações como duplicações, repetições e variações genéticas.

3.2 Alinhamento ótimo

O alinhamento ótimo entre duas seqüências é o que apresenta a maior pontuação possível de ser construída (SETUBAL; MEIDANIS, 1997). Se mais de um alinhamento apresentar a mesma pontuação e esta for a máxima, todos esses alinhamentos são considerados ótimos. A pontuação do alinhamento ótimo é considerada uma métrica para a similaridade das seqüências.

Mas, como computar o(s) alinhamento(s) ótimo(s) ? Enumerar todas as possibilidades é uma tarefa difícil, já que o número alinhamentos possíveis é expo-

nencial em relação ao tamanho das seqüências (SMITH; WATERMAN, 1981), dado aproximadamente pela fórmula $A_n = (1 + \sqrt{2})^{2n+1}\sqrt{n}$, onde n é o tamanho das seqüências.

3.3 Algoritmos baseados em programação dinâmica

Utilizando programação dinâmica, é possível determinar-se o alinhamento ótimo em tempo polinomial. Isso é possível pois esta classe de algoritmos não computa todas as possibilidades, apenas os subalinhamentos ótimos (alinhamento ótimos de prefixos) são considerados para a composição do alinhamento ótimo.

Algoritmos de programação dinâmica são adequados para problemas de otimização que podem ser decompostos em subproblemas de mesma estrutura. A solução de cada subproblema é memorizada e usada mais de uma vez para resolver problemas maiores. No caso, o alinhamento ótimo depende dos alinhamentos ótimos de prefixos das seqüências.

Para memorizar os resultados intermediários é usada uma tabela, ou matriz, chamada matriz de programação dinâmica ou de similaridade. Ela guarda os resultados intermediários durante a construção da solução global.

3.4 Comparação global - Needleman e Wunsch

Needleman e Wunsch, em 1970 (NEEDLEMAN S. B. E WUNSCH, 1970) apresentaram o primeiro algoritmo para computar os melhores alinhamentos entre duas seqüências baseado em programação dinâmica. Nesse algoritmo, o problema de encontrar o melhor alinhamento entre duas seqüências é dividido em subproblemas de alinhar pares de bases, uma base de cada seqüência e, possivelmente,

espaços. A solução de cada subproblema considera as soluções dos subproblemas menores, mais a escolha de um dos três pareamentos possíveis:

- alinhar as bases das duas seqüências;
- alinhar um espaço na primeira seqüência com a base da segunda;
- alinhar a base da primeira seqüência com um espaco na segunda.

Para encontrar os melhores alinhamentos possíveis, a matriz de similaridade é construída. Os maiores valores encontrados na matriz correspondem aos alinhamentos com a maior pontuação. Para computar cada célula $S_{i,j}$ da matriz, o algoritmo calcula a pontuação dos três alinhamentos possíveis e seleciona o de maior valor, conforme a equação:

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + 1 & match \\ S_{i-1,j-1} - 1 & mismatch \\ S_{i-1,j} - 2 & gap \\ S_{i,j-1} - 2 & gap \end{cases}$$

Seguindo o esquema de pontuação citado anteriormente, a figura 3.2 mostra a matriz de similaridade produzida pela aplicação do algoritmo sobre as seqüências AGT e AACGT. As setas indicam qual dos três valores da equação de recorrência foi a origem da maior pontuação para cada célula. Pode existir mais de uma seta por posição, indicando valores iguais. O cálculo inicia-se no canto superior esquerdo da matriz (célula $S_{1,1}$).

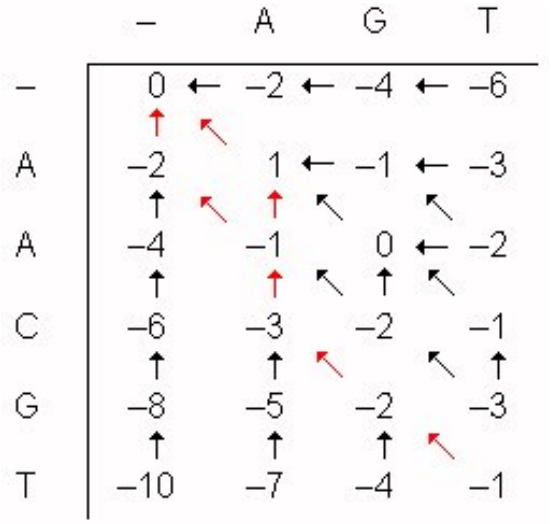

FIGURA 3.2: Alinhamento global seqüências AGT e AACGT

3.5 Comparação local - Smith-Waterman e variações

Relações surpreendentes têm sido descobertas entre proteínas que tem pouca similaridade global, mas nas quais subseqüências similares podem ser encontradas. Nesse sentido, a identificação de subseqüências similares é provavelmente o método mais útil e prático de comparar duas seqüências (OLIVER; SCHMIDT; MASKELL, 2005).

Um método de alinhamento local difere do global pois pode considerar a possibilidade do alinhamento de uma subseqüência da primeira seqüência com uma subseqüência da segunda e não somente das seqüências completas. O método de alinhamento local é capaz de identificar sub-regiões de similaridade entre as seqüências (SMITH; WATERMAN, 1981). No contexto de alinhamento local, a similaridade de duas seqüências é definida como sendo a pontuação máxima entre todos os possíveis alinhamentos locais. Em 1981 (SMITH; WATERMAN, 1981) Smith e Waterman propuseram um método de alinhamento local, baseado em duas modificações no algoritmo de Needleman-Wunsch:

- inclusão de um zero como possibilidade de escolha na função de máximo da equação de recorrência. Como conseqüência, passa a não existir números negativos na matriz, permitindo a busca de subalinhamentos e transformando o método de alinhamento global em local;

- modificação da penalidade para espaços juntos, introduzindo uma formula de cálculo para esta situação baseado no comprimento da seqüência de espaços.

Com referência ao segundo item, a inclusão da penalidade variável para espaços juntos aumenta a complexidade de tempo do algoritmo de $O(n^2)$ para $O(n^3)$. Gotoh (GOTOH, 1982), em 1982, reduziu a complexidade novamente para $O(n^2)$, usando uma função para o cálculo de espaços juntos. Quando espaços aparecem juntos, a penalização é da forma $u + k \times v$, onde u é uma penalidade de abertura do espaço, v é uma penalidade para a extensão do espaço e k é o número de espaços sucessivos.

3.6 Paralelismo potencial

Pela equação de recorrência, observa-se que cada célula da matriz pode ser calculada assim que estiverem disponíveis os elementos $S_{i,j-1}, S_{i-1,j}$ e $S_{i-1,j-1}$.

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + 1 & match \\ S_{i-1,j-1} - 1 & mismatch \\ S_{i-1,j} - 2 & gap \\ S_{i,j-1} - 2 & gap \end{cases}$$

Assim, o cálculo do valor de cada célula S_{ij} da matriz depende do valor da célula a esquerda (S_{ij-1}), da célula acima (S_{i-1j}) e da célula na diagonal superior-esquerda (S_{i-1j-1}). Dada esta dependência de dados, uma forma de paralelizar

é calcular as células de uma mesma antidiagonal "ao mesmo tempo". As antidiagonais podem ser computadas a partir do canto superior esquerdo. Células de uma antidiagonal dependerão somente de células antidiagonais calculadas previamente. A este tipo de computação dá-se o nome de computação em onda (TRELLES, 2001). A figura 3.3 ilustra o paralelismo possível.

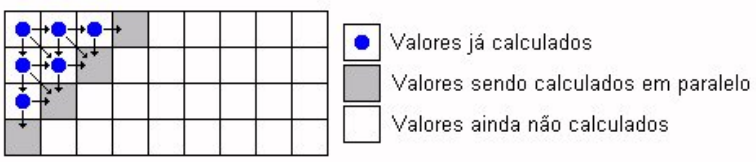

FIGURA 3.3: Paralelismo possível em programação dinâmica

Esse potencial de paralelismo é importante pois permite uma abordagem que pode reduzir a complexidade de tempo. Isto porque, enquanto o número de células da matriz de similaridade é quadrático ($m \times n$), o número de antidiagonais é linear ($m+n-1$), em relação ao tamanho das seqüências (m, n). Por exemplo, se ambas seqüências tiverem 1.000 bases (um número pequeno para aplicações reais), o número de células da matriz é 1.000.000, enquanto o número de antidiagonais é 1.999 (aproxidamente 500 vezes menor). Supondo que o cálulo completo, se pudermos calcular em paralelo as antidiagonais, leve 1 hora, o cálculo em série das células levaria aproximadamente 21 dias. Se o número de bases for 1.000.000 (um milhão), teremos 1.000.000.000.000 (um trilhão) de células e 1.999.999 (quase dois milhões) de antidiagonais. Se for possível o cálculo em paralelo e este leve 1 dia, o cálculo serial das células levaria aproximadamente 1.370 anos. Assim, o cálculo em paralelo pode tornar o uso destes algoritmos viável, em aplicações onde o excessivo tempo necessário antes era impeditivo.

4 Hardware para Alinhamento de Seqüências

4.1 Hardware dedicado versus processadores de uso geral

Processadores de uso geral (em inglês GPP - *General Purpose Processors*) são construídos para atender diversos tipos de aplicações. Funcionam executando um programa e, tradicionalmente, em um modelo de arquitetura de Von Neuman. Por abrangerem diversos tipos de aplicações, os processadores de uso geral não têm muitas funções especializadas. Além disso, buscar, decodificar e executar instruções é um *overhead* para este tipo de processador. Já um circuito integrado projetado especificamente para uma aplicação (ASIC - *Application Specific Integrated Circuit*) freqüentemente tem desempenho superior, pois não depende de programa e é especializado numa aplicação. Disso decorre menor gasto de área em chip, tempo de projeto, de energia, de matéria prima, entre outros. Soluções construídas em hardware têm a vantagem do paralelismo inerente ao circuito eletrônico, onde muitos fluxos de dados podem existir ao mesmo tempo. Abordagens em *hardware* apresentam elevados índices de aceleração quando comparadas a abordagens em *software*, principalmente quando o algoritmo apresenta potencial de paralelismo. A principal desvantagem de implementações em hardware é

a menor flexibilidade. O tempo e o custo para fabricação de circuitos integrados são grandes, comparados ao *software*. Por outro lado, se forem produzidos em escala, o alto custo inicial do projeto pode ser diluído pelas unidades, tornando o custo da solução em *hardware* mais atraente.

4.2 Arquiteturas reconfiguráveis e FPGAs

Entre os critérios de flexibilidade (observado nas soluções por *software*) e de desempenho (observado nas soluções por *hardware*) estão os dispositivos de lógica programável. Os dispositivos de lógica programável podem assumir, por meio de um processo de reconfiguração, o comportamento de um circuito eletrônico desejado. Os componentes internos desses dispositivos recebem uma programação que define sua funcionalidade. Com o uso de arquiteturas reconfiguráveis, que utilizam lógica programável, aplicações podem ter desempenho próximo a de um ASIC com flexibilidade próxima a de software. As arquiteturas reconfiguráveis tiveram seu início com os PLDs (*Programmable Logical Devices*) e CPLDs (*Complex PLDs*). Uma evolução dos PLDs são os FPGAs (*Field Programmable Gate Arrays*). Nos FPGAs, os elementos de lógica programável (ou blocos de lógica) podem ser configurados para se comportarem como funções booleanas, combinacionais, matemáticas mais complexas ou como memória. As conexões entre os blocos de lógica também são configuráveis. A figura 4.1 ilustra a estrutura básica de um FPGA, com blocos de lógica e interconexões configuráveis.

FPGAs são geralmente mais lentos que ASICs, não conseguem implementar projetos tão complexos quanto aqueles e usam mais energia. No entanto, têm vantagens como menor tempo de projeto (e time-to-market), capacidade de reprogramação (para correção de erros, modificações em campo, ou mesmo dinamicamente, para uso do algoritmo) e, como são produzidos em escala, podem ter um preço atrativo. Hoje em dia, são usados tanto para prototipação de circuitos

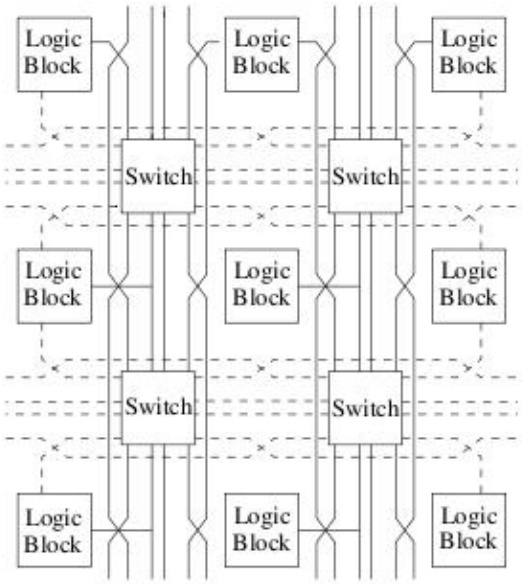

FIGURA 4.1: Estrutura básica de um FPGA

integrados como para implementação final de soluções.

4.3 Arranjos sistólicos de elementos de processamento

O termo sistólico para um arranjo de elementos de processamento foi citado pela primeira vez por H. T. Kung em 1979 (KUNG; LEISERSON, 1979), por notar analogia entre esta estrutura e o mecanismo de bombeamento de sangue do coração. Um arranjo sistólico é uma disposição regular de elementos de processamento nos quais os dados fluem somente entre os vizinhos mais próximos, existindo diversos fluxos de dados simultâneos. Cada processador, a cada ciclo, recebe as entradas de um ou mais vizinhos, processa e gera saída para um ou mais vizinhos. Alguns tipos de arranjos sistólicos pode ser vistos na figura 4.2.

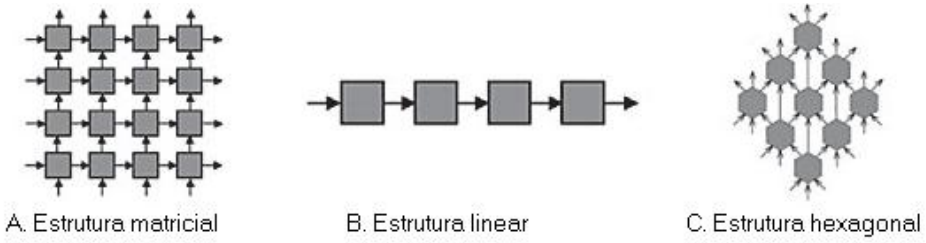

FIGURA 4.2: Elementos de processamento em arranjos sistólicos

4.4 Explorando o paralelismo em programação dinâmica

Lipton e Lopresti (LIPTON; LOPRESTI, 1985) mostraram que o paralelismo no algoritmo de comparação local de Smith-Waterman pode ser explorado por um arranjo sistólico de elementos de processamento (ou processadores), permitindo a construção paralela da matriz de programação dinâmica. A utilização de arranjos sistólicos permite dois tipos de mapeamento, emulação das diagonais (bidirecional, cada elemento processador é responsável pelo cálculo de uma diagonal da matriz) ou emulação de coluna (unidirecional, cada elemento produz uma coluna da matriz), conforme ilustrado na figura 4.3.

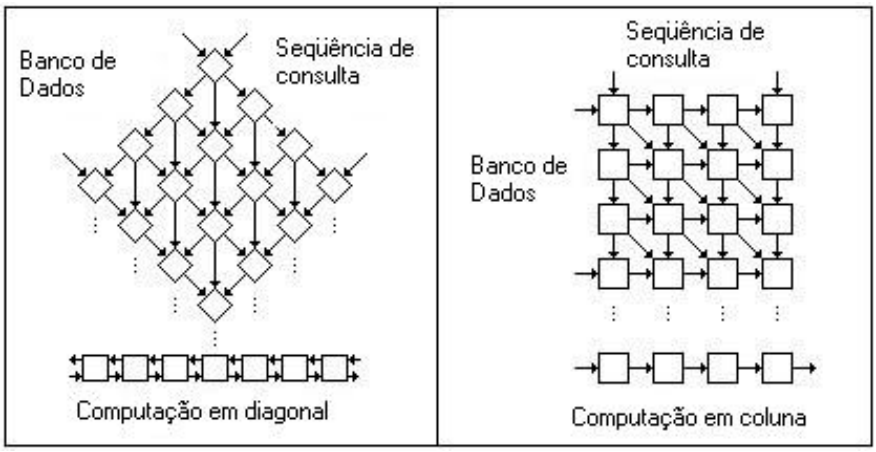

FIGURA 4.3: Mapeamento de programação dinâmica em arranjos sistólicos

Na emulação de diagonal são usados $n + m + 1$ processadores (m, n tamanho das seqüências), que é o número de diagonais. Na emulação de colunas, utilizando arranjo sistólico linear, o número de processadores depende somente do número de bases da seqüência de consulta. Para este projeto, foi escolhida emulação de colunas, principalmente por ter o número de processadores dependendo somente do tamanho da seqüência de consulta. A implementação paralela do algoritmo de Smith-Waterman reduz a complexidade de tempo de quadrática, $O(m \times n)$, para linear, $O(m + n)$. Para implementar o algoritmo no sistólico linear, a cada base da seqüência de consulta deve corresponder um processador, conforme ilustrado pela figura 4.4. O primeiro processador tem fixa a primeira base da seqüência de consulta, o segundo processador a segunda base e assim sucessivamente.

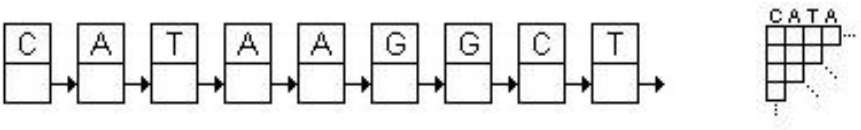

FIGURA 4.4: Correspondência entre as bases da seqüência de consulta e os processadores

Para o cálculo simultâneo de cada antidiagonal da matriz de similaridade, a seqüência do banco de dados é deslocada da esquerda para direita, a partir do primeiro processador, de forma a atravessar o sistólico. A cada ciclo de relógio, as bases do banco de dados são deslocadas uma posição para a direita (figura 4.5). gerando um novo pareamento com as bases de consulta, o que permite o cálculo da antidiagonal seguinte.

Observa-se pela figura 4.5 que, conforme as bases da seqüência do banco-de-dados são deslocadas, são gerados os pareamentos de bases necessários para o cálculo de cada antidiagonal. No exemplo, temos:

- no tempo T1, para o cálculo da primeira antidiagonal: estão pareadas as primeiras bases das seqüências (C e A)

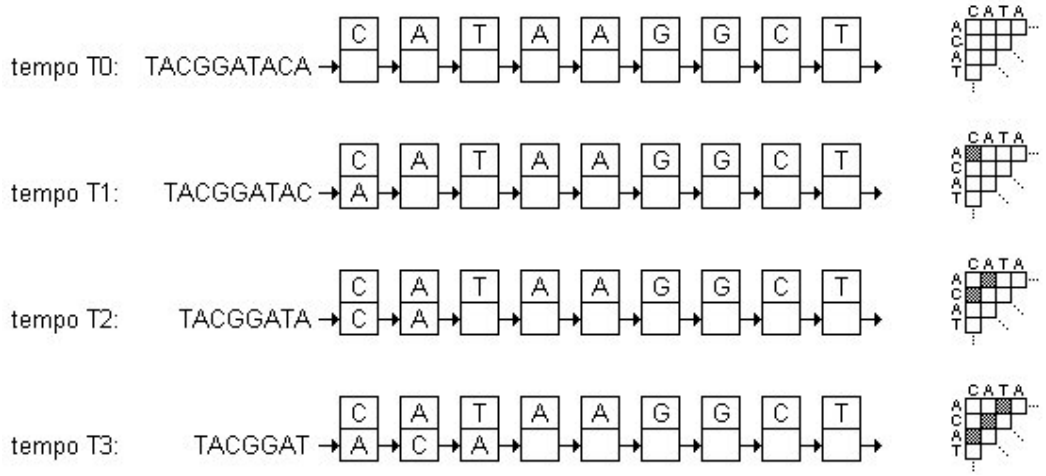

FIGURA 4.5: Os três primeiros ciclos de computação de antidiagonais

- no tempo T2, para o cálculo da segunda antidiagonal: estão pareadas
  - a primeira da seqüência de consulta (C) com a segunda da seqüência do banco-de-dados (C);
  - a segunda da seqüência de consulta (A) com a primeira da do banco-de-dados (A).

A cada ciclo, tem-se os pareamentos de bases necessários para computar uma antidiagonal. Além das bases pareadas, para o cálculo de cada célula da matriz, precisamos, segundo a equação de recorrência, do valor da célula à esquerda na matriz, da acima e da célula na diagonal superior-esquerda.

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + 1 & diagonal \\ S_{i-1,j-1} - 1 & diagonal \\ S_{i-1,j} - 2 & cima \\ S_{i,j-1} - 2 & esquerda \end{cases}$$

O valor de cada célula S_{ij}, depende do valor de S_{i-1j-1}, S_{i-1j} e S_{ij-1}.

$$\begin{array}{ccc} S_{i-1j-1} & & S_{i-1j} \\ & \searrow & \downarrow \\ S_{ij-1} & \rightarrow & S_{ij} \end{array}$$

Por esta dependência de dados e, com cada antidiagonal da matriz sendo calculada em um ciclo de relógio, vemos que o cálculo de uma antidiagonal depende de células somente das duas antidiagonais anteriores. Um processador para o cálculo de $S_{i,j}$, tem que ter memorizado o valor de $S_{i,j-1}$ e $S_{i-1,j}$, presentes na antidiagonal (ciclo) anterior e $S_{i-1,j-1}$, presente duas antidiagonais (ciclos) atrás. As células que serão necessárias para cálculos futuros são memorizadas internamente por cada processador. A memorização destes valores é feita da seguinte forma:

- $S_{i-1,j}$ foi gerado por este mesmo processador (está na mesma coluna) no tempo $t-1$ (um ciclo de relógio atrás ou na antidiagonal anterior). Então, como será necessário no ciclo seguinte, cada valor gerado pelo processador é memorizado em um registrador. Na figura 4.6, temos uma representação da memorização dos dados por um processador. O valor calculado para a célula corrente, V_i, é armazenado no registrador L (de linha), para uso no ciclo seguinte;

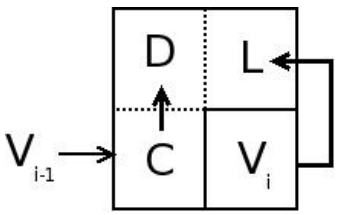

FIGURA 4.6: Elemento de processamento - memorização das células vizinhas

- $S_{i,j-1}$ foi calculado pelo processador vizinho da esquerda, em $t-1$. Na figura, o valor recebido do vizinho da esquerda, V_{i-1}, é memorizado no registrador C (de coluna), para uso no ciclo seguinte;

- $S_{i-1,j-1}$ foi calculado pelo processador à esquerda (coluna à esquerda) em $t-2$ (duas antidiagonais atrás). Assim, este dado corresponde ao valor do registrador C (item acima), com um ciclo de diferença (atraso). É representado pela letra D (diagonal).

As três setas na figura 4.6 representam as movimentações de dados que ocorrem, a cada ciclo, com os valores das células. Estes três registradores (C, D e L), que contém os valores das células vizinhas, mais as bases de consulta (Bc) e do banco-de-dados (Bbd), são as informações necessárias para o cálculo de cada célula, conforme a equação de recorrência. Na figura 4.7, está representado este cálculo, que é combinacional.

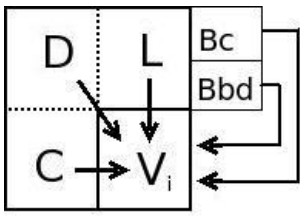

FIGURA 4.7: Elemento de processamento - cálculo combinacional da célula

Esse modelo de elemento de processamento sistólico é o que foi escolhido para implementação neste projeto. O sistema proposto será descrito com detalhes no capítulo 6.

5 Revisão do Estado da Arte

5.1 Utilizando FPGAs

Em 1985, Lipton e Lopresti (LIPTON; LOPRESTI, 1985) mostraram que o paralelismo existente no algoritmo de Smith-Waterman poderia ser mapeado em uma estrutura linear sistólica, reduzindo a complexidade de tempo do algoritmo de $O(m \times n)$ para $O(m + n)$. Na estrutura proposta, cada elemento de processamento do vetor sistólico era responsável pelo cálculo de uma das diagonais da matriz de similaridade. A cada ciclo de relógio, as seqüências se deslocavam em sentidos opostos e mais valores iam sendo calculados. Se as duas seqüências a serem comparadas possuíssem comprimentos iguais a m e n, a estrutura linear proposta deveria ter um comprimento de $m + n - 1$ (o número de diagonais). De acordo com bits gerados pelos elementos de processamento, um contador ia sendo incrementado ou decrementado. No final do processo, o último valor do contador indicava quão próximas eram as seqüências comparadas. Se o valor fosse baixo, indicava que as seqüências eram bem parecidas, com um valor igual a zero no caso das seqüências serem idênticas. Valores altos no contador indicavam que as seqüências eram bem diferentes. Não era possível recuperar nenhum alinhamento entre as duas seqüências. O resultado obtido era somente um valor indicador da proximidade entre elas.

Baseadas neste trabalho de Lipton e Lopresti, foram propostas várias imple-

mentações. A quase totalidade destas não inclui a recuperação dos alinhamentos, diferentemente do presente trabalho. Ao invés disso, utilizam elementos de processamento de 2 *bits* e entregam somente um indicador de proximidade entre as seqüências. Os alinhamentos que apresentaram a maior pontuação não são apontados pelo *hardware*. A recuperação dos melhores alinhamentos constitui um importante diferencial do presente trabalho.

Em 1992, Hoang (HOANG, 1992) descreveu uma solução bastante similar a de Lipton e Lopresti, utilizando a arquitetura SPLASH (*Stanford Parallel Applications for Shared Memory*) uma matriz lógica linear programável desenvolvida pelo *Supercomputer Researh Center* (SRC), que utilizava 32 FPGAs XC3090 da Xilinx [75]. O SPLASH-2 foi construído em 1993, funcionando como um *attached-processor*. Conecta-se a estação Sun SPARC Station 2 por um cartão adaptador SBus. Pode operar síncrona ou assincronamente com o host e utiliza transferência DMA. A programação foi feita em VHDL. Uma SPLASH-2 com 16 placas cada um com 17 XC4010 obteve 43 GCUPS.

Em 1998, utilizando o acelerador SAMBA (*Systolic Accelerator for Molecular Biological Applications*), Lavenier propõe uma nova abordagem para o problema (LAVENIER, 1998), ainda utilizando uma arquitetura sistólica. O SAMBA é uma vetor sistólico dedicado à comparação de seqüências biológicas. Implementa uma versão parametrizável do algoritmo de Smith-Waterman permitindo a computação de alinhamento global ou local com ou sem penalidades para os espaços. A aceleração obtida pelo SAMBA comparado à estações de trabalho padrão variou de 50 a 500, dependendo da aplicação.

Outra solução foi apresentada por Yamaguchi, Maruyama e Konagaya em 2002 (YAMAGUCHI; MARUYAMA; KONAGAYA,). Nela uma placa PCI contendo uma FPGA XCV2000E da Xilinx (43200 células lógicas) foi utilizada, sendo possível a implementação de 144 elementos de processamento. Cada elemento de processamento leva quatro ciclos de relógio para gerar seu resultado. Nessa solu-

ção não é dito nada sobre a utilização de uma arquitetura sistólica. A comparação é dividida em vários pedaços devido às limitações da memória interna da FPGA.

Uma nova implementação da solução de Hoang ocorreu em 2002, utilizando tecnologia mais moderna, como a família Virtex de FPGAs da Xilinx. A solução de Hoang também deu origem ao HokieGene (PUTTEGOWDA *et al.*, 2003), um sistema recongurável em tempo de execução baseado na placa Osiris desenvolvida pelo *Information Sciences Institute* - DARPA. A placa contém 2 Xilinx Virtex II. Funciona calculando a *edit distance* entre células sucessivas. Considerando que essa distância é no máximo 2, utiliza somente 2 bits para representar cada *edit-distance*. Essas informações ficam em memória RAM, compartilhada via PCI com o processador do *host*. Não é detalhado como é o pós-processamento dos dados.

Uma proposta é feita em 2003 por Yu, Kwong, Lee e Leong (YU *et al.*, 2003), da Universidade Chinesa de Hong Kong. Essa utiliza a plataforma Pilchard (LEONG *et al.*, 2001), que tem um Xilinx Virtex XCV1000E-6 conectado via slot de memória SDRAM à placa-mãe de um microcomputador *host*. A opção pelo conector SDRAM (133 MHz / 64 bits) reduz a latência, em comparação a implementações com conector PCI. Foram usadas implementações propostas por Lipton e Lopresti (LIPTON; LOPRESTI, 1985). Os elementos de processamento do vetor sistólico trabalham somente sobre dados de 1 bit. Eles calculam não o *score* acumulado mas, sim, a *edit-distance* entre elementos sucessivos. Uma lógica externa faz a acumulação das *edit-distances*. 4032 elementos de processamento couberam no XCV1000E-6, freqüência máxima de 202 MHz segundo o ISE/Xilinx. No entanto, por questões relacionadas ao acoplamento com o *host*, o circuito operou na mesma velocidade da memória (133 MHz). O FPGA fica mapeado no espaço de memória do processador *host*. A leitura dos resultados é feita por *polling*. Prevê ser possível 814 GCUPS de desempenho (4032 elementos a 202 MHz), mas obteve somente 136 GCUPS. Segundo o texto, melhorias feitas

na parte de interface poderiam levar o dispositivo ao desempenho pleno.

Em 2005, Oliver, Schmidt e Maskell (OLIVER; SCHMIDT; MASKELL, 2005), da Nanyang Technological University de Singapura, implementaram um vetor sistólico de elementos de processamento em FPGA. Como no presente trabalho, as bases da seqüência de consulta ficam fixas nos elementos de processamento e as bases do banco-de-dados percorrem o vetor da esquerda para a direita. Foi implementada penalização dos espaços linear e *affine* (com penalidades diferentes para abertura e extensão de uma série de espaços). Utilizando FPGAs Xilinx Virtex II XC2V6000, o desempenho obtido foi de 13,9 GCUPS para penalidade linear de espaço (252 elementos de processamento couberam) e 7,6 GCUPS para penalidade *affine* de espaços (169 elementos de processamento couberam).

Tom Van Court e Martin C. Herbordt (COURT; HERBORDT, 2006) utilizaram, em 2006, uma placa PCI Annapolis Micro Systems WildstarII-Pro, com 2 Xilinx Virtex II Pro XC2VP70-5 para implementar Smith-Waterman e Needleman-Wunsch. Para alinhamento de DNAs por Smith-Waterman, couberam 174 elementos de processamento por XC2VP70. O desempenho foi de 13 GCUPS (aceleração declarada de 451 em relação a um Intel Xeon 3 GHz).

Em 2007, na Universidade de Delaware, foi implementado um projeto utilizado um módulo XtremeData XD1000 (XTREMEDATA, 2007), que consiste em um Altera Stratix II conectado a um soquete de processador AMD Opteron (HyperTransport). Usa a infra-estrutura da placa-mãe para o FPGA agir como co-processador. Estando diretamente em um soquete de processador, o FPGA pode compor um sistema fortemente acoplado com processadores Opteron, compartilhando a memória principal. Não são dados detalhes sobre recuperação dos alinhamentos ou maiores informações sobre a divisão do trabalho entre o FPGA e os processadores Opteron. O desempenho de pico declarado é 2.048 GCUPS (12.288 elementos de processamento a 166,67 MHz). Em 2007 foi anunciada a linha de produtos XD2000i da XtremeData, com FPGAs Altera Stratix III e

compatível com Intel FSB (Front Side Bus).

Shum e Truong (LI; SHUM; TRUONG, 2007) da Universidade de Toronto implementaram em 2007 o processador Nios da Altera em FPGA, comparando duas abordagens: somente *software* e *software* mais instrução customizada. A instrução customizada calcula uma célula da matriz por ciclo. Células de processamento foram combinadas num *grid* de 8 por 8 (64 células), resultando em 24,5 MCUPS (aceleração de 160 em relação à abordagem somente em *software*). Foram usados FPGAs EPS140. Estimado que, com placas estado-da-arte, o *grid* poderia chegar a 1.000 por 1.000 e o desempenho a até 23,8 GCUPS.

Steven Guccione e Eric Keller, funcionários da Xilinx, propuseram em 2007 (GUCCIONE; KELLER, 2007) uma abordagem também baseada em Lipton e Lopresti. Os elementos de processamento utilizam somente 2 bits para representar a diferença de pontuação entre células sucessivas na matriz. A acumulação da pontuação dos alinhamentos deve ser feita por uma agente externo. Ao invés de ser usado VHDL ou Verilog foi usado JBits, um conjunto de ferramentas e APIs em Java da Xilinx, para a família Virtex. Segudo o texto, JBits foi usado para tirar-se vantagem em diversas oportunidades de customização do circuito em tempo de execução. Usando FPGAs Virtex XCV1000-6, couberam 4.000 elementos de processamento (de 2 bits) a 188 MHz, totalizando 750 GCUPS. Se a Virtex II XC2V6000-5 fosse usada, calculam que teriam 11.000 elementos de processamento a mais de 280 MHz, totalizando 3.200 GCUPS.

Tratando-se de soluções comerciais, o sistema *HyperSeq* (HARPER, 2007) foi criado pela empresa Adaptive Genomics Corporation. A solução é composta de um *front-end* baseado em Linux e um ou mais módulos de *hardware* reconfigurável (com FPGAs). Cada módulo de *hardware* é montado em um gabinete de rack de altura 2U e os módulos comunicam-se por conexões padrão de rede. Não necessitam de mais energia ou refrigeração que servidores *blade*. A expansão pode ser feita adicionando-se mais módulos de *hardware*. Em 2006, o desempenho de-

clarado foi de 139 GCUPS em uma versão simplificada (não considera espaços) e 2,8 GCUPS numa versão mais completa. A placa *Decypher Engine* (SOLUTIONS, 2007) é outra solução comercial, que implementa BLAST, Smith-Waterman e modelos de cadeia oculta de Markov. Essas são configurações diferentes sobre a mesma placa FPGA Decypher Engine. Essa placa foi criada pela empresa Timelogic e funciona conectada, via barramento PCI (33 ou 66 MHz), a servidores Linux, Windows ou Solaris. Descrita como uma solução escalável, cada servidor pode ter diversas placas Decypher Engine, com ganho linear para cada placa. Também ser escalada para vários servidores, cada um com diversas placas. O desempenho em CUPS da Decypher Engine não foi revelado. A aceleração reportada foi de 82 a 144, comparando uma placa Decypher com um cluster de 32 Pentium III 1 GHz.

5.2 Utilizando outras plataformas

5.2.1 Utilizando ASICs

Algumas soluções utilizando VLSI também foram propostas, como BioScan (SINGH *et al.*,) em 1991, o Genematcher2 (produto comercial descontinuado) e as Proclets de Yang (YANG, 2002) em 2002. Essa ultima utiliza uma unidade de processamento (chamada de Proclet) para cada célula da matriz de similaridade e, dessa forma, torna-se muito onerosa, dada as dimensões que a matriz pode atingir.

Além dessas soluções, Andrea Di Blas, David M. Dahle, Mark Diekhans, Leslie Grate e Jeffrey Hirschberg, da Universidade da California, construíram em 1997 o Kestrel (BLAS *et al.*, 2005). Esse pode ser considerado um ASIC com um certo grau de flexibilidade. É uma arquitetura do tipo SIMD, com 512 elementos de processamento de 8-bits cada, montados de maneira linear. Montado em uma

única placa conectada pelo barramento PCI a um computador com Linux ou Windows, esse processador foi construído com foco na análise de seqüências de proteínas e de DNA. No entanto, é programável o suficiente para ser aplicados em áreas da química, processamento de imagens, *machine-learning* e outros. O desempenho medido, em 1997, foi de 400 MCUPS.

5.2.2 Instruções SIMD em processadores de uso geral

Processadores de uso geral tem trazido no seu conjunto de instruções cada vez mais extensões do tipo SIMD (*Single Instruction Multiple Data*). Essas extensões têm aparecido na linha Intel como MMX (*Multimedia Extensions*), MMX2, SSE (*Streaming SIMD Extensions*), SSE2,SSE3 e futuramente SSE4. Na linha AMD, as instruções SIMD são chamadas *3DNow*, *Enhanced 3DNow* e *Extended 3DNow*.

Michael Farrar (FARRAR, 2007) implementou, em 2007, uma variação do algoritmo de Smith-Waterman utilizando instruções SSE2. Os processadores usados foram Intel Xeon Core2Duo 2 GHz. A aplicação não tira vantagem dos dois núcleos do processador (por ser *monothread*). Registradores SIMD de 128-bits foram divididos em 16 células de 8 bits. Uma instrução pode operar, então, em 16 células em paralelo. O desempenho foi de 2,5 GCUPS em média e 3 GCUPS de pico.

O Cell (BIOINFORMATICS, 2007) é uma solução comercial da CLC Bioinformatics que usa instruções SIMD de processadores convencionais. Segundo a empresa, a busca de nucleotídeos teve uma aceleração de 85 vezes e a busca de proteínas de 45 vezes. Utilizando um Intel Core2Duo 2,17 GHz, foi alcançado 9,3 GCUPS.

TABELA 5.1: Comparação de desempenho CPU-GPU. (GFlops baseado em operações multiplicação-soma) (VOSS *et al.*, 2005)

Processador	**Tipo**	**GFLOPS**	**Vazão**
Pentium 4 - 3.4 GHz	CPU	13,6	5.96 GB/s
NVIDIA GeForce 6800 Ultra	GPU	51,2	32,7 GB/s
ATI Radeon X800 XT	GPU	66,6	33,4 GB/s

TABELA 5.2: Taxa de crescimento do número de transistores em CPUs e GPUs (VOSS *et al.*, 2005)

Processador	**Taxa Anual**	**Taxa na década**
CPU	1,5	60
GPU	>2	>1000

5.2.3 GPGPU - *General Purpose Computing on Graphics Processing Units*

Aréa que utiliza GPUs (*graphical processing units*), vendidas em placas de vídeo de mercado para realizar computação em geral. Devido à indústria de jogos e aplicações 3D, tem sidos lançados processadores gráficos cada vez com maior freqüência de operação, grau de paralelismo, memória interna e taxa de transferência para a memória principal. Os principais fabricantes de processadores gráficos são NVIDIA (CORPORATION, 2007) e AMD, divisão "*Graphics and Video Processors*", antiga empresa ATI (AMD, 2007). A tabela 5.1 mostra uma comparação de desempenho entre CPUs e GPUs.

A tabela 5.2 mostra o crescimento do número de transistores em GPUs e em CPUs (VOSS *et al.*, 2005).

SWBoost (Smith-Waterman Boost) (GENBOOST, 2007) é uma solução da Genboost, uma *joint-venture* entre a empresa Elaide e o BMR-Genomics, da Universidade de Padova, Itália. Em 2007, construíram essa solução apenas com GPUs de prateleira, obtendo um desempenho razoável a baixo custo. Segundo os auto-

res, se utilizado um cluster com placas gráficas, o ganho de desempenho é linear, multiplicado pelo número de máquinas. Não especificado um limite para tal. O desempenho alcançado variou entre 5 e 10 GCUPS.

O PheeGee (*Phenotype Genotype Exploration on a Desktop GPU Grid*) (SINGH *et al.*, 2007) propõe a criação de um grid de computadores voluntários, onde as tarefas computacionalmente intensas são realizadas nas GPUs dos nós do grid. Considera que muitas pessoas já tem GPUs modernas instaladas em seus micros, e que a maioria destas pessoas manterá suas GPUs atualizadas com os novos lançamentos. Isso reúne um grande poder de processamento a um custo quase zero. É usado o middleware de grid BOINC (*Berkeley Open Infrastructure for Networked Computing*). Foram usadas GPUs NVIDIA GeForce (8800GTX, 7900GTX, 7950GT e 7800GTX). Usando somente uma GPU NVIDIA GeForce 8800GTX, o pico de desempenho foi 897 MCUPS (*speed-up* de 20 em relação a um Pentium 4 de 3 GHz). Um grid com 6 GPUs atingiu 3,2 GCUPS (*speed-up* de 72) e um com 10 GPUs atingiu 4,5 GCUPS (*speed-up* de 101).

6 Análise e Descrição da Proposta

Considerando o embasamento dado nos capítulos anteriores, para implementar um sistema de alinhamento local, é proposta neste texto, e foi prototipada, uma implementação baseada em *hardware* dedicado. O uso de *hardware* dedicado permite a construção de um arranjo de processadores em um único *chip*. Esses elementos de processamento irão cooperar, trocando informações e executando de forma paralela o algoritmo de alinhamento local de Smith e Waterman. A execução em paralelo reduz o tempo necessário para a execução do algoritmo. Neste capítulo, é descrita a proposta do sistema e são analisadas questões pertinentes a esta. Os resultados obtidos com a prototipação, incluindo o desempenho, podem ser vistos no próximo capítulo.

6.1 Algoritmo escolhido

O algoritmo de alinhamento local de Smith-Waterman foi escolhido para implementação, com simplificação para a penalidade de espaços juntos. A simplificação é que a penalização (pontuação negativa) para o uso de espaços (*gaps*) será sempre a mesma, independente do espaço ser ou não antecedido por outro espaço (na versão original a penalidade para os espaços varia, se estes ocorrem

sucessivamente (GOTOH, 1982)). O esquema de pontuação escolhido, para cada coluna do alinhamento, é:

- +1 para bases iguais pareadas (*match*)
- -1 para bases diferentes pareadas *mismatch*
- -2 para uso de um espaço (*gap*) e uma base

6.2 Arquitetura do sistema

O sistema é composto por *hardware* e *software*. A parte mais computacionalmente intensa é feita em *hardware*: a matriz de similaridade é computada e os "melhores" alinhamentos (incluindo o ótimo) são determinados. O conceito de melhores alinhamentos será, neste contexto, melhor definido abaixo. Em *software*, os alinhamentos indicados por *hardware* são reconstruídos e então visualizados.

A parte em *hardware* é composta, essencialmente, por um conjunto de processadores, implementado em um único *chip*. Esses estão dispostos em um arranjo do tipo sistólico unidirecional (LIPTON; LOPRESTI, 1985) . Como vimos, na aplicação desse tipo de arranjo, cada elemento de processamento é responsável pelo cálculo de uma coluna da matriz de similaridade.

Diferente do que se poderia esperar, é proposto que a saída do *hardware* não seja a matriz de similaridade. Ao invés disto, a saída será composta somente de células selecionadas da matriz. Como o objetivo é encontrar os melhores alinhamentos possíveis, as células mais interessantes da matriz são as que apresentam a maior pontuação. Estas serão selecionadas pelo *hardware* para serem comunicadas ao meio externo. Essa estratégia deve-se a um problema de "gargalo de comunicação", esperado se fossemos comunicar toda a matriz de similaridade. Esse assunto será melhor detalhado na próxima seção.

6.2.1 O problema

Com os processadores calculando a matriz de similaridade poderíamos considerar, em uma primeira abordagem, comunicar toda esta matriz ao meio externo. No entanto, essa abordagem implicaria na necessidade de uma vazão grande de saída de dados, considerando os recursos disponíveis hoje. Será explicado abaixo como, comunicando-se toda a matriz, o volume de dados gerado seria o limitante do desempenho global do sistema.

O relativamente grande volume de dados gerado vem do fato que cada processador pode, a cada ciclo de relógio, computar uma célula da matriz de similaridade. Exemplificando, supondo que sejam utilizados 10 *bits* para representar a pontuação de cada célula. Com 10 *bits* poderíamos representar uma pontuação de até 1023 ($2^{10} - 1$). Supondo, agora, que tenhamos 512 processadores no *chip*. O número de bits de dados gerados pelo *hardware* a cada ciclo de relógio seria, então, 10 (*bits* por processador) vezes 512 (processadores). Ou seja, 5120 *bits* de dados de saída por ciclo de relógio. Para poder dar saída a estes dados ao mesmo passo em que fossem gerados, o circuito integrado que implementa esse conjunto de processadores teria que ter 5120 sinais (pinos) de saída. Essa necessidade de 5120 pinos de comunicação iria inviabilizar o projeto, tanto em termos de não estarem disponíveis componentes com tantos pinos de saída, quanto no fato de que, se tais componentes estivessem disponíveis, grande seria a dificuldade de conectá-los a barramentos padrão de dados, como PCI ou USB.

Uma posível solução para adequar o volume de dados produzidos à vazão de saída é a introdução de ciclos de espera no processamento. Periodicamente, os processadores iriam ficar ociosos, esperando a saída dos dados já calculados. Quanto maior for a proporção entre a capacidade de gerar dados e a vazão de saída, maior seria o número de ciclos de espera obrigatoriamente inseridos. Esses períodos de ociosidade dos processadores fariam com que o sistema trabalhasse

com desempenho reduzido. Assim, na abordagem de comunicar a matriz de similaridade, observamos que a vazão de saída seria o limitante do desempenho do sistema como um todo. Na busca por soluções que permitissem um maior desempenho, alternativas foram consideradas e uma foi escolhida, conforme veremos na próxima seção.

6.2.2 Solução proposta

Como alternativas para evitar o gargalo de comunicação, foram consideradas a compressão dos dados e formas diversas de representação da matriz (CARVALHO, 2003). Observamos que, mesmo que a matriz fosse comunicada sem dificuldades, externamente ao *chip*, teria que ser feita uma busca na matriz, para que fossem determinados os alinhamentos (células) de interesse. Apesar de menos complexo que o cálculo em si da matriz, a determinação de máximos levaria a uma leitura em um espaço de complexidade quadrática (em relação ao tamanho das seqüências). Mesmo que pudessemos comunicar sem dificuldades toda a matriz, um trabalho de complexidade quadrática teria que ser executado em *software*, tomando tempo e impactando negativamente no desempenho global do sistema.

De outra maneira, para remover o gargalo de comunicação e a limitação de desempenho associada, é proposto que a matriz de similaridade não seja totalmente comunicada ao meio externo. Ao invés disto, os melhores alinhamentos são determinados ainda em *hardware*. Isto é feito pela identificação de máximos locais na matriz de similaridade. Somente esses máximos locais, que representam os alinhamentos selecionados, são comunicados pelo *hardware*, reduzindo o volume de dados de saída e removendo o gargalo de comunicação.

Na matriz de similaridade, cada célula representa um alinhamento possível. As células com os maiores valores na matriz correspondem aos alinhamentos com a maior pontuação possível. Os máximos locais representam os melhores

alinhamentos possíveis de serem construídos, naquela localidade da matriz. No contexto desta proposta, os máximos locais são referentes a uma coluna da matriz. Conseqüentemente, são referentes a um processador. Assim, nesta proposta, os máximos locais são mantidos e apresentados por processador, correspondendo a máximos de colunas da matriz.

No conjunto dos máximos locais comunicados está sempre incluso o máximo global (maior máximo local), que representa um alinhamento ótimo entre as seqüências. Assim, o sistema sempre apresenta, dentre os alinhamentos selecionados, o ótimo (ou um ótimo).

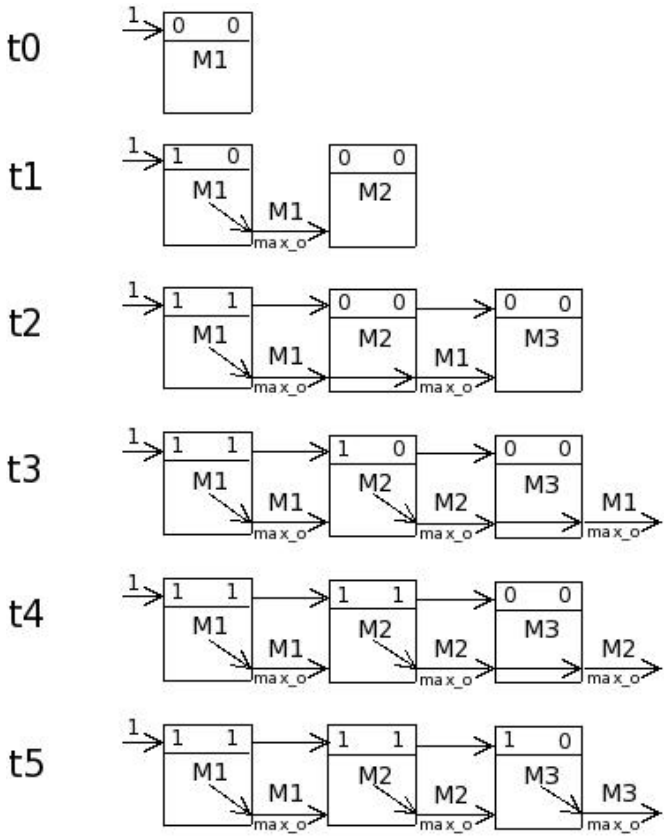

FIGURA 6.1: Mecanismo de apresentação dos máximos

O mecanismo proposto para apresentação dos máximos é ilustrado na figura 6.1. A figura representa um vetor sistólico, com 3 processadores, após o término

do cálculo da matriz, durante 6 ciclos de relógio. A partir do ciclo t3, os máximos locais (M1,M2,M3) de cada processador são observados como saída do terceiro processador e, conseqüentemente, saída do vetor como um todo. Para que a exibição dos máximos locais aconteça corretamente, através do último processador à direita, um sinal que pode ser interpretado como "apresente seu máximo" é propagado, com dois ciclos de relógio de diferença, entre cada processador. Na figura, os seguintes eventos podem ser observados ao longo do tempo:

- **t0:** o cálculo da matriz já terminou. O primeiro processador tem seu máximo local armazenado (M1). O sinal que "solicita" que este processador apresente seu máximo é ativado (número 1, na parte superior esquerda do processador);
- **t1:** devido ao sinal ativado no ciclo anterior, o primeiro processador apresenta seu máximo. O valor M1 é informado na saída `max_o` do processador. O sinal para apresentação do máximo é armazenado numa memória interna do primeiro processador (número 1, acima da linha, dentro do processador);
- **t2:**
  - o máximo do primeiro processador (M1) é copiado pelo segundo, da entrada para saída `max_o`;
  - o sinal para apresentar o máximo é copiado dentro do primeiro processador, da primeira para a segunda memória interna;
- **t3:**
  - o máximo do primeiro processador (M1) é copiado pelo terceiro, da entrada para saída `max_o`. Este valor M1, sendo uma saída do terceiro processador, constitui a primeira saída do vetor sistólico;

  - o sinal para apresentar o máximo (número 1) chega ao segundo processador. Em resposta, este apresenta na saída `max_o` seu próprio máximo local, M2.

- **t4:**

  - o máximo do segundo processador (M2) é copiado pelo terceiro, da entrada para saída `max_o`;
  - o sinal para apresentar o máximo é copiado dentro do segundo processador, da primeira para a segunda memória interna.

- **t5:**

  - o sinal para apresentar o máximo (número 1) chega ao terceiro processador. Em resposta, este apresenta na saída `max_o` seu próprio máximo local, M3.

Vemos que, a partir do ciclo t3, os máximos locais de cada processador são apresentados como saída do vetor sistólico. O sinal que "solicita" a apresentação dos máximos (número 1 na figura), chega com dois ciclos de diferença entre cada processador. Este sincronismo permite a correta apresentação dos máximos locais, através do último elemento da direita. O retardo foi obtido com o uso de duas memórias em série, mostradas na figura como os números acima da linha dentro de cada processador.

Apresentados os máximos, existem duas possibilidades: todos são comunicados ao meio externo ao *chip*, ou uma lógica, ainda em *chip*, pode compará-los, selecionando somente os maiores para comunicar. Na implementação deste projeto, todos os máximos locais são comunicados, no entanto existe uma lógica adicional que ordena essa saída. O maior máximo local (o máximo global) é o primeiro a sair, depois a saída segue em ordem decrescente de pontuação. Assim

as primeiras saídas são as que representam o melhores alinhamentos. Comunicados os máximos locais, os alinhamentos que estes representam podem ser reconstruídos e visualizados por um usuário do sistema. O tempo necessário para a reconstrução não é grande, pois é proporcional ao tamanho das subseqüências presentes no alinhamento (e não das seqüências completas).

6.2.3 Implementação da solução

Nesta proposta, podemos dividir cada processador do arranjo sistólico essencialmente em duas partes: lógica de cálculo do valor de cada célula e lógica de manutenção e apresentação do valor máximo local de cada processador.

6.2.3.1 Lógica de cálculo do valor de cada célula

Esta lógica, que está contida em cada processador, é responsável pelo cálculo do valor das células, através da avaliação da função de máximo da equação de recorrência. As penalidades foram escolhidas conforme abaixo:

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + 1 & match \\ S_{i-1,j-1} - 1 & mismatch \\ S_{i-1,j} - 2 & gap \\ S_{i,j-1} - 2 & gap \end{cases}$$

A implementação dessa parte do elemento de processamento foi feita conforme descrito na seção 4.4. São usados três registradores para memorizar os valores das células vizinhas (esquerda, acima, diagonal superior-esquerda), necessários para o cálculo de cada célula. Outro registrador é usado para memorizar a base-do-banco de dados sendo comparada no momento. Aos valores das células vizinhas, presentes nos registradores, são somadas constantes positivas ou negativas, dependendo se ocorre um *match* (+1), *mismatch* (-1) ou *gap* (-2) entre as bases.

Feito esse cálculo, conforme a equação, os resultados são submetidos a uma função de máximo. O resultado é o máximo entre as 3 células vizinhas citadas, somados ou subtraídos de constantes de acordo com o esquema de pontuação e com as bases envolvidas. Finalmente, para concluir o cálculo do valor da célula, resta verificar se este máximo resultante é negativo. Se for, é substituído por zero, por ser alinhamento local onde não devem existir número negativos na matriz.

6.2.3.2 Lógica de pontuação máxima local

Esta lógica tem a finalidade de memorizar e apresentar o máximo valor de pontuação calculado por cada processador. Esses máximos locais, representantes dos melhores alinhamentos, serão comunicados como saída do *hardware*. A apresentação dos máximos é coordenada por um sinal de entrada em cada processador. Esse mecanismo foi descrito em detalhes na seção 6.2.2. Além do valor das células que são máximo local, a localização desta é também memorizada e comunicada. Memorizando a pontuação e as coordenadas dos máximos locais, temos as informações necessárias para a posterior localização dos melhores alinhamentos, sem o custo inerente ao armazenamento e/ou à comunicação de toda a matriz de similaridade. Nessa implementação, a célula é localizada na matriz utilizando como coordenadas:

- a coluna onde ocorre: vimos que esta coincide com o número do processador que a produziu. Ou seja, se o elemento de processamento que estiver informando esse máximo for o primeiro, então essa célula está na primeira coluna da matriz. Assim, essa coordenada não precisa ser armazenada pelos processadores junto à pontuação máxima, pois coincide com a posição deste processador;

- a antidiagonal onde ocorre: esta informação não pode ser deduzida, sendo armazenada por cada processador junto à pontuação do máximo local, para

posterior relocalização da célula.

Então, quando um processador for armazenar um novo máximo local, deve memorizar, junto com a pontuação ocorrida, a antidiagonal onde ocorre essa célula. O número da antidiagonal, mais o número da coluna da célula (deduzida da posição do processador) são as informações que serão utilizadas para posterior relocalização da célula.

Para termos uma forma eficiente de cada processador saber o número da antidiagonal de cada célula, é proposto o seguinte mecanismo. Como vimos, cada antidiagonal é calculada em um ciclo de relógio. A primeira antidiagonal é calculada no primeiro ciclo após o *reset* (tempo 1), a segunda no segundo ciclo (tempo 2) e assim sucessivamente. Então, conclui-se que o número da antidiagonal onde uma célula ocorre coincide com o número de ciclos passados, desde o *reset*, até o momento de sua produção. Nesta implementação, para cada processador poder armazenar, junto ao máximo local, o tempo de produção deste, utiliza-se um contador de ciclos de relógio, único para todos os processadores. O valor atual de ciclos de relógio é uma entrada para todos os processadores. Quando cada elemento for memorizar um novo máximo, memoriza também o ciclo de relógio corrente àquele instante. Como vimos, o ciclo de relógio corresponde à antidiagonal em produção naquele instante.

Com a coluna (deduzida da posição do processador) e a antidiagonal (armazenada pelos processadores), podemos localizar cada célula onde ocorreu um máximo local. Assim, pontuação, coluna e antidiagonal, dos máximos locais produzidos, são as informações que irão compor a saída do *hardware*.

6.2.4 Quando apresentar os máximos locais ?

Além da lógica para manter o máximo local, cada processador deve ter um sinal de entrada que indica quando apresentar seu máximo. Duas abordagens são

possíveis para a questão de quando apresentá-los:

- ao final de todo o processamento (ao final da entrada e somente uma vez);
- periodicamente, permitindo mais de 1 ciclo de coleta de máximos, gerando mais informações sobre bons alinhamentos.

A apresentação ao final do processamento é uma abordagem mais simples. O sinal de início da apresentação dos máximos pode ser vinculado ao sinal que indica o final da entrada de dados para o sistólico. Quando a seqüência do banco-de-dados termina, o sinal de apresentação de máximo do primeiro processador à esquerda é ativado. Este apresenta seu máximo e, com dois ciclos de atraso, ativa o sinal de apresentação de máximo do processador à sua direita (o segundo). Isto permite que o máximo local de todos os processadores cheguem ao processador mais à direita, daí sendo saída do sistólico. Maiores detalhes sobre este mecanismo são dados na seção 6.4, que descreve os componentes de *hardware*.

Na outra abordagem possível, a leitura dos máximos pode ser feita periodicamente. O sinal que indica início da apresentação dos máximos locais pode ser ativado após um determinado intervalo de tempo, por uma lógica adicional. Sempre que esse sinal é ativado, inicia-se um ciclo no qual todos os processadores fazem a apresentação de seus máximos. O interessante desta abordagem é que, após apresentar seu máximo local, cada elemento de processamento pode zerar seu registrador interno de máximo. Isto segmenta a matriz, de forma que teremos informação sobre mais máximos locais desta. Por exemplo, a cada 500 bases da seqüência de entrada, podemos ativar a apresentação dos máximos locais. Dessa forma, o máximo local de cada processador será referente às últimas 500 células computadas, e não sobre todas as células computadas por este processador. A principal vantagem dessa abordagem é o maior número de informações que teríamos como saída, com mais máximos locais (e bons alinhamentos) identificados.

No entanto, em ambas as abordagens (apresentação ao final ou periódica) as maiores pontuações da matriz serão recuperadas. A coleta periódica é vantajosa quando for desejada a recuperação de mais alinhamentos, além do número de processadores. Por exemplo, supondo que tenhamos 500 processadores. Coletando os máximos ao final do trabalho teremos informação sobre 500 máximos locais da matriz. Se a opção for por coleta periódica e, por exemplo, coletando 10 vezes durante o processamento, teremos informação sobre 5.000 (500×10) máximos locais. Obviamente, tanto o máximo global quanto as 500 células com maior pontuação global vão estar presentes nos dois conjuntos, tanto no de 500 quanto no de 5.000. Nesta implementação e prototipação foi escolhida a coleta dos máximos locais somente ao final de todo o processamento do sistólico, ligeiramente mais simples de ser implementada e apresentando um número satisfatório de máximos locais, além do máximo global. A maior simplicidade de implementação reverte-se em benefício, pois, com menos lógica utilizada, temos mais espaço disponível em *chip*, o que permite a criação de um maior número de processadores.

6.3 Entradas e saídas do sistema

6.3.1 Entradas

6.3.1.1 Seqüências

Conforme vimos, na abordagem utilizando processadores em um arranjo sistólico unidirecional, cada elemento é responsável pelo cálculo de uma coluna da matriz de similaridade. Assim, cada processador sempre está computando células referentes a uma mesma base (C,G,A ou T) da seqüência de consulta. As bases da seqüência de consulta são, então, fixas para cada processador, não constituindo um dado de entrada.

Já a seqüência do banco-de-dados, conforme vimos, é um dado de entrada para o arranjo sistólico. As bases entram no sistólico pelo processador mais à esquerda. Desse elemento, são repassadas ao seguinte, deste ao próximo e assim por diante até que todas as bases, uma por vez, cheguem ao último processador à direita. Como são quatro as possibilidades de bases nitrogenadas (C,G,A ou T), são usados 2 *bits* para representá-las. Além destes, um sinal de 1 *bit* é utilizado para indicar o final das bases.

Na prototipação realizada, a seqüência do banco-de-dados foi pré-carregada em uma memória RAM, no mesmo *chip* onde estão os processadores. Foi utilizada uma placa FPGA do tipo "stand-alone", ou seja, desvinculada de um microcomputador. Ao iniciar o processamento, a seqüência do banco-de-dados flui, da memória no *chip* onde foi pré-carregada para o arranjo sistólico. Como vimos, um sinal de 1 *bit* é utilizado para sinalizar o final da seqüência. Assim, a seqüência do banco-de-dados é uma entrada do conjunto de processadores. Contudo, na prototipação feita, não constituiu uma entrada do *hardware* como um todo, pois foi pré-carregada em memória.

6.3.1.2 Demais entradas

Conforme veremos na próxima seção, a exibição da saída do *hardware* é feita por partes. Um sinal de entrada de 1 *bit* é usado, para o usuário requisitar a exibição da próxima parte. Esse sinal está ligado a um botão de apertar, presente na placa de prototipação. Cada vez que o usuário pressiona esse botão, está solicitando que o *hardware* apresente a próxima saída.

Um sinal de *reset* também é necessário para a correta inicialização do circuito. Assim que o sinal de *reset* é ativado e desativado, o processamento tem início. A primeira base da seqüência do banco-de-dados é lida pelo processador mais à esquerda. As demais são lidas na seqüência. Todos os sinais são síncronos em

relação ao relógio de referência, que também é uma entrada.

6.3.2 Saídas

Nesta proposta, a opção é obter o máximo local produzido por cada processador. Dentre eles, o máximo global da matriz que representa o alinhamento ótimo. Esses máximos locais serão a saída do sistema. Um sinal de 1 *bit* é utilizado para sinalizar o início da saída. Junto à ativação desse sinal, teremos disponíveis na saída do sistólico informações sobre o primeiro máximo local (o máximo do primeiro elemento à esquerda). Essas informações são a pontuação da célula e sua localização. Como vimos, a localização da célula será informada pelos processadores em termos da antidiagonal onde ocorreu, sendo a coluna onde ocorre dedutível, pois corresponde ao número do processador que a apresenta.

Na prototipação realizada, os máximos locais (pontuação e localização) são apresentados ao final do processamento e armazenados em memória RAM, ainda no mesmo *chip*. Na primeira posição de RAM, teremos o máximo local do primeiro processador (à esquerda), na segunda posição o máximo do segundo processador e assim por diante.

Ao final desse processo, quando todos os máximos locais estiverem armazenados, uma lógica adicional será responsável por fazer a leitura destes. Determinado por comparação o máximo global, essa lógica irá apresentá-lo como primeira saída do *hardware*. A partir deste, os demais máximos locais serão apresentados, em ordem decrescente de pontuação. Assim, os melhores alinhamentos encontrados, representados pelo valor e localização das células com os máximos locais, são as saídas propostas do *hardware*.

6.3.2.1 Primeira saída - valor do máximo global

A primeira saída apresentada pelo *hardware* será a pontuação do máximo global da matriz de similaridade, que corresponde à pontuação de um alinhamento ótimo. Na figura 6.2, podemos ver a placa de prototipação usada exibindo a pontuação do máximo global calculado em um teste. A exibição é feita em hexadecimal, em um *display* de sete segmentos. Na figura, temos 1F em hexadecimal que corresponde a 31 em decimal. Para essas duas seqüências de teste, 31 é, então, a pontuação do alinhamento local ótimo.

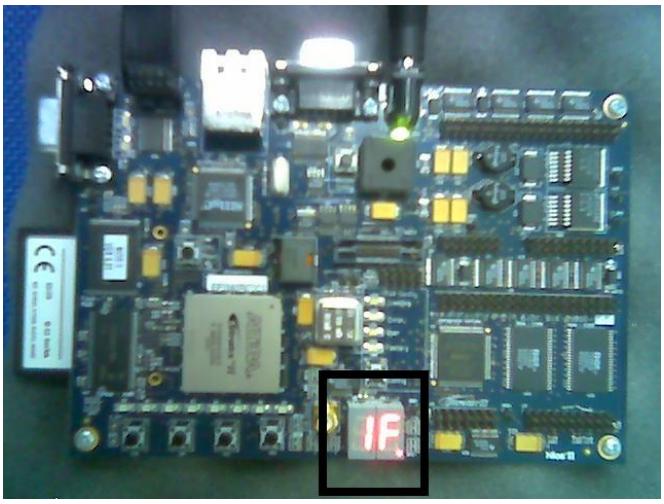

FIGURA 6.2: Primeira saída do *hardware* - pontuação do alinhamento ótimo

6.3.2.2 Segunda saída - coluna do máximo global

Na figura 6.3, vemos que após o usuário apertar um botão na placa, o *display* exibe a próxima informação: a coluna de ocorrência do máximo global. No caso, 7B em hexadecimal equivalente a 123 em decimal.

A informação da coluna do máximo é produzida da seguinte forma. Vimos que, após a apresentação dos máximos locais pelos processadores, estes são gravados em uma memória RAM no mesmo *chip*. Por essa lógica de gravação, a posição

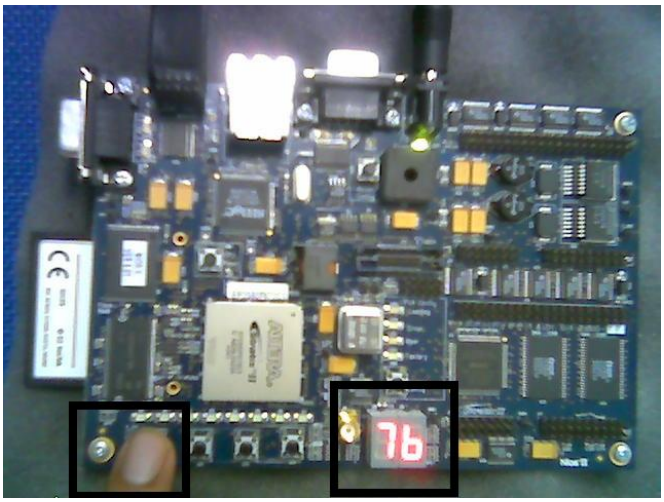

FIGURA 6.3: Segunda saída do *hardware* - coluna de término do alinhamento ótimo

de cada máximo na memória RAM coincide com o número do processador que a produziu. Ou seja, na primeira posição em memória encontramos o máximo local do primeiro processador, na segunda posição o máximo do segundo e assim por diante. Por sua vez, o número do elemento que produziu um máximo coincide com a coluna na matriz onde este ocorre. Assim, o máximo local na primeira posição de memória RAM é o máximo local do primeiro elemento, que é o máximo local da primeira coluna da matriz e, assim, sucessivamente. Dessa forma, pode-se deduzir a coluna da matriz onde ocorre um máximo local, pela posição deste na memória RAM.

Utilizando como exemplo a situação da figura 6.3, vemos que nesse caso o máximo global foi observado na posição 123 da memória RAM. Isto implica em que o processador que o produziu foi o centésimo vigésimo terceiro. Assim, sabemos que esse máximo global informado ocorreu na coluna 123 da matriz de similaridade.

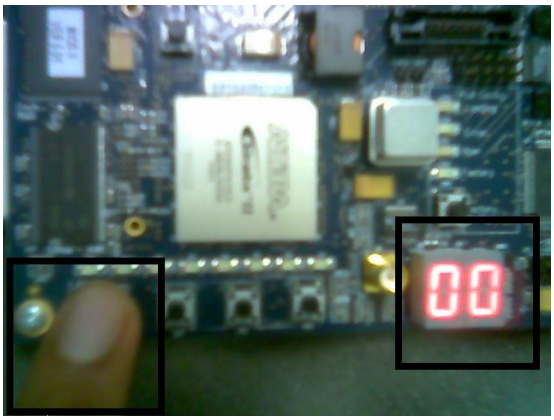

FIGURA 6.4: Terceira saída do *hardware* - antidiagonal de término do alinhamento ótimo - *byte* mais significativo

6.3.2.3 Terceira e quarta saída - antidiagonal do máximo global

Após estas duas informações, valor e coluna, serem exibidas, temos a exibição da última informação referente ao máximo global. Para possibilitarmos a relocalização da célula na matriz, foi armazenado pelo processador e é exibido o número da antidiagonal onde este máximo local ocorreu.

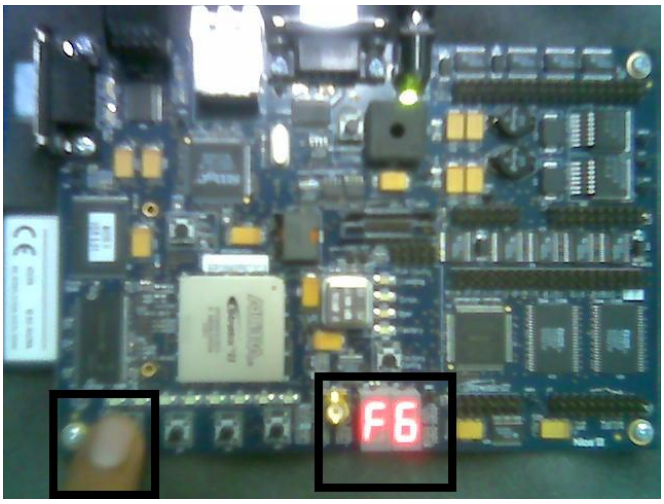

FIGURA 6.5: Quarta saída do *hardware* - antidiagonal de término do alinhamento ótimo - *byte* menos significativo

Como vimos na seção 6.2.3, o número da antidiagonal corresponde ao ciclo de relógio no momento da produção da célula. Cada processador armazena esse

tempo de produção (antidiagonal), junto com o valor do máximo local. Quando o máximo de cada processador é informado para os vizinhos e destes até a memória RAM, o número da antidiagonal da célula sempre acompanha a pontuação, permitindo sua relocalização na matriz. Nas figuras 6.4 e 6.5, vemos a exibição da antidiagonal onde ocorreu o máximo global. Pelo número de *bits* usados, a representação é feita em dois bytes separados. No caso, 00F6 em hexadecimal equivale a 246 em decimal.

Completas as informações (valor, coluna e antidiagonal), temos todos os dados referentes ao máximo global, o primeiro máximo local informado. A partir das coordenadas indicadas, podemos, em *software*, reconstruir o alinhamento ótimo. Essa reconstrução é feita em tempo proporcional ao tamanho das subseqüências presentes no alinhamento, e não das seqüências completas. Após a reconstrução, o alinhamento pode ser visualizado por um usuário do sistema.

6.3.2.4 Próximas saídas - demais máximos locais

A partir desse momento, onde foram apresentadas todas as informações sobre o máximo global, ao apertar novamente o botão na placa, o *hardware* irá informar os próximos melhores alinhamentos (sempre em triplas: valor,coluna e antidiagonal). Se o próximo valor exibido for novamente o mesmo, conclui-se que existem, no mínimo, dois alinhamentos ótimos (com o mesmo valor de pontuação). Se o próximo valor exibido for diferente do primeiro (máximo global), então deve ser um maior máximo local. Sucessivamente são exibidos todos os máximos locais, em ordem decrescente de pontuação e a partir do máximo global, enquanto se aperte os botões na placa ou até que todos sejam exibidos.

6.4 Descrição dos componentes *hardware*

Nesta seção serão descritos individualmente os componentes de *hardware* propostos. A descrição será feita em termos de suas entradas e saídas, lógica interna e estrutura interna. Para nomear os sinais de entrada e saida, o seguinte padrão é utilizado:

- sufixo "_i" (de input) para nomear os sinais de entradas;
- sufixo "_o" (de output) para nomear os sinais de saída.

Além do componente de mais alto nível da aplicação, descrito na próxima seção (6.4.1), um componente adicional, de *interface*, foi criado. O objetivo desse componente é, exclusivamente, a adaptação da saída de dados aos limites da placa de prototipação utilizada. A saída proposta para o circuito da aplicação são triplas (pontuação, coluna e antidiagonal), referentes a cada máximo local. No entanto, a placa de prototipação usada possui somente dois displays de sete segmentos, insuficientes para exibição simultânea dos componentes da tripla.

Assim, um componente de *interface*, externo ao de mais alto-nível da aplicação, foi criado. Esse componente divide a saída, exibindo-a no *display* em quatro partes. As partes são: pontuação, coluna e antidiagonal, esta última separada em primeiro e segundo *byte*. Uma parte é exibida por vez, nos *displays* de sete segmentos da placa. Cada vez que o usuário aperta um botão na placa, a próxima parte é mostrada. Somente quando o usuário já apertou o botão quatro vezes (e as quatro partes foram exibidas), informações sobre o próximo máximo local são solicitadas ao componente de mais alto nível da aplicação (sinal *max_proximo_i* ativado, veja 6.4.1). A nova saída, que são informações sobre o próximo máximo local, é recebida pelo componente de *interface* e novamente dividida e apresentada em quatro partes.

Se houvessem *diplays* suficientes, as informações da saída (triplas) poderiam ser exibidas simultâneamente e esse componente de *interface* não seria necessário. Por ser dispensável em outras situações e de implementação trivial, esse componente de *interface* não é considerado essencial para o projeto e não será descrito em detalhes como os demais.

6.4.1 Componente de mais alto nível da aplicação

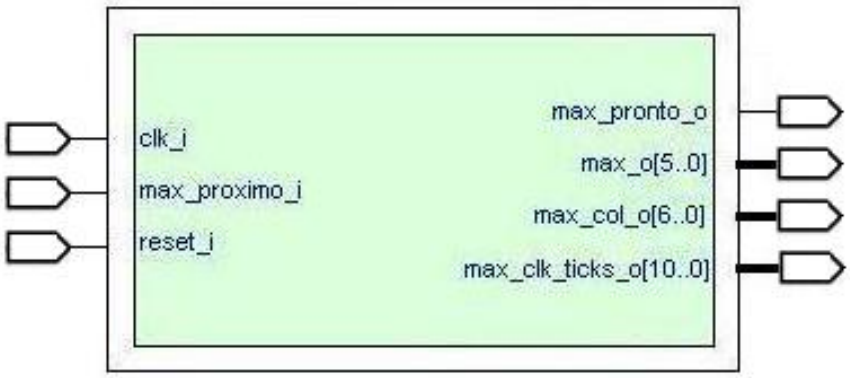

FIGURA 6.6: Componente de *hardware* de mais alto nível

Esse componente é o de mais alto-nível pois engloba todos os demais (exceto o componente de adaptação à placa (ou de *interface*) que divide a saída em partes, se esse for necessário). Na figura 6.6 temos uma representação do tipo "caixa-preta" do circuito de mais alto nível da aplicação proposto. As saídas deste componente, como veremos com mais detalhes abaixo, são os máximos locais (pontuação e coordenadas) apresentados pelos processadores. Esses máximos são exibidos em ordem decrescente de pontuação, partindo-se do máximo global. O próximo máximo é pedido acionando-se o sinal `max_prox_i`.

Entradas:

- `clk_i`: relógio de referência para todos os componentes do circuito;

- `reset_i`: sinal de inicialização (*reset*) do circuito. Ao ser ativado é atribuído um valor inicial para as memórias, levando o circuito a um estado inicial;

- `max_proximo_i`: cada vez que é acionado, o circuito exibe informações sobre o próximo máximo local (em ordem decrescente e iniciando por um máximo global). Na prototipação realizada, um novo máximo local é pedido quando as quatro partes (valor, coluna e a antidiagonal em 2 bytes) do máximo anterior já foram exibidas. A cada quatro vezes que o usuário aperta o botão na placa, é concluída a exibição de um máximo local e o circuito de *interface* ativa o sinal `max_proximo_i`, pedindo as informações do próximo máximo local.

Saídas:

- `max_pronto_o`: este sinal quando ativado indica o início das saídas. As saídas são os máximos locais de cada coluna (pontuação e coordenadas) e são exibidas nos demais sinais de saída (`max_o`, `max_col_o` e `max_clk_ticks_o`, conforme abaixo). A primeira saída são os dados do máximo global. Depois, a cada ativação de `max_proximo_i`, é exibido o próximo máximo local, em ordem decrescente de pontuação. As informações dos máximos são disponibilizadas pelos sinais que seguem;

- `max_o`: valor da célula do máximo local. Pode ser visto como a pontuação do alinhamento representado pela célula;

- `max_col_o`: coluna onde ocorreu a célula. Usada para podermos localizá-la na matriz;

- `max_clk_ticks_o`: tempo de relógio quando da produção da célula. O tempo de relógio equivale ao número da antidiagonal onde a célula está

presente. A coluna e o número da antidiagonal são as informações utilizadas para localizar a célula e reconstruir o alinhamento que ela representa.

Lógica interna: A seqüência de consulta é fixa e a do banco-de-dados é uma entrada, pré-carregada numa memória do circuito. Quando o sinal de *reset* é ativado e desativado, inicia-se o processamento. A primeira base do banco-de-dados vai para o primeiro processador, no próximo ciclo a segunda vai para este elemento e assim por diante. Quando a seqüência do banco-de-dados termina (e todas as células da matriz foram calculadas), uma lógica interna ativa o início da apresentação dos máximos locais.

Os máximos são armazenados em uma memória RAM. Ao final, uma lógica apresenta-os em ordem decrescente de pontuação. A apresentação inicia-se por um um máximo global (maior pontuação entre os máximos locais). O sinal `max_pronto_o` indica que uma saída do circuito está disponível. Essa saída é composta pelos sinais `max_o`, `max_col_o` e `max_clk_ticks_o`. Esses três componentes são mantidos nas linhas de saída até que a entrada `max_proximo_i` seja ativada, indicando que o usuário já leu essa informação e deseja uma nova. Cada vez que o sinal `max_proximo_i` é ativado, o circuito apresenta o próximo máximo local (pontuação e coordenadas), em ordem decrescente de pontuação, até que todos os máximos locais sejam exibidos.

Estrutura interna:

Na figura 6.7 temos uma representação da estrutura interna do elemento de mais alto nível.

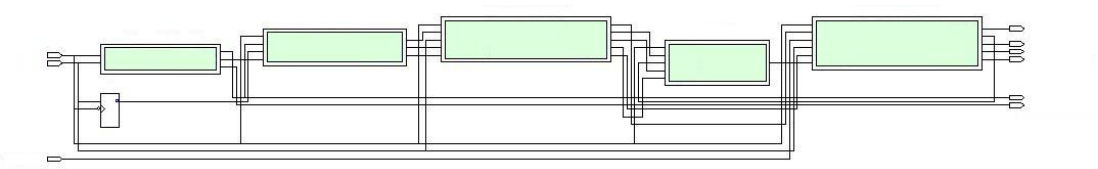

FIGURA 6.7: Componente de *hardware* de mais alto nível (visão interna)

Podemos ver a existência de cinco sub-componentes, os quais dividimos em três partes:

- lógica de entrada: composta pelo primeiro elemento à esquerda. Nesta prototipação, a seqüência do banco-de-dados fica pré-carregada numa memória interna desse componente. Os sinais `bbd_i` (base do banco-de-dados) e `fim_bbd_i` (fim da bases) são gerados por esse componente e são as entradas do conjunto de processadores (seção 6.4.2 abaixo);

- conjunto de processadores: um arranjo sistólico unidirecional composto de processadores, conforme descrito em 6.4.2 e na literatura revisada. Esses processadores implementam em paralelo o algoritmo de Smith-Waterman de alinhamento local. A maior parte da área do chip corresponde a implementação desses elementos de processamento;

- lógica de saída, composta pelos três elementos mais à direita na figura 6.7.

  - o subcomponente mais próximo do conjunto de processadores é responsável por ler a saída deste, ou seja o máximo local de cada processador. Faz duas atividades: determina o máximo global por comparação dos máximos locais e armazena os máximos locais em uma memória RAM.

  - ao centro temos a memória RAM. É escrita com os máximos locais dos processadores pelo componente acima e é lida, para formar a saída do circuito, pelo componente abaixo;

  - o elemento mais à direita da figura 6.7 é o responsável pela saída de dados do circuito como um todo. Suas entradas são os dados na memória RAM (os máximos locais), e o máximo global (maior máximo local, informado pelo primeiro componente desta sublista). Sua saída é a saída do circuito como um todo, ou seja, um máximo local por vez, a partir do máximo global, em ordem decrescente de pontuação.

A descrição completa desse componente, em linguagem de descrição de *hardware* (Verilog), encontra-se no anexo A.1.

6.4.2 Arranjo sistólico de processadores

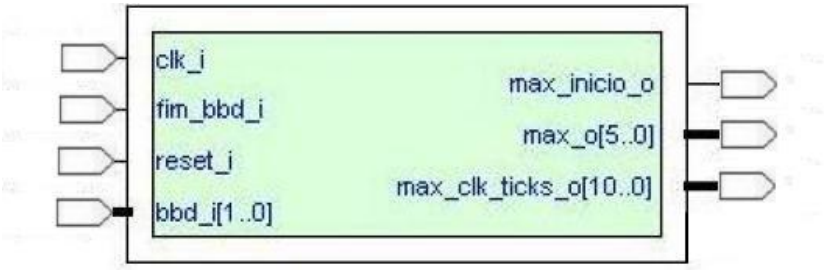

FIGURA 6.8: Componente de *hardware* - arranjo sistólico de processadores

Representado na figura 6.8, o conjunto de processadores recebe como entrada as bases da seqüência do banco-de-dados e calcula a matriz de similaridade. Ao final do processamento, apresenta como saída as células com pontuação máxima de cada coluna. A primeira saída é o máximo local da primeira coluna, a segunda o máximo da segunda e assim sucessivamente. Um sinal de 1 *bit* é usado para sinalizar o início das saídas. Para cada máximo local da saída, é informado o valor e a antidiagonal da célula. A coluna da célula é deduzida do número de ciclos de relógio passados desde o início da saída. No primeiro ciclo, o máximo da primeira coluna está sendo apresentado, no segundo o máximo da segunda coluna e assim por diante.

Entradas:

- `clk_i` e `reset_i`: sinais de relógio e de reset;
- `bbd_i`: as bases do banco-de-dados, lidas uma por ciclo de relógio após o reset. Sinal com 2 *bits* (4 possibilidades: C,G,A ou T);
- `fim_bbd_i`: quando ativo, sinaliza o final das bases do banco-de-dados (final da entrada).

Saídas:

- `max_inicio_o`: sinaliza o início da saída dos máximos locais. Quando esse sinal é ativado, a saída de dados (presente nos sinais abaixo) é valida e apresenta informações do máximo local do primeiro processador. No segundo ciclo, as informações são sobre o máximo local do segundo processador e assim por diante.
- `max_o`: contém o valor do máximo local sendo apresentado;
- `max_clk_ticks_o`: contém a antidiagonal do máximo local sendo apresentado, que coincide com o número de ciclos de relógio passados desde o *reset* até a sua produção.

Lógica interna: O algoritmo de alinhamento de seqüências de Smith-Waterman é implementado em um conjunto de processadores em um arranjo sistólico unidirecional. Cada antidiagonal é calculada em um ciclo de relógio. Cada processador calcula uma coluna da matriz. Os máximos de cada processador (coluna da matriz) são memorizados por estes. Ao final da entrada, sinalizado por `fim_bbd_i`, é solicitado ao primeiro processador da esquerda que apresente seu máximo (sinal `am_i` do processador). O processo de apresentação de máximos começa, com o máximo local do primeiro processador indo para o segundo, daí para o terceiro e assim segue por todo o vetor até que chegue ao último processador (da direita) e daí constitua uma saída do arranjo. O segundo processador, após passar o máximo do primeiro para o terceiro, apresenta seu próprio máximo local para este terceiro. E assim sucessivamente, até que todos os máximos locais saiam do arranjo sistólico, representador por pontuação e coordenadas. Esse mecanismo foi descrito com mais detalhes na seção 6.2.2.

Estrutura interna: Na figura 6.9 vemos uma representação da estrutura interna do conjunto de processadores. Podemos vê-los como um vetor unidirecional. A direção da comunicação entre eles é somente da esquerda para a direita.

Cada processador somente lê dados do seu vizinho à esquerda e somente comunica dados ao vizinho da direita. As entradas do conjunto como um todo estão ligadas às entradas do processador mais à esquerda. As saída do circuito como um todo estão ligada às saidas do processador mais à direita.

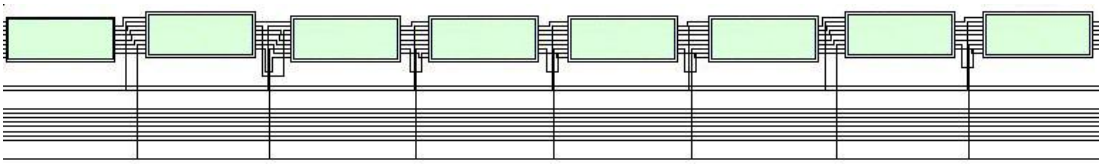

FIGURA 6.9: Componente de *hardware* - arranjo sistólico (visão interna)

A descrição do arranjo sistólico de processadores, em linguagem de descrição de *hardware* Verilog encontra-se no anexo A.2.

6.4.3 Processador

FIGURA 6.10: Componente de *hardware* - processador

Na figura 6.10 vemos a representação do tipo "caixa-preta" de cada processador que compõe o arranjo sistólico descrito acima.

Entradas: Todas as entradas dos processadores (exceto o *clock*) estão ligadas às saídas de seus vizinhos da esquerda. As entradas do primeiro processador (à

esquerda) são as entradas do arranjo sistólico como um todo.

- `clk_i` e `reset_i`: sinais de relógio e de reset. O sinal de *reset* passa por um registrador interno a cada processador, fazendo com que chegue com um ciclo de diferença entre cada processador. Isto é necessário para a inicialização das memórias dos processadores no tempo certo, representando as bordas da matriz (valor 0);
- `bbd_i`: as bases do banco-de-dados, uma por ciclo de relógio, iniciando após o reset. Sinal com 2 *bits* (4 possibilidades: C,G,A ou T);
- `fim_bbd_i`: quando ativo, sinaliza o final das bases do banco-de-dados (final da entrada);
- `col_i`: valor da célula calculada pelo vizinho à esquerda. Será memorizado para o cálculo da próxima célula nesse processador (será o valor de S_{ij-1} durante o cálculo de S_{ij});
- `am_i`: sinal para o processador apresentar sua célula com o valor máximo calculado. Essa saída será apresentada nos sinais `max_o` e `max_clk_ticks_o`;
- `max_i`: pontuação recebida do processador vizinho da esquerda como sendo o máximo local de um vizinho à esquerda (do mais próximo ou de outros);
- `max_clk_ticks_i`: antidiagonal onde ocorreu o máximo local referido no item acima;
- `clk_ticks_i`: relógio global, informa o número de ciclos decorrido após o *reset*.

Saídas: Todas as saídas dos processadores estão ligadas às entradas de seus vizinhos da direita. As saídas do último processador à direita são as saídas do arranjo sistólico como um todo.

- `reset_o`: por este sinal, cada processador passa o *reset* para o vizinho à direita. Como visto, o sinal chega com um ciclo de diferença entre cada processador, garantindo a inicialização das memórias com valor zero no momento adequado;

- `bbd_o`: base do banco-de-dados sendo informada ao processador da direita. O valor presente em `bbd_o` é o mesmo lido em `bbd_i`, com um ciclo de diferença. Assim as bases do banco-de-dados são passadas entre os processadores;

- `fim_bbd_o`: sinaliza o fim do dado acima, as bases do banco-de-dados;

- `valor_o`: valor da célula calculada neste ciclo. Memorizado pelo processador à direita, para este poder calcular uma célula no próximo ciclo (será o valor de S_{ij-1} durante o cálculo de S_{ij} pelo vizinho). As pontuações das células não constituem uma saída do arranjo sistólico, exceto as células selecionadas como máximo local, comunicadas ao final do processamento;

- `am_o`: sinaliza para o processador à direita apresentar seu máximo local. O sinal chega com dois ciclos de diferença entre cada processador (na descrição em Verilog no anexo A.3, vemos o uso de dois registradores entre os sinais `am_i` e `am_o`. Isto permite o sincronismo que faz com que o máximo de cada processador seja apresentado como saída do sistólico. Enquanto esse sinal é mantido inativo, cada processador somente "copia" os máximos dos vizinhos à esquerda nas saídas. Quando esse sinal é ativado, cada processador passa a apresentar na saída o seu máximo local calculado;

- `max_o`: esta saída contém significados diferentes em dois momentos:

  - quando o sinal `am_i` está desativado, o processador deve somente passar para direita o máximo local lido do processador da esquerda;

  - quando o sinal `am_i` está ativado, o processador deve aprentar seu máximo. Neste momento, `max_o` contém a pontuação do máximo local do processador. Isto faz com que todos máximos cheguem ao final do arranjo e daí sejam uma saída do sistólico.

- `max_clk_ticks_o`: antidiagonal da célula com pontuação apresentada em `max_o` (acima).

Lógica interna: Cada processador calcula as células de uma coluna da matriz de similaridade. Tem fixa a base de consulta e recebe como entrada as bases do banco-de-dados. A cada ciclo, recebe também o valor da célula vizinha à esquerda, calculado pelo processador adjacente à esquerda. Durante o cálculo, mantém informações sobre a célula com valor máximo calculado por si até então. Ao final das bases do banco-de-dados, sinalizado por `fim_bbd_i`, termina o cálculo. Está determinado então o máximo local (desta coluna). Quando solicitado (pelo sinal `am_i`), cada processador deve apresentar sua célula com pontuação máxima (saídas `max_o` e `max_clk_ticks_o`). Até que seja solicitado que apresente seu máximo, enquanto o sinal `max_i` estiver inativo, cada processador deve somente "copiar" o máximo lido na entrada (`max_i` e `max_clk_ticks_i`) para a saída (`max_o` e `max_clk_ticks_o`). Isto permite a passagem dos máximos locais entre os processadores e daí para consituírem uma saída do conjunto como um todo.

Estrutura interna:

Na figura 6.11 vemos uma representação da estrutura interna de cada processador. É formado por dois subcomponentes. O da esquerda na figura é o responsável pelo cálculo do valor das células, e é descrito com detalhes na seção 6.4.4. O subcomponente da direita mantém e apresenta o máximo local calculado por este processador. É descrito com detalhes na seção 6.4.5.

A descrição do processador, em linguagem de descrição de *hardware* Verilog, encontra-se no anexo A.3.

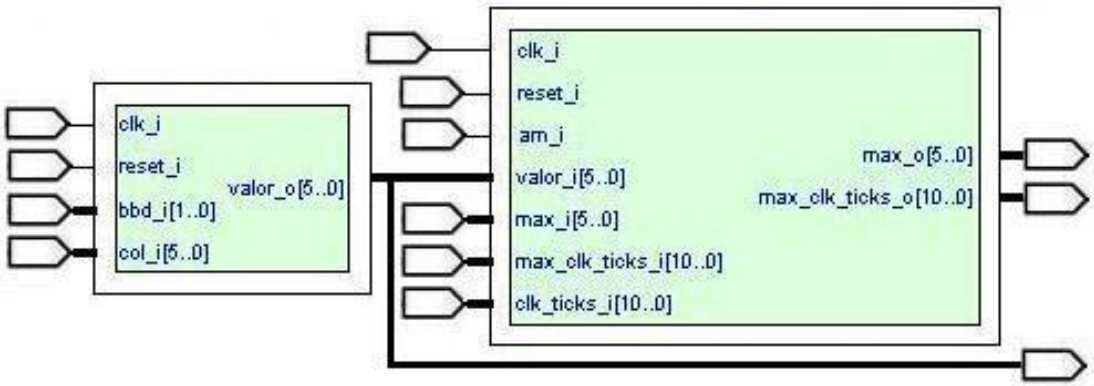

FIGURA 6.11: Componente de *hardware* - processador (visão interna)

6.4.4 Processador - subcomponente de cálculo das pontuações

Na figura 6.12 vemos o componente de cada processador que é responsável pelo cálculo da pontuação das células da matriz de similaridade. Essencialmente, a pontuação é computada utilizando-se a equação de recorrência do algoritmo de alinhamento local de Smith-Waterman. A implementação segue o modelo de processador descrito em 4.4. Vimos que, para o cálculo de cada célula S_{ij}, são considerados, além das bases envolvidas, o valor das células S_{ij-1} (coluna à esquerda), S_{i-1j-1} (diagonal superior esquerda) e S_{i-1j} (linha acima). Esses valores ficam armazenados em registradores (memórias) internos a cada processador. É um componente de mais baixo nível, composto somente de elementos primitivos como registradores, seletores, operadores aritméticos e de comparação.

Entradas:

- `clk_i` e `reset_i`: sinais de relógio e de reset;
- `bbd_i`: as bases do banco-de-dados, uma por ciclo de relógio, iniciando após o reset;
- `col_i`: valor da célula calculada pelo vizinho à esquerda. Será memorizado

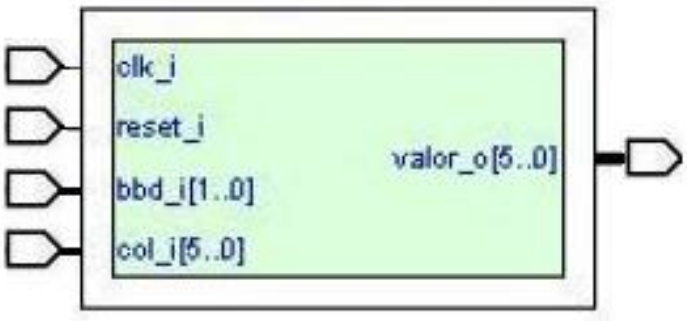

FIGURA 6.12: Componente de *hardware* - processador - cálculo das pontuações

para o cálculo dá célula neste processador (no próximo ciclo, será o valor de S_{ij-1} durante o cálculo de S_{ij}).

Saídas:

- `valor_o`: a saída deste componente é somente o valor calculado para cada célula, todas de uma mesma coluna da matriz. A cada ciclo, após o *reset*, uma célula da coluna é computada e informada por estes sinais de saída. O valor desta saída vai ser utilizado com duas finalidades:
  - para o processador vizinho a direita calcular as próximas células;
  - para alimentar a lógica de máximo local deste processador.

Lógica interna:

Para calcular cada célula S_{ij}, o processador utiliza as seguintes informações:

- base do banco-de-dados desta posição (lida de `bbd_i` e memorizada em um registrador);
- a célula na posição S_{ij-1}. Esta célula é o valor calculado pelo processador à esquerda, um ciclo atrás. Foi lida da entrada `col_i` e fica armazenada em um registrador, chamado nesta implementação de `col_r` (`col` de coluna e `_r` de registrador);

- a célula na posição S_{i-1j-1}. Esta célula é o valor calculado pelo processador à esquerda, dois ciclo atrás. A implementação é a seguinte: o valor do registrador `col_r` passa para o registrador `diag_r` (`diag` de diagonal), com um ciclo de diferença. Com a computação de uma antidiagonal por ciclo, o valor que no ciclo anterior era da coluna à esquerda, um ciclo depois passa a ser o da diagonal superior esquerda;

- a célula na posição S_{i-1j}. Esta célula é o valor calculado por este próprio processador, um ciclo atrás. Assim, a cada ciclo cada processador armazena o valor que computou em um registrador (`lin_r`, `lin` de linha pois na matriz esta célula está na linha acima de S_{ij}).

Assim, para a memorização das células vizinhas, são utilizados três registradores internos, vistos como os retângulos mais escuros na figura 6.12. Conforme a equação de recorrência, ao valor de S_{ij-1} e de S_{i-1j} é subtraído 2 (penalidade para *gap*). Ao valor de S_{i-1j-1}, é somado ou subtraído 1, se as bases casam (*match*) ou não (*mismatch*), respectivamente. Ainda conforme a equação de recorrência, é computado o máximo entre estes três valores (resultados da somas e subtrações). O máximo entre os três é o valor para esta célula S_{ij}. Se este valor for negativo, é substituído por zero, pois estamos lidando com alinhamento local de seqüências onde não existem números negativos na matriz.

Estrutura interna:

Na figura 6.13 temos uma representação da estrutura interna. Podemos ver este componente como sendo composto das seguintes partes:

- três registradores (`lin_r`, `col_r` e `diag_r`), que armazenam o valor das células vizinhas a esta sendo calculada;

- uma lógica para aplicar os bônus e penalidades (+1, -1 ou -2) sobre os registradores acima;

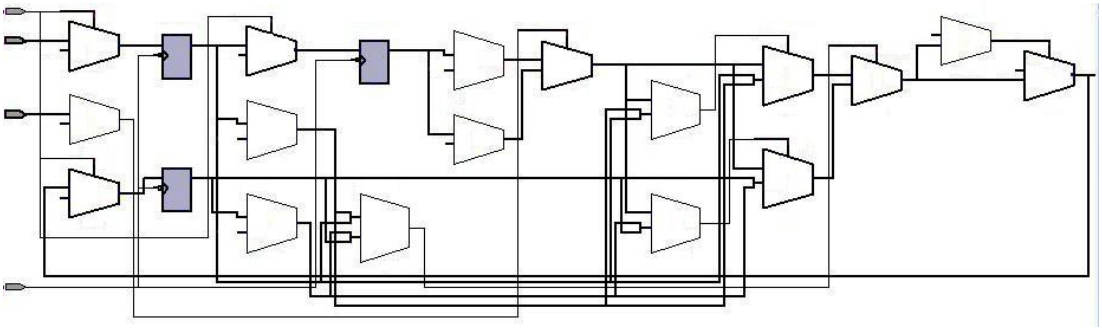

FIGURA 6.13: Componente de *hardware* - processador - cálculo das pontuações (visão interna)

- uma lógica para computar o valor máximo entre os resultados do item anterior;
- uma lógica para verificar se o máximo é um valor negativo. Se for, é substituído por zero.

Podemos dizer que a saída (`valor_o`) é composta pelo máximo entre zero e o valor das células vizinhas, depois de aplicadas as penalidades ou bônus. A descrição completa deste componente, em linguagem Verilog, encontra-se no anexo A.4.

6.4.5 Processador - subcomponente de cálculo do máximo local

Esse subcomponente do processador recebe o valor calculado por este processador para cada célula, sendo responsável por manter e apresentar a que obteve a maior pontuação (o máximo local do processador e conseqüentemente da coluna da matriz). Como o anterior, também é um componente de mais baixo nível, composto de elementos primitivos como registradores, seletores e comparadores.

Entradas:

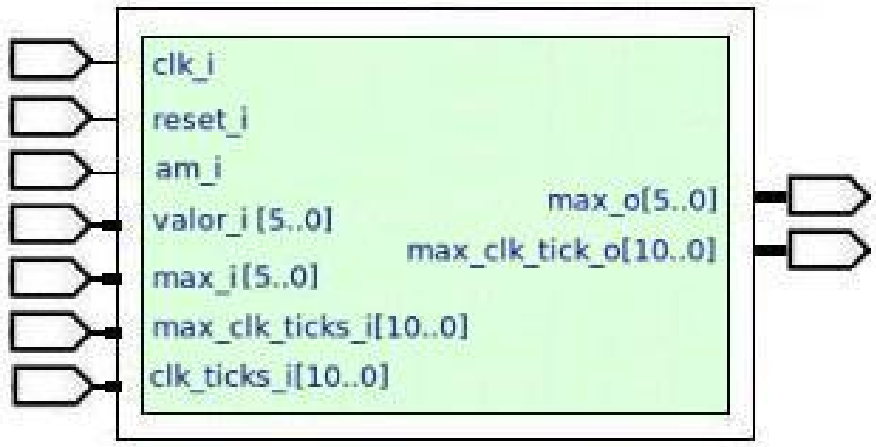

FIGURA 6.14: Componente de *hardware* - processador - cálculo do máximo local

- `clk_i` e `reset_i`: sinais de relógio e de reset;
- `valor_i`: valor da célula, calculado neste ciclo por este processador, especificamente pelo componente 6.4.4.
- `am_i`: sinal para o processador apresentar o resultado do processamento, a célula calculada que obteve o valor máximo (o máximo local). Esta saída solicitada é apresentada nos sinais `max_o` e `max_clk_ticks_o`.
- `max_i`: pontuação recebida do processador vizinho da esquerda, como sendo o máximo local daquele. Estes sinais de entrada permitirão a passagem dos máximos locais entre processadores vizinhos, de forma que todos ”fluam” até o último processador à direita e, daí, sejam uma saída do sistólico. O mecanismo de passagem dos máximos foi descrito em 6.2.2.
- `max_clk_ticks_i`: antidiagonal onde ocorreu o máximo local apresentado pelo vizinho, conforme item anterior;
- `clk_ticks_i`: relógio global, informa o número de ciclos decorrido após o *reset*. Usado para cada processador ter disponível o número da antidiagonal que está calculando neste momento.

Saídas:

- `max_o`: pontuação de um máximo local, que será:
  - o máximo deste próprio processador, se a entrada `am_i` estiver ativada. Neste caso, a entrada ativada indica ao processador que é o momento de apresentar seu próprio máximo local;
  - o máximo recebido do processador à esquerda, se a entrada `am_i` estiver desativada. Neste caso, o processador deve copiar, para a saída `max_o`, o máximo lido do processador vizinho (entrada `max_i`).

  Este mecanismo implementa a passagem dos máximos entre os processadores, até a saída do vetor, conforme descrito na seção 6.2.2.

- `max_clk_ticks_o`: antidiagonal onde ocorreu a pontuação de máximo local apresentada em `max_o` (item anterior). Conforme visto, a antidiagonal coincide com o ciclo de relógio no momento do cálculo da célula. A contagem dos ciclos é feita por um relógio global e informada para cada processador por meior da entrada `clk_ticks_i`.

Lógica interna:

O sinal de *reset* inicializa o valor do máximo armazenado com zero. Desligado o *reset*, cada célula calculada pelo processador é comparada com o máximo armazenado. Se o valor calculado para a célula corrente for maior que o máximo armazenado, é armazenado como novo máximo. O valor da célula corrente é lido da entrada `valor_i`. Ao final do cálculo da matriz, inicia a fase de apresentação dos máximos de cada processador. Os máximos recebidos do processador à esquerda (sinais `max_i` e `max_clk_ticks_i`) são simplesmente copiados para a saída, nos sinais `max_o` e `max_clk_ticks_o`. Isto até que o sinal `am_i` seja ativado. A ativação deste sinal indica que é o momento deste elemento de processamento apresentar seu próprio máximo nas saídas. Então, no ciclo seguinte,

`max_o` e `max_clk_ticks_o` irão conter a pontuação e antidiagonal da célula com valor máximo produzida por este processador.

Estrutura interna:

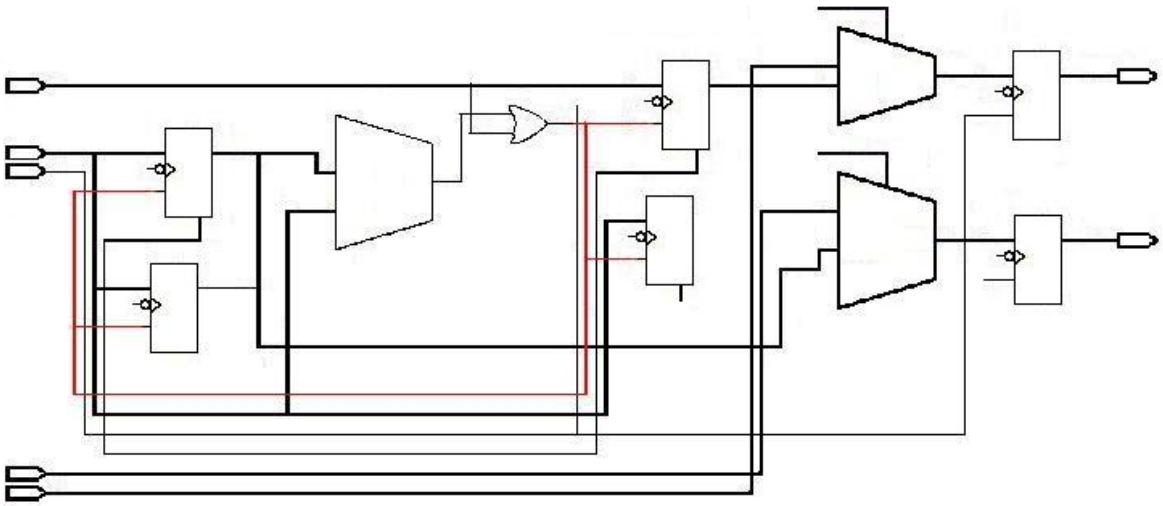

FIGURA 6.15: Componente de *hardware* - processador - cálculo do máximo local (visão interna)

A figura 6.15 dá uma visão da estrutura interna deste componente, que é composto, essencialmente, por memória para armazenar o máximo local, uma lógica de comparação e uma lógica para apresentação dos máximos. Quando o valor da célula corrente é maior que o máximo, temos um novo máximo que é gravado na memória. O sinal `am_i` serve como seletor da saída (`max_o` e `max_clk_ticks_o`), escolhendo entre o máximo deste próprio elemento ou o máximo lido do vizinho, implementando o mecanismo de passagem de máximos entre os processadores. A descrição completa deste componente em linguagem Verilog encontra-se no anexo A.5.

6.5 Descrição do *software*

Para o usuário final, a informação relevante é a pontuação e a visualização dos alinhamentos selecionados. Como vimos, as saídas do *hardware* são as células da matriz que apresentaram valor máximo local por coluna. Estas células representam os alinhamentos selecionados, incluindo o ótimo. A reconstrução e a visualização desses alinhamentos podem ser feitas por um *software*. A reconstrução é feita em tempo quadrático ($O(m \times n)$) em relação ao tamanho das subseqüências envolvidas (não das seqüências completas).

Neste projeto foi desenvolvido um *software* que recebe a saída do *hardware*, mais as seqüências (lidas de um arquivo), reconstrói os alinhamentos e proporciona sua visualização. Como vimos, as saídas do *hardware* são triplas (valor,coluna e antidiagonal), sobre máximos locais da matriz. Estas triplas identificam o alinhamento que será reconstruído. Segue uma visão geral do funcionamento do *software*:

- recebe como argumento uma tripla (pontuação, coluna e antidiagonal). Esta célula, indicada pelo *hardware*, representa o último pareamento de bases do alinhamento a ser reconstruido;
- a partir da coluna e da antidiagonal da célula indicada, deduz-se a linha onde esta ocorre. A linha da célula indica a base do banco-de-dados onde termina o alinhamento. A coluna da célula indica a base de consulta onde termina o alinhamento;
- lê as seqüências do arquivo, até o ponto de término do alinhamento selecionado;
- inverte a ordem dos caracteres de cada seqüência. O primeiro caracter passa a ser o último, o segundo o penúltimo e assim por diante. O primeiro

caracter das seqüências invertidas coincide com o caracter indicado pelo *hardware* como término do alinhamento;

- pelo algoritmo de Smith-Waterman, constrói a matriz de similaridade para as seqüências invertidas. Sobre quando parar a construção da matriz, duas abordagens possíveis são:
  - quando a pontuação esperada (indicada pelo hardware) é alcançada. Isto leva a descoberta de um alinhamento com aquela pontuação, terminado no ponto indicado pelo *hardware*. No entanto, podem existir outros com a mesma pontuação e que terminem na mesma posição. Estes alinhamentos não seriam identificados nesta abordagem;
  - quando todos os alinhamentos iniciados na primeira célula tiverem sido calculados, considerando-se primeira célula a do canto superior-esquerdo da matriz. A primeira célula da matriz das seqüências invertidas é a posição indicada pelo *hardware* como término do alinhamento selecionado. Assim, esta abordagem garante que todos os alinhamentos que terminem na posição indicada pelo *hardware* e que apresentem a pontuação esperada foram identificados. A desvantagem em relação a abordagem anterior é que esta pode levar um tempo maior de computação.

O alinhamento que possui a pontuação máxima desta região das seqüências invertidas coincide com o alinhamento indicado pelo *hardware*, exceto pela ordem dos caracteres nas seqüências que é invertida. O pareamento de bases observado nos dois casos é o mesmo. Assim, o alinhamento selecionado pelo hardware é identificado na matriz. Feito isto, a visualização, escrevendo-se uma seqüência acima da outra, é preparada. Um exemplo de saída do *software* construído é dada a seguir.

```
       2 3 ...                                       123
       G  A  T  T  T  A  G  C   A  G  G  T  A  T  A  G
  2 G  1
  3 A     2
  4 T        3
  5 G           2
  6 T              3
  7 A                 4
  ...                    ...
115 A                    ...   29
116 C                            28
117 T                               27
118 G                                  28
119 T                                     29
120 A                                        30
121 A                                           29
122 A                                              30
123 G                                                 31
```

```
  G  A  T  T  T ...  C  G  G  G  A  C  A  G  A  A  G  G  T  A  T  A  G
  G  A  T  G  T ...  C  C  A  G  A  C  C  G  A  C  T  G  T  A  A  A  G
-------------------------------------------------------------------------
  1  2  3  2  3 ... 27 26 25 26 27 28 27 28 29 28 27 28 29 30 29 30 31
```

A reconstrução reproduzida acima foi feita a partir dos exemplos de saída de *hardware* apresentados na seção 6.3.2: pontuação 31, coluna de término 123 e antidiagonal de término 246. Com estas informações, mais as seqüências de teste lidas de um arquivo, o *software* reconstruiu o alinhamento e sua visualização. O código-fonte do software desenvolvido, em linguagem C, está no anexo B.1.

6.6 Outras otimizações

Quanto ao número de *bits* usados para representar as pontuações na matriz, cabe a seguinte observação. Quanto menor o número de *bits*, menos área em

chip gastará cada processador. Consequentemente, mais processadores podem ser inclusos no arranjo sistólico, aumentando o grau de paralelismo no cálculo.

No entanto, se um número insuficiente de *bits* é usado, ocorre *overflow* do valor. Quando detectado, o *overflow* não é um problema muito grande. Basta configurar novos processadores, com um número maior de *bits* e recalcular a área onde ocorreu *overflow*. Esta técnica também é citada por Farrar (FARRAR, 2007). O principal problema do *overflow* é quando este não é detectado, levando a resultados equivocados.

Na proposta deste trabalho, a lógica de máximo local, existente em cada elemento de processamento, permite a detecção de *overflow*. Não é necessária lógica adicional. Para exemplificar, podemos supor uma representação de 8 *bits* para a pontuação. O maior número sem sinal possível de ser representado neste caso é 255 ($2^8 - 1$). Acontece *overflow* se tivermos este número (255) e a ele somarmos 1 (*match* entre bases).

Na proposta deste projeto, sempre poderíamos saber as regiões da matriz onde a pontuação chegou a este valor. Isto devido à lógica de máximo local existente em cada processador. Numa lógica de 8 *bits*, se for memorizado 255 como máximo local de um processador, qualquer número comparado com este vai ser sempre menor ou igual. Isto porque 255 é o maior número sem sinal possível de ser representado em 8 *bits*, garantindo sua permanência como máximo local. Quando este valor for apresentado como máximo deste processador, sabemos que esta é uma região onde ocorreu um *overflow*, se o próximo pareamento de bases é um *match*. Assim, seguramente podemos usar um número reduzido de *bits*, pois a lógica de máximo local garante que, se houverem regiões onde os valores se aproximaram do *overflow*, estas regiões serão identificadas. As regiões de *overflow* devem ser então reprocessadas, com processadores que usem um número maior de *bits* para representar as pontuações.

Sobre a representação dos números internamente aos processadores, é usada uma lógica baseada em 2. Podemos dizer que, em cada processador, o valor 2 corresponde a pontuação 0, 3 corresponde a 1 e assim por diante. De uma forma geral, a pontuação real é a pontuação representada no processador menos 2. Com uma lógica baseada em 2, fica impossível gerar números negativos, porque a maior penalidade aplicável é -2 ($2-2=0$). Assim, não lidamos com números negativos. Sem números negativos, os comparadores do processador não precisam verificar o *bit* de sinal (pois este não existe). Isto simplifica e previne erros na descrição do *hardware*, pois todos os valores são sem sinal, além de, possivelmente, gerar um circuito mais simples.

7 Resultados

7.1 Plataforma de prototipação

O *hardware* proposto neste texto foi prototipado, utilizando-se uma placa com FPGA da Altera. O *chip* FPGA presente na placa é um Stratix II S60 (EP2S60F672C3). Esta placa é do tipo *stand-alone*, ou seja, funciona desconectada do barramento de um microcomputador. Para a saída dos dados, foram usados os dois *displays* de sete segmentos disponíveis na placa. Para a entrada dos dados, foi usado somente um botão de apertar (*push-button*) da placa.

7.2 Número de elementos de processamento

A descrição de *hardware* feita é quanto ao número de processadores instanciados dentro do arranjo sistólico. Para uma determinada plataforma de implementação, podemos determinar por meio de estimativas e testes o número máximo de processadores possíveis de serem criados no *chip*. Nesta prototipação, para a plataforma descrita na seção 7.1, foi possível a criação de 636 elementos de processamento no FPGA. Foram usados elementos de processamento com 5 *bits* para a representação das pontuações. Como discutido em 6.6, se ocorrer *overflow* dos valores este será detectado e um reprocessamento da região deve ocorrer, com processadores que utilizem um número maior de *bits* para representar as

pontuações.

7.3 Freqüência máxima de operação

A ferramenta do fabricante do FPGA, ao final do processo de síntese, diz-nos a freqüência máxima de operação do circuito descrito. O valor informado é um limite seguro até o qual podemos elevar a freqüência, respeitando as características físicas do dispositivo (como os tempos característicos das memórias e o tempo de propagação dos sinais). Na prototipação realizada, a ferramenta Quartus da Altera informou que, utilizando os 636 elementos de processamento citados na seção anterior, a freqüência máxima de operação seria pouco mais de 78 MHz (figura 7.1).

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From
1	Worst-case tsu	N/A	None	9.340 ns	reset_i
2	Worst-case tco	N/A	None	15.349 ns	top:inst|sistolico_input:sist
3	Worst-case th	N/A	None	-2.613 ns	reset_i
4	Clock Setup: 'clk_i'	N/A	None	78.18 MHz (period = 12.791 ns)	top:inst|sistolico:sistolico1|
5	Total number of failed paths				

FIGURA 7.1: Freqüência máxima de operação da prototipação

7.4 Desempenho global

Para sistemas de alinhamento de seqüências que utilizam programação dinâmica, o desempenho pode ser medido pelo número de células da matriz atualizadas por unidade de tempo. Em inglês, a sigla usada para tal medida é *CUPS* (*cell updates per second*, em português, atualizações de células por segundo). No *hardware* proposto, cada processador calcula uma célula da matriz por ciclo de relógio. Como podemos ter 636 processadores operando a até 78 milhões de ciclos

de relógio por segundo (78 MHz), o cálculo do desempenho global pode ser feito da seguinte forma:

$$636 \times 78 \times 10^6 = 49,6 \text{ GCUPS}$$

Ou seja, na prototipação realizada, aproximadamente 50 bilhões de células da matriz de similaridade puderam ser calculadas em 1 segundo.

7.5 Desempenho global comparado

A tabela 7.1 mostra o desempenho obtido na prototipação, comparado ao de outras soluções publicadas. A tabela está ordenada por ordem crescente de desempenho. A coluna "plataforma" pode conter os valores: software (solução somente em *software*), FPGA (arquitetura reconfigurável), ASIC (chip específico para aplicação), GPU (processadores gráficos) ou SIMD (instruções vetoriais em processadores de uso geral) .

TABELA 7.1: Desempenho comparado - ordem crescente de desempenho

Descrição	**Plataforma**	**Desemp. absoluto**	**Desemp. relativo**
Intel Xeon 2.6 GHz(HARPER, 2007)	Software	53 MCUPS	0,001
Pentium III 500 MHz (ROGNES; SEEBERG, 2000)	SIMD	150 MCUPS	0,003
The UCSC Kestrel Parallel Processor (BLAS *et al.*, 2005). Resultados de 1997	ASIC	400 MCUPS	0,008
8 Pentium III 600 MHz (ROGNES; SEEBERG, 2000)	SIMD	1,5 GCUPS	0,03
Hyperseq. Vários módulos de *hardware*. Considerando alinhamentos com espaços (HARPER, 2007)	FPGA	2,8 GCUPS	0,056
Striped SmithWaterman speeds database searches six times ... (FARRAR, 2007). Instruções Intel SSE2	SIMD	3 GCUPS	0,06
PheGee (SINGH *et al.*, 2007). Grid de *desktops* com 10 GPUS NVIDIA GeForce	GPU	4,5 GCUPS	0,09
SWBoost: Smith-Waterman Boost (GENBOOST, 2007)	GPU	5-10 GCUPS	0,1
Cray XD1 (STRENSKI, 2005)	FPGA	5-15 GCUPS	0,1
Hyper Customized Processors for ... (OLIVER; SCHMIDT; MASKELL, 2005). Penalidade *affine* de espaços	FPGA	7,6 GCUPS	0,152
Cell - CLC Bionformatics (BIOINFORMATICS, 2007) . Intel SSE2	SIMD	9,3 GCUPS	0,186
Cadeias ocultas de Markov (JACOB *et al.*, 2007)	FPGA	10 GCUPS	0,2
Families of FPGA-based accelerators for ... (COURT; HERBORDT, 2006). 2 Virtex II Pro	FPGA	13 GCUPS	0,26
Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs (OLIVER; SCHMIDT; MASKELL, 2005). Penalidade linear de espaços	FPGA	13,9 GCUPS	0,278
Nios mais instrução customizada (1.000.000 células). Desempenho estimado (LI; SHUM; TRUONG, 2007)	FPGA	23,8 GCUPS	0,476
Cadeias ocultas de Markov (MADDIMSETTY, 2006)	FPGA	34,4 GCUPS	0,688
Splash-2 (WOO *et al.*, 1995). 16 placas cada com 17 Xilinx XC4010	FPGA	43 GCUPS	0,86
Prototipação da proposta deste trabalho	**FPGA**	**50 GCUPS**	**1**
A Smith-Waterman Systolic Cell (YU *et al.*, 2003). Virtex XCV1000E via SDRAM (módulo Pilchard)	FPGA	136 GCUPS	2,72
Hyperseq. Vários módulos de *hardware*. Não considera alinhamentos com espaços (HARPER, 2007)	FPGA	139 GCUPS	2,78
Gene Matching Using JBits (GUCCIONE; KELLER, 2007). 4000 processadores de 2 *bits*. Virtex XCV1000	FPGA	750 GCUPS	15
Hookiegene (PUTTEGOWDA *et al.*, 2003). Placa OSIRIS com 2 Virtex II. 7000 processadores de 2 *bits*	FPGA	1260 GCUPS	25,2
Bioinformatics case study from the CAPSL Laboratory (ZHANG; TAN; GAO, 2007). Módulo XtremeData XD1000: Stratix II via AMD HyperTransport	FPGA	2048 GCUPS	40,96
Gene Matching Using JBits (GUCCIONE; KELLER, 2007). Estimados 11000 processadores (2 *bits*) se usada Virtex II XC2V6000. Desempenho estimado	FPGA	3200 GCUPS	64

8 Conclusão

O volume de informações em bancos-de-dados biológicos tem crescido mais do que os recursos computacionais disponíveis, desafiando a bionformática a apresentar técnicas cada vez mais eficientes de processamento destes dados. Uma das operações mais importantes na bionformática é o alinhamento de seqüências. O algoritmo de alinhamento local de Smith-Waterman é o que apresenta os melhores resultados, mas sua complexidade de tempo quadrática tem impedido o uso em larga escala. Algoritmos heurísticos, apesar de não serem exatos, têm freqüentemente sido usados nesses casos.

Neste trabalho, foi apresentada uma abordagem baseada em *hardware* reconfigurável dedicado para reduzir a complexidade de tempo do algoritmo de Smith-Waterman. O paralelismo potencial é explorado por um arranjo sistólico de elementos de processamento em um único *chip*. Um gargalo de comunicação previsto é evitado, selecionando-se ainda em *hardware* os alinhamentos que mais interessam. O alinhamento ótimo e outros "bons" alinhamentos - os máximos por coluna da matriz - são comunicados ao meio externo e podem ser reconstruídos por *software*.

A prototipação foi realizada em FPGA, usando um Altera Stratix II S60. Foi possível implementar 636 elementos de processamento, a uma freqüencia máxima de operação de 78 MHz, resultando em um desempenho de aproximadamente 50 GCUPS.

Numa continuação deste projeto, poderia ser feita a integração do *hardware* dedicado descrito com um microcomputador. O padrão de comunicação de dados mais comumente usado nestes casos é o PCI, apesar de existirem módulos com FPGA e conectores de outros padrões, inclusive com desempenho superior (YU *et al.*, 2003) (ZHANG; TAN; GAO, 2007). Poderia ser criado um *cluster* ou um *grid* de microcomputadores com *hardware* dedicado acoplado.

Outros projetos poderiam abranger o problema da divisão de um alinhamento de seqüências, para execução do algoritmo de alinhamento por partes. Devido a insuficiência de recursos no *chip*, nem sempre é possível que toda a seqüência de consulta seja comparada de uma só vez. Neste caso, as seqüências devem ser particionadas, os problemas parciais solucionados e as soluções parciais combinadas, obtendo-se a solução final. Visto a limitação do espaço em *chip*, o desenvolvimento deste mecanismo de particionamento do problema é importante para o uso prático de soluções como a apresentada neste trabalho.

O objetivo de ajudar a viabilizar o uso de algoritmos exatos para alinhamento de seqüências foi alcançado, visto o ganho de desempenho observado. A aceleração em relação a uma solução somente em *software* foi cerca de 1.000. O uso de arquiteturas reconfiguráveis confirmou ser de grande utilidade para a aceleração do alinhamento de seqüências e de outros algoritmos de programação dinâmica.

Referências Bibliográficas

AMD. **AMD (Advanced Micro Technologies), divisão graphics and vídeo processors (antiga ATI)**. 2007. Disponível em: <http://www.ati.amd.com/>. Acesso em: 2007.

BIOINFORMATICS, C. **Cell**. 2007. Disponível em: <http://www.clccell.com>. Acesso em: 2007.

BIOLOGIA Molecular: O que é biologia molecular. 2006. Disponível em: <http://www.biomol.org/>. Acesso em: 2006.

BLAS, A. D.; DAHLE, D. M.; DIEKHANS, M.; GRATE, L.; HIRSCHBERG, J. The ucsc kestrel parallel processor. **IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS**, v. 16, n. 1, 2005.

BLAST: Ncbi (national center for biotechnology information) - blast (basic alignment sarch tool). 2006. Disponível em: <http://www.ncbi.nlm.nih.gov/BLAST/>. Acesso em: 2006.

BYUN, J.-H. **A Scalable Multi-FPGA Network System for Gene Sequencing**. Disserta (Mestrado) — University of North Carolina at Charlote, USA, 2005.

CARVALHO, L. G. A. **Uma Abordagem em Hardware para Algoritmos de Comparação de Seqüências Baseados em Programação ao Dinâmica**. Disserta (Mestrado) — Universidade de Brasília, 2003.

COMISSION, E. Prospective analysis of the relationship and synergy between medical informatics (mi) and bioinformatics (bi). **White paper EC-IST**, v. 35024, 2001.

CORPORATION, N. **NVidia**. 2007. Disponível em: <http://www.nvidia.com/>. Acesso em: 2007.

COURT, T. V.; HERBORDT, M. C. Families of fpga-based accelerators for approximate string matching. 2006. Disponível em: <http://www.bu.edu/caadlab/05MatchExt.pdf>.

(EBI), E. B. I. **UniProtKB/TrEMBL PROTEIN DATABASE**. 2007. Disponível em: <http://www.ebi.ac.uk/swissprot/sptrrline stats/index.html>. Acesso em: 2007.

FARRAR, M. Striped smithwaterman speeds database searches six times over other simd implementations. **Bioinformatics**, v. 23, n. 2, p. 156–161, 2007. Disponível em: <http://farrar.michael.googlepages.com/Smith-waterman>.

FASTA: Fasta sequence comparison at the university of virginia. 2006. Disponível em: <http://fasta.bioch.virginia.edu/>. Acesso em: 2006.

GENBANK. **GenBank Statistics**: Estatísticas do genbank. 2006. Disponível em: <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>. Acesso em: 2006.

GENBOOST. **SWBoost (Smith-Waterman Boost)**. 2007. Disponível em: <http://www.genboost.com/>. Acesso em: 2007.

GOTOH. An improved algorithm for matching biological sequences. **J. Mol. Biol.**, v. 162, p. 705–708, 1982.

GUCCIONE, S. A.; KELLER, E. Gene matching using jbits. **Xilinx Inc.**, 2007.

HARPER, S. **Accelerating Smith-Waterman Alignment with the HyperSeqTM System**. [S.l.], 2007. Disponível em: <http://www-.adaptivegenomics.com/pdf/Accelerating_Smith-Waterman_Processing.pdf>. Acesso em: 2007.

HOANG, D. T. **A Systolic Array for the Sequence Alignment Problem**. [S.l.], 1992. Disponível em: <citeseer.ist.psu.edu/hoang92systolic.html>.

JACOB, A.; LANCASTER, J.; BUHLER, J.; CHAMBERLAIN, R. D. Preliminary results in accelerating profile hmm search on fpgas. In: **Proceedings of 6th IEEE International Workshop on High Performance Computational Biology, ACM symposium on Applied computing**. [S.l.: s.n.], 2007.

KUNG, H.; LEISERSON. Systolic arrays for vlsi sparse matrix. In: **Proceedings of the Society for Industrial and Applied Mathematics**. Santa Monica, CA, USA: [s.n.], 1979. p. 256–282.

LAVENIER, D. **SAMBA : Systolic Accelerator for Molecular Biological Applications**. [S.l.], 1998. 22 p. p. Disponível em: <citeseer.ist.psu.edu/lavenier96samba.html>.

LEONG, P.; LEONG, M.; CHEUNG, O.; TUNG, T.; KWOK, C.; WONG, M.; LEE., K. H. Pilchard - a recongurable computing platform with memory slot interface. In: **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**. [S.l.: s.n.], 2001.

LI, I. T.; SHUM, W.; TRUONG, K. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). **BMC Bioinformatics**, v. 8, p. 185, 2007.

LIPTON, R.; LOPRESTI, D. A systolic array for rapid string comparison. In: **Chapel Hill Conference on VLSI, Chapel Hill, NC, USA**. [S.l.: s.n.], 1985. p. 363–376.

MADDIMSETTY, R. P. **ACCELERATION OF PROFILE-HMM SEARCH FOR PROTEIN SEQUENCES IN RECONFIGURABLE HARDWARE**. Disserta (Mestrado) — Henry Edwin Sever Graduate School of Washington University, Saint Louis, Missouri, USA, 2006.

MENG, X.; CHAUDHARY, V. An adaptive data prefetching scheme for biosequence database search on reconfigurable platforms. In: **Proceedings of the 2007 ACM symposium on Applied computing**. [S.l.: s.n.], 2007. p. 140–141.

MYERS; W., E.; MILLER, W. Optimal alignments in linear space. **Computer Applications in the Biosciences**, v. 4, p. 11–17, 1988.

NEEDLEMAN S. B. E WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. p. 443–453, 1970.

OLIVER, T.; SCHMIDT, B.; MASKELL, D. Hyper customized processors for bio-sequence database scanning on fpgas. In: **Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, FPGA, Monterey, California, USA**. [S.l.: s.n.], 2005.

PUTTEGOWDA, K.; WOREK, W.; PAPPAS, N.; DANDAPANI, A.; ATHANAS, P. A run-time recongurable system for gene-sequence searching. In: **Proceedings of the International VLSI Design Conference**. [S.l.: s.n.], 2003.

ROGNES, T.; SEEBERG, E. Six-fold speed-up of smith-waterman sequence databases searches using parallel processing on common microprocessors. **Bioinformatics**, v. 16, n. 8, p. 699–706, 2000. Disponível em: <http://bioinformatics.oxfordjournals.org/cgi/content/abstract/16/8/699>.

SETUBAL, J.; MEIDANIS, J. Introduction to computational molecular biology. **PWS Pub**, 1991.

SETUBAL, J. C.; MEIDANIS. **J. Computational Molecular Biology**, 1997.

SINGH, A.; CHEN, C.; LIU, W.; MITCHELL, W.; SCHMIDT, B. Phegee: Phenotype genotype exploration on a desktop gpu grid. 2007.

SINGH, R.; DETTLO, W.; CHI, V.; HO, D.; TELL m; WHITE, C.; ALTSCHUL, S.; ERICKSON, B. **BioSCAN: A dynamically reconfigurable systolic array for biosequence analysis**. Disponível em: <citeseer.ist.psu.edu/163511.html>.

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

SOLUTIONS, T. B. **Decypher**. 2007. Disponível em: <http://timelogic.com/decypher_intro.html>. Acesso em: 2007.

STRENSKI, D. The cray xd1 computer and its reconfigurable architecture. 2005. Disponível em: <http://www.ncsa.uiuc.edu/Conferences/RSSI/2005/docs/Strenski.ppt>.

TRELLES, O. On the parallelization of bioinformatic applications. 2001.

VOSS, G.; SCHRöDER, A.; MüLLER-WITTIG, W.; SCHMIDT, B. Biological sequence alignment on graphics processing units. 2005. Disponível em: <http://www.ntu.edu.sg/home/asbschmidt/paper/BioGPU.pdf>.

WOO, S. C.; OHARA, M.; TORRIE, E.; SINGH, J. P.; GUPTA., A. The splash-2 (stanford parallel applications for shared memory) programs: Characterization and methodological considerations. In: **In Proceedings of the 22nd International Symposium on Computer Architecture**. Santa Monica, CA, USA: [s.n.], 1995. p. 24–36. Disponível em: <ftp://www-flash.stanford.edu/pub/splash2/splash2$_i$*sca*95.*ps.Z*>.

XTREMEDATA. Xd1000 fpga coprocessor module for socket 940. 2007. Disponível em: <http://www.capsl.udel.edu/pub/doc/memos/memo078.pdf>.

YAMAGUCHI, Y.; MARUYAMA, T.; KONAGAYA, A. **High Speed Homology Search with FPGAs**. Disponível em: <citeseer.ist.psu.edu/yamaguchi02high.html>.

YANG, B. H. W. **A parallel implementation of Smith-Waterman sequence comparison algorithm**. [S.l.], 2002.

YU, C.; KWONG, K.; LEE, K.; LEONG, P. A smith-waterman systolic cell. In: **13th International Conference on Field-Programmable Logic and Applications. Springer-Verlag LNCS**. Lisboa, Portugal: [s.n.], 2003. p. 2778:375–384.

ZHANG, P.; TAN, G.; GAO, G. R. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. 2007. Disponível em: <http://www.xtremedatainc.com/xd1000e brief.html>.

Anexo A - Descrição do hardware em Verilog

A.1 Componente de mais alto nível da aplicação

```
`include "parametros.v"

module top
    (
        input clk_i, input reset_i,
        output max_pronto_o, input max_proximo_i,
        output [`SCORE_MAX_BITS-1:0] max_o,
        output [`SEQ_C_TAM_BITS_DEF-1:0] max_col_o,
        output [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_o,

        output fim_bbd_o,
        //informa progresso da entrada para exibicao
        output [7:0] bbd_falta_msb_o
    );

    //Sequencia de consulta - definida em parametros.v
    localparam SEQ_C = `SEQ_C_DEF;
    localparam SEQ_C_TAM = `SEQ_C_TAM_DEF;
    localparam SEQ_C_TAM_BITS = `SEQ_C_TAM_BITS_DEF;

    //Sequencia do banco de dados
```

```
localparam SEQ_BD_TAM = `SEQ_BD_TAM_DEF;
localparam SEQ_BD_TAM_BITS = `SEQ_BD_TAM_BITS_DEF;

//memoria ROM onde estara a sequencia do banco de dados
//cada byte (8 bits) pode conter ateh 4 bases (2 bits)
localparam ROM_SIZE    = `ROM_SIZE_DEF;
localparam ROM_SIZE_BITS = `ROM_SIZE_BITS_DEF;

/*
 * Acrescenta um ciclo de atraso, permitindo que sistolico_input
 * esteja um ciclo adiante do sistolico, gerando entradas
 * corretamente
 */
reg reset_r;
always @(`CLOCK_EDGE_PRINCIPAL clk_i) reset_r <= reset_i;

//sinais sistolico_input -> sistolico
wire [1:0] bbd_w; wire fim_bbd_w;
//entrada do sistolico, bases do banco de dados
sistolico_input
      #(
         .SEQ_BD_TAM(SEQ_BD_TAM), .SEQ_BD_TAM_BITS(SEQ_BD_TAM_BITS),
         .ROM_SIZE_BITS(ROM_SIZE_BITS)
      )
      sistolico_input1
      (
         .clk_i(clk_i), .reset_i(reset_i),
         .bbd_o(bbd_w),  //bases para o sistolico
         .fim_bbd_o(fim_bbd_w),    //sinaliza final das bases
      .bbd_falta_msb_o( bbd_falta_msb_o )  //p exibicao do progresso
      );

assign fim_bbd_o = fim_bbd_w;

//sinais sistolico -> sistolico_output
wire max_inicio_w;
wire [`SCORE_MAX_BITS-1:0] max_w;
wire [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_w;
//arranjo sistolico de elementos de processamento
sistolico
   #( .SEQ_C_TAM(SEQ_C_TAM), .SEQ_C(SEQ_C) )
   sistolico1
   (
```

```
        .clk_i(clk_i), .reset_i(reset_r),
        //entradas: bases do banco de dados e sinal fim das bases
        .bbd_i(bbd_w), .fim_bbd_i(fim_bbd_w),
        //saida: maximos de cada coluna e tempo de producao destes
        .max_inicio_o(max_inicio_w),
        .max_o(max_w), .max_clk_ticks_o(max_clk_ticks_w)
    );

//fios sistolico_output -> porta de escrita da RAM
wire ram_wen_w; wire [SEQ_C_TAM_BITS-1:0] ram_waddr_w;
wire [`SCORE_MAX_BITS+`CLK_TICKS_MAX_BITS-1:0] ram_data_in_w;
//fios sistolico_output -> apresent_max
wire max_global_pronto_w; wire [`SCORE_MAX_BITS-1:0] max_global_w;

sistolico_output
    #(
        //le SEQ_C_TAM maximos locais de max_i
        .SEQ_C_TAM(SEQ_C_TAM),
        .SEQ_C_TAM_BITS(SEQ_C_TAM_BITS)
    )
    sistolico_output1
    (
        .clk_i(clk_i), .reset_i(reset_i),
        //sinais vindos do sistolico
        .max_inicio_i(max_inicio_w),
        .max_i(max_w), .max_clk_ticks_i(max_clk_ticks_w),
        //sinais indo para a RAM
        .ram_wen_o(ram_wen_w), .ram_waddr_o(ram_waddr_w),
        .ram_data_o(ram_data_in_w),
        //sinais indo para apresent_max
        .max_global_pronto_o(max_global_pronto_w),
       //maximo entre os maximos de todas as colunas
        .max_global_o(max_global_w)
    );

//fios apresent_max -> porta leitura RAM
wire [SEQ_C_TAM_BITS-1:0] ram_raddr_w;
wire [`SCORE_MAX_BITS+`CLK_TICKS_MAX_BITS-1:0] ram_data_out_w;

 apresent_max
    #(
        .SEQ_C_TAM_BITS(SEQ_C_TAM_BITS), //largura saida max_col_o
        .RAM_SIZE(SEQ_C_TAM),    //no. posicoes de memoria a considerar
```

```
        .RAM_SIZE_BITS(SEQ_C_TAM_BITS)
    )
    apresent_max1
    (
         .clk_i(clk_i), .reset_i(reset_i),
        //de sistolico_output
        .max_global_pronto_i(max_global_pronto_w),
        .max_global_i(max_global_w),
        //para RAM
        .ram_addr_o(ram_raddr_w), .ram_data_i(ram_data_out_w),
        //pinos meio externo
        .max_proximo_i(max_proximo_i),
        .max_pronto_o(max_pronto_o),
        .max_o(max_o), .max_col_o(max_col_o),
        .max_clk_ticks_o(max_clk_ticks_o)
    );


ram ram1
    (
         .clock(clk_i),
        //escrita
        .wren(ram_wen_w), .wraddress(ram_waddr_w),
        .data(ram_data_in_w),
        //leitura
        .rdaddress(ram_raddr_w),
        .q(ram_data_out_w)
    );
```

endmodule

A.2 Arranjo sistólico de elementos de processamento

```
module sistolico
    #(
        //o parametro abaixo eh a sequencia de consulta. Serah
            //usada para instanciar do elementos de processamento
        parameter     [0:SEQ_C_TAM*2-1] SEQ_C = 1'b0,
            //tamanho da sequencia de consulta
        parameter    SEQ_C_TAM = 0
    )
    (
        input clk_i, reset_i,
        //bases do banco do dados e sinal fim das bases
        input [1:0] bbd_i, input  fim_bbd_i,
        //sinal inicio dos maximos e maximos locais com tempo de producao
        output reg max_inicio_o,
        output [`SCORE_MAX_BITS-1:0] max_o,
        output [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_o
    );

    /*
     * Cria contador para manter o numero desta antidiagonal
     * Sinal servira de entrada para o sistolico e para os elementos
     * de processamento, para estes armazenarem em qual tempo
     * armazenaram seus valores maximos locais
    */
    wire clk_ticks_reset_w;
    //relogio conta numero desta antidiagonal (ciclos apos reset)
    reg [`CLK_TICKS_MAX_BITS-1:0] clk_ticks_r;
    always @(`CLOCK_EDGE_PRINCIPAL clk_i) clk_ticks_r <=
                clk_ticks_reset_w ? 1'b0 : clk_ticks_r+1'b1;

    reg max_inicio_r;
    always @(`CLOCK_EDGE_PRINCIPAL clk_i) begin
        max_inicio_o <= max_inicio_r;  max_inicio_r <= max_inicio_w;
    end


    /*
     * Instancia vetor de processamento sistolico,
     * de forma parametrizada, via generate-for
```

```
 */
assign clk_ticks_reset_w = reset_w[1];

/*
 * Sao criados "vetores" de wires para conectar os elementos
 * de processamento que serao  instanciados via "generate for".
 * Seguem o diagrama abaixo:
 *
 * bbd_w[0]  -> +--+ -> bbd_w[1]  ...-> +--+ ->bbd_w[SEQ_C_TAM]
 * valor_w[0]-> |00| -> valor_w[1]...-> |YY| ->valor_w[SEQ_C_TAM]
 * reset_w[0]-> +--+ -> reset_w[1]...-> +--+ ->reset_w[SEQ_C_TAM]
 *...
 * Precisamos de SEQ_C_TAM+1 conjuntos de cada tipo de fios
 */
wire reset_w [0:SEQ_C_TAM];
//base do banco de dados e sinal de fim das bases
wire [1:0] bbd_w [0:SEQ_C_TAM]; wire fim_bbd_w [0:SEQ_C_TAM];
//valores produzidos para cada celula da matriz
wire [`SCORE_MAX_BITS-1:0] valor_w [0:SEQ_C_TAM];
//sinal "apresentar seu maximo" e maximos de cada coluna
wire am_w [0:SEQ_C_TAM];
//maximos das colunas
wire [`SCORE_MAX_BITS-1:0] max_w [0:SEQ_C_TAM];
//tempos de producao daqueles maximos
wire [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_w[0:SEQ_C_TAM];

genvar index;
generate for (index = 0; index < SEQ_C_TAM; index = index + 1)
begin : generate_eps
   eps #( SEQ_C[index*2+:2] ) eps_instancia (
      //clock
      .clk_i( clk_i ),
      //reset
      .reset_i( reset_w[index] ), .reset_o( reset_w[index+1] ),
      //base do banco de dados
      .bbd_i( bbd_w[index] ), .bbd_o( bbd_w[index+1] ),
      .fim_bbd_i(fim_bbd_w[index]), .fim_bbd_o(fim_bbd_w[index+1]),
      //valores calculados (nao irao sair do sistolico)
      .col_i( valor_w[index] ), .valor_o( valor_w[index+1] ),
      //relogio
      .clk_ticks_i( clk_ticks_r ),

      //sinal controla apresentacao dos maximos
```

```
        .am_i( am_w[index] ), .am_o( am_w[index+1] ),
        //valores maximos locais das colunas
        .max_i( max_w[index] ),
        .max_clk_ticks_i( max_clk_ticks_w[index] ),
        .max_o( max_w[index+1] ),
        .max_clk_ticks_o( max_clk_ticks_w[index+1] )
    );
end
endgenerate

//fios conectados ao primeiro elemento
assign reset_w[0] = reset_i; assign bbd_w[0] = bbd_i;
//borda da matriz com valor 0 (representacao em excesso de 2)
 assign valor_w[0] = 2'b10;
//sinal fim da entrada
assign fim_bbd_w[0] = fim_bbd_i; assign am_w[0] = fim_bbd_i;

//fios conectados ao ultimo elemento
 wire max_inicio_w = fim_bbd_w[SEQ_C_TAM];
//desfaz notacao excesso de 2
assign max_o = max_w[SEQ_C_TAM]-2'b10;
assign max_clk_ticks_o = max_clk_ticks_w[SEQ_C_TAM];

endmodule
```

A.3 Elemento de processamento

```
module eps
  #(
    parameter    BASE_CONSULTA    = 2'b00
  )
  (
    input clk_i, input reset_i, output reg reset_o,

    //bases do banco de dados
    input [1:0] bbd_i, output reg [1:0] bbd_o,
    input fim_bbd_i, output reg fim_bbd_o,

    //valores da celula aa esquerda entrando e...
    input  [`SCORE_MAX_BITS-1:0] col_i,
    //... valor desta celula saindo
    output [`SCORE_MAX_BITS-1:0] valor_o,

    //relogio global que conta tempo decorrido
    input [`CLK_TICKS_MAX_BITS-1:0] clk_ticks_i,
    //sinal constrola apresenta maximos
    input am_i, output reg am_o,

    //informacoes de maximos locais, entrando e saindo:
    //pontuacao e ...
    input  [`SCORE_MAX_BITS-1:0] max_i,
    output [`SCORE_MAX_BITS-1:0] max_o,
    //...antidiagonal (coincide com tempo de producao)
    input  [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_i,
    output [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_o

  );

  //instancia sub-modulo que calcula o valor de cada celula (valor_o)
  eps_calc_valor #(BASE_CONSULTA) eps_calc_valor1 (
    .clk_i( clk_i ), .reset_i( reset_i ), .bbd_i( bbd_o ),
    .col_i( col_i ), .valor_o( valor_o )
  );

  //instancia sub-modulo que armazena maximo e controle sua exibicao
  //o reset_i eh ligado ao reset_o (registrados resetados
  //um ciclo depois)
```

```
  eps_calc_max eps_calc_max1 (
    .clk_i( clk_i ), .reset_i( reset_o ), .valor_i( valor_o ),
    .am_i( am_o ),
    .clk_ticks_i( clk_ticks_i ), //sinal do relogio global
    .max_i( max_i ), .max_o( max_o ),
    .max_clk_ticks_i( max_clk_ticks_i ),
    .max_clk_ticks_o( max_clk_ticks_o )
  );

  always @(`CLOCK_EDGE_PRINCIPAL clk_i ) reset_o <= reset_i;

  /* O registrador bbd_o eh carregado com o inverso binario da base
  * de consulta, provocando um mismatch, o que faz o score decrescer.
  * Proximo ao reset, isso impede a carga  de valores maoires que
  * zero nos registradores. Proximo ao final das bases, faz com que
  * o registrador de maximo nao memorize valor gerado apos termino
  * da entrada (manter maximo correto)
  */
  always @(`CLOCK_EDGE_PRINCIPAL clk_i or posedge reset_i )
    bbd_o   <= (reset_i||fim_bbd_o) ? ~BASE_CONSULTA : bbd_i;

  always @(`CLOCK_EDGE_PRINCIPAL clk_i )
    fim_bbd_o <= fim_bbd_i;

  /*
   * sinal am_i (apresentar maximo) controla as saidas max_o e
   * max_clock_ticks_o:
   *    - qdo ativo, apresentar nestas saidas o maximo calculado
   *      por este EPS (coluna)
   *    - qdo inativo, repassar maximo lido em max_i/max_clock_ticks_i
   *
   * Cada EPS contera dois registradores, 2 ciclos p am_i
   * atravessar cada EPS. Assim, o maximo de cada EPS saira
   * pelo elemento mais aa direita, um apos o outro.
   */
  reg am_r; //ciclo adicional de atraso entre am_i e am_o
  always @(`CLOCK_EDGE_PRINCIPAL clk_i )
    begin am_r <= am_i; am_o <= am_r; end

endmodule
```

A.4 Elemento de processamento - subcomponente de cálculo das pontuações

```
module eps_calc_valor
   #(
      /* Base da sequencia de consulta que esta
       * associada a este processador:
            * C (00), G (01), A(10) ou T (11)
        */
      parameter BASE_CONSULTA    = 2'b00
   )
   (
         input  clk_i, reset_i,
      input [1:0] bbd_i,
      //valores da celula aa esquerda entrando
      input  [`SCORE_MAX_BITS-1:0] col_i,
      //valor calculado para esta celula saindo
      output [`SCORE_MAX_BITS-1:0] valor_o
   );

   //penalizacoes escolhidas
   localparam PONT_MATCH = 1'b1, PONT_MISMATCH = 1'b1;
   localparam PONT_GAP = 2'b10;

   /*
    * Registradores para valores das celulas vizinhas
    * Usados a seguir no calculo combinacional desta celula
    */
   reg [`SCORE_MAX_BITS-1:0] lin_r, col_r, diag_r;
   always @(negedge clk_i or posedge reset_i ) begin
      //representacao em excesso de 2
      if(reset_i) begin  col_r<=2; diag_r<=2; lin_r<=2; end
      else begin col_r<=col_i; diag_r<=col_r; lin_r<=valor_o; end
   end

   /*
    * Calculo combinacional do valor da celula da matriz,
    * segundo equacao de recorrencia:
    * - calcula diagonal, linha e coluna com penalidades
    * - calcula maximo entre estes valores
    * - se valor for negativo, escolhe 0 como valor final
    */
```

```
    //fios para logica combinacional - valores com penalidades
    wire [`SCORE_MAX_BITS-1:0] diag_p = (bbd_i==`BASE_CONSULTA) ?
          diag_r+`PONT_MATCH : diag_r-`PONT_MISMATCH;
//    Logica alternativa abaixo. Em testes, nao foi verificado ganho
//    wire [`SCORE_MAX_BITS-1:0] diag_p = diag_r+
//        ((bbd_i==`BASE_CONSULTA) ? `PONT_MATCH : -`PONT_MISMATCH);

    wire [`SCORE_MAX_BITS-1:0] lin_p  = lin_r-`PONT_GAP;
    wire [`SCORE_MAX_BITS-1:0] col_p  = col_r-`PONT_GAP;
    //calcula valor maximo entre fios acima
    wire [`SCORE_MAX_BITS-1:0] valor_o_tmp = lin_p > col_p
            ? (lin_p > diag_p ? lin_p : diag_p)
            : (col_p > diag_p ? col_p : diag_p);

    //alinhamento local, maximo entre valor_o_tmp e 2 (baseado em 2)
    assign valor_o = (valor_o_tmp>1) ? valor_o_tmp : 2'b10;

endmodule
```

A.5 Elemento de processamento - subcomponente de cálculo do máximo local

```
module eps_calc_max
    (
        input  clk_i, reset_i,
        input [`SCORE_MAX_BITS-1:0] valor_i,

        //sinal controla apresenta maximos
        input am_i,

        input  [`SCORE_MAX_BITS-1:0] max_i,
        output reg [`SCORE_MAX_BITS-1:0] max_o,

        input  [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_i,
        output reg [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_o,

        input [`CLK_TICKS_MAX_BITS-1:0] clk_ticks_i
    );

    /*
     * Calculo do valor maximo produzido por este elemento
     * reset das logicas abaixo eh reset_o ao inves de reset_i, evitando
     * tomar como maximos valores invalidos, produzidos antes do reset
     * conta o tempo desde o ultimo reset p/ ter o tempo de producao
     * do maximo
     */
    reg [`SCORE_MAX_BITS-1:0]      max_r; //maximo produzido
    reg [`CLK_TICKS_MAX_BITS-1:0] max_clk_ticks_r; //tempo de producao


    always @(`CLOCK_EDGE_PRINCIPAL clk_i or posedge reset_i ) begin
        if( reset_i ) begin
            max_r <= `SCORE_MAX_BITS'b10; //2
            max_clk_ticks_r <= `CLK_TICKS_MAX_BITS'b0;
        end
        else begin
            /* Nao estamos em reset. Entao (observe que max_clock_ticks_r
             * acompanha max_r), se:
             * - am_i ligado: "ao mesmo tempo" o maximo local estah sendo
             *       apresentado. Neste caso como o maximo jah foi
```

```
            *       apresentado, o registrador de maximo eh reiniciado com
            *       valor_o corrente, iniciando um novo ciclo de maximo
            * - ou se valor_o > max_r: novo maximo local, armazena-lo
            * - senao o valor do maximo permanece o mesmo (max_r <= max_r)
            */
           if( (valor_i > max_r) || am_i ) begin
              max_r <= valor_i; max_clk_ticks_r <= clk_ticks_i;
           end
        end
    end

    /*
    * O maximo de cada elemento de processamento eh passado p o elemento
    * aa direita e deste p os demais (estilo shift-register) de forma
    * que todos os maximos (e tempos) de todos os elementos cheguem ao
    * elemento mais aa direita do sistolico e entao para a saida
    * max_o e max_clock_ticks_o:
    * contem o valor maximo produzido por este sistolico (e o tempo
    * de producao) (se am_i ativado) ou carrega valor lido das entradas
    * max_i e max_clock_ticks_i
    */
    always @(`CLOCK_EDGE_PRINCIPAL clk_i ) begin
        if(am_i) begin max_o<=max_r; max_clk_ticks_o<=max_clk_ticks_r; end
        else begin max_o <= max_i; max_clk_ticks_o <= max_clk_ticks_i; end
    end

endmodule
```

Anexo B - Código-fonte do software

B.1 *Software* de reconstrução e visualização de alinhamento

```
#include "sw.h"

/*
 * GLOBAIS
 */
char sc[MAX_BASES+1], sbd[MAX_BASES+1];  //+1 para fgets
int sc_tam, sbd_tam;

//espaco para 3 antidiagonais. indice [0] sempre 0 (borda da matriz)
int ad[3][MAX_BASES+1]; int *ad_t0, *ad_t1, *ad_t2;

//espaco para 3 antidiagonais. indice [0] sempre 0 (borda da matriz)
int vd[3][MAX_BASES+1]; int *vd_t0, *vd_t1, *vd_t2;

//+1 para '\0' e printf
char sc_inv[MAX_BASES+1], sbd_inv[MAX_BASES+1];
int sc_inv_tam, sbd_inv_tam;

//matriz com os alinhamentos das sequencias invertidas
```

```
int m_alin_inv[MAX_BASES_ALIN][MAX_BASES_ALIN];
int m_alin_inv_vd[MAX_BASES_ALIN][MAX_BASES_ALIN];
//matriz com os alinhamentos das sequencias na ordem normal
int m_alin[MAX_BASES_ALIN][MAX_BASES_ALIN];

//global por performance em percorre_alinhamento
int bc_corte, bbd_corte;
//indices maximos observados na matriz de
//alinhamentos das sequencias invertidas
int m_alin_inv_max_bc = 0,m_alin_inv_max_bbd = 0;

//maiores valores de bbd e bc na matriz invertida onde ocorre o maximo
int maximo, maximo_num_dig;
int maximos_max_bc, maximos_max_bbd;

/*
 * PROTOTIPOS
 */
void corta_e_inverte_sequencias( ); void calcula_maximo_matriz();
void imprime_m_alin_inv(); void imprime_m_alin(); void le_sequencias();
void imprime_alinhamento( char *al_l1, char *al_l2,
      unsigned *al_score, unsigned al_i_inicio );
int percorre_alinhamento(   char al_l1[2*MAX_BASES_ALIN],
      char al_l2[MAX_BASES_ALIN], int al_score[MAX_BASES_ALIN],
      int bbd, int bc, int al_i );
int m_alin_inv_para_m_alin();
/*
 * Reconstroi um alinhamento maximo local, dada a posicao de termino
 * Argumentos: antidiagonal onde ocorreu o termino do alinhamento que se
 * deseja reconstruir, base de consulta deste alinhamento onde esta o
 * termino do alinhamento, score esperado para a posicao, para fim de
 * conferencia.
 *
 * - Corta as sequencias na posicao indicada e inverte-as
 * - Gera a matriz de similaridade de todos os alinhamentos que iniciem
 *   na posicao 1,1. A posicao 1,1 (inicio das seq invertidas) equivale
 *   a posicao indicada como termino dos alinhamentos nas sequencias
 *   normais.
 * - Verifica se o score maximo encontrado eh o esperado
 * - A partir de cada ponto com valor maximo na na matriz invertida
 *   percorre o caminho ate o inicio. Cada caminho eh um alinhamento,
 *   que se deseja visualizar. Gera a visualizacao
 */
```

```
main( int argc, char **argv )
{
    if( argc != 4 ) {
        printf( "\nSoftware complementar - reconstrucao de alinhamentos "
                "indicados por hardware\n--"\
                "\nAlinhamento local de sequencias pelo algoritmo de "
                "Smith-Waterman"\
                "\nInformar 3 argumentos: coluna onde ocorreu o termino "
                "do alinhamento, \ntempo de producao desta posicao e "
                "score esperado\n\n" );
        exit( RC_ARGS );
    }

    le_sequencias(); //sc, sc_tam, sbd, sbd_tam
#ifdef DEBUG
    printf( "\n%d bases na sequencia de consulta, %d bases na "
            "sequencia do banco de dados", sc_tam, sbd_tam );
#endif

    /*
     * Le os argumentos, a base de consulta onde ocorre o termino
     * do alinhamento a antidiagonal onde ocorreu este termino e o
     * score esperado para a posicao. Deduz, a partir da antidiagonal
     * e da base de consulta, a base do banco de dados
    */
    bc_corte = atoi(argv[1]); int score_esperado = atoi(argv[3]);
    bbd_corte = atoi(argv[2])-bc_corte;
#ifdef DEBUG
    printf( "\nSolicitados alinhamentos otimos que terminam na posicao: "
            "base de consulta %d, base do banco de dados %d "\
            "(antidiagonal %d)", bc_corte, bbd_corte, atoi(argv[1]) );
#endif

    /*
     * Corta as sequencias na posicao indicada e inverte-as
     * sc e sbd para sc_inv e sbd_inv
     */
    sc_inv_tam = bc_corte+1; sbd_inv_tam = bbd_corte+1;
    corta_e_inverte_sequencias( );

    //cada antidiagonal depende das duas anteriores.
    //armazena tb vetores de direcao
    memset( ad[0], 0, MAX_BASES+1 ); memset( ad[1], 0, MAX_BASES+1 );
```

```
    memset( ad[2], 0, MAX_BASES+1 );

    memset( vd[0], 0, MAX_BASES+1 ); memset( vd[1], 0, MAX_BASES+1 );
    memset( vd[2], 0, MAX_BASES+1 );

    ad_t0 = ad[0];  ad_t1 = ad[1]; ad_t2 = ad[2]; vd_t0 = vd[0];
    vd_t1 = vd[1]; vd_t2 = vd[2];

    /*
     * Calcula primeira antidiagonal fora do laco
     */
    vd_t0[0]=BIT_DIAG; //permite o surgimento de 1 alinhamento
//    printf( "\n" );
    calcula_ad_complementar( 0, 0, ad_t0, ad_t1, ad_t2 );
    vd_t0[0]=0;   //evita outros alinhamentos que nao o primeiro
    rotaciona_ponteiros( ad_t0, ad_t1, ad_t2 );
    rotaciona_ponteiros( vd_t0, vd_t1, vd_t2 );

    /*
     * Calculas proximas antidiagonais
     * variaveis bc_inv_i e bbd_inv_i marcam inicio das antidiagonais
     */
    int bc_inv_i, bbd_inv_i;
    //encontra o proximo valor correto apos a primeira [1,1]
    //proxima ad inicia em 1,2
    if( sbd_inv_tam > 1 ) { bc_inv_i=0; bbd_inv_i=1; }
    //proxima ad inicia em 2,1
    else if( sc_inv_tam  > 1 ) { bc_inv_i=1; bbd_inv_i=0; }
    else { printf("\nTamanho das sequencias menor igual a 1"); return; }
    for(;;) {

       //calcula antidiagonal ad_t2 (tempo 2) baseado nas ad_t0 e ad_t1
       if( ! calcula_ad_complementar( bc_inv_i, bbd_inv_i, ad_t0,
                                      ad_t1, ad_t2 ) )
         break;

       /*
        * Logica de incremento dos indices das sequencias de consulta e
        * do banco de dados onde inicia a proxima antidiagonal a ser
        * calculada. Condicao de parada deste laco
        * eh ultima antidiagonal calculada
        */
       //primeira parte das antidiagonais, avanca base do banco de dados
```

```
        //de inicio. Nao permite que se torne igual sbd_inv_tam
        if( bbd_inv_i < (sbd_inv_tam-1) )       bbd_inv_i++;
        //segunda parte, alcancou ultima base do banco de dados,
        //avanca base de consulta
        else if( bc_inv_i < (sc_inv_tam-1) )    bc_inv_i++;
        //fim, ultima antidiagonal foi calculada
        else break;

        //mais uma antidiagonal calculada, avanca o tempo
        rotaciona_ponteiros( ad_t0, ad_t1, ad_t2 );
        rotaciona_ponteiros( vd_t0, vd_t1, vd_t2 );
    }

    /*
     * Neste ponto, temos a matriz de similaridade e os vetores de
     * direcao das sequencias invertidas, iniciando no ponto
     * indicado pelos argumentos do programa. Supondo ser um ponto
     * de alinhamento maximo local, espera-se que nesta matriz
     * invertida surja o mesmo valor maximo que na matriz normal.
     * Os terminos de alinhamento que tiverem os valores maximos na
     * matriz invertida sao os inicios de alinhamento otimos na
     * matriz das sequencias na ordem normal
     *
     * Supondo que o alinhamento inidicado eh um alinhamento maximo
     * local, o maximo da matriz invertida deve ser o mesmo do maximo
     * da matriz normal. O codigo abaixo verifica esta condicao,
     * sinalizando se valores diferentes foram indicados. Se os valores
     * diferirem eh possivel que nao seja um alinhamento maximo local.
     * Neste caso, o mesmo pode nao ser observado quando as sequencias
     * forem invertidas
     */
    calcula_maximo_matriz(); //maximo, maximos_max_bbd, maximos_max_bc
    maximo_num_dig = calcula_digitos( maximo );

#ifdef DEBUG
    //imprime os alinhamentos invertidos que iniciam pto de maximo
    imprime_m_alin_inv( );
#endif

    if( maximo == score_esperado )
        printf( "\nScore maximo desta regiao de semelhanca confere "
                "com esperado: %d", maximo );
    else
```

```
        printf( "\nScore maximo desta regiao de semelhanca difere "
                "do esperado. Encontrado: %d Esperado: %d",
                maximo, score_esperado );

    /*
     * Para cada termino de alinhamento otimo da matriz invertida,
     * temos o inicio de um alihamento otimo na matriz com as sequencias
     * na ordem normal. Aqui inicia construcao dos alinhamentos normais
     */
    int alin_num = m_alin_inv_para_m_alin();
    printf( "\nVisualizacao dos alinhamentos na matriz de similaridade:"
            "\n %d alinhamento%s", alin_num, (alin_num!=1?"s":"") );

    imprime_m_alin();

    if( maximo != score_esperado )
        return RC_SCORE_DIF;

    return RC_OK;
}


/*
 * Calcula antidiagonal que inicia na posicao bc_inv_i e bbd_inv_i
 *
 * Os vetores de direcao em vd sao 3 bits BIT_DIAG BIT_LIN e BIT_COL
 * nao mutuamente exclusivos, uma celula pode ter vindo de dois
 * valores se ambos tiverem o mesmo valor apos o calculo da penalidade
 */
int
calcula_ad_complementar( int bc_inv_i, int bbd_inv_i )
{
    /*
     * Se ocorrerem duas ad sucessivas sem scorese != 0 (e vd != 0)
     * entao o alinhamento ao certo terminou
     */
    static int num_ad_zeradas_sucessivas = 0;
    //indica em celulas diferentes de zero neste ad
    int flag_ad_zerada = TRUE;
    //score maximo da antidiagonal e base de consulta deste
    int ad_max = 0, ad_max_bc = 0;
    /*
     * Percorre as posicoes desta antidiagonal,
```

```
    * incrementando a  base de consulta e decrementa do banco de dados
    * Provavelmente logica () && () abaixo eh redundante, condicoes
    * viram TRUE ao mesmo tempo
    */
   for( ; (bc_inv_i<sc_inv_tam) && (bbd_inv_i>=0);
          bc_inv_i++,bbd_inv_i-- ) {

      //deixa intocada ad_t2[0], borda da matriz
      int bc_inv_i_mais_1 = bc_inv_i+1;

      /* Calcula valores penalizados */
      //diagonal -> coluna anterior, 2 tempos atras
      int diag_p = ad_t0[bc_inv_i_mais_1-1] +
         ((sc_inv[bc_inv_i]==sbd_inv[bbd_inv_i])?
         PONT_MATCH:-(PONT_MISMATCH));
      //coluna   -> coluna anterior, 1 tempo atras
      int col_p  = ad_t1[bc_inv_i_mais_1-1] - PONT_GAP;
      //linha    -> mesma coluna, 1 tempo atras
      int lin_p  = ad_t1[bc_inv_i_mais_1]   - PONT_GAP;

      /* Computa maximo */
      if( diag_p > col_p ) {  //col nao eh o maior

         if( diag_p > lin_p ) {  //diag maior
            ad_t2[bc_inv_i_mais_1]=diag_p;
            vd_t2[bc_inv_i_mais_1]=BIT_DIAG;
         }
         else if( lin_p  > diag_p )  { //lin maior
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_LIN;
         }
         else /*lin e diag iguais*/  {
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_LIN|BIT_DIAG;
         }
      } else if( col_p > diag_p ) { //diag nao eh o maior
         if( col_p > lin_p ) { //col maior
            ad_t2[bc_inv_i_mais_1]=col_p;
            vd_t2[bc_inv_i_mais_1]=BIT_COL;
         }
         else if( lin_p > col_p ) { //lin maior
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_LIN;
```

```
        }
        else /*col e lin iguais*/ {
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_LIN|BIT_COL;
        }
    } else {  //col == diag
        if( lin_p > col_p ) {  //lin maior
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_LIN;
        }
        else if( col_p > lin_p ) {  //col e diag maiores
            ad_t2[bc_inv_i_mais_1]=col_p;
            vd_t2[bc_inv_i_mais_1]=BIT_COL|BIT_DIAG;
        }
        else /*lin, col e diag iguais*/ {
            ad_t2[bc_inv_i_mais_1]=lin_p;
            vd_t2[bc_inv_i_mais_1]=BIT_DIAG|BIT_COL|BIT_LIN;
        }
    }

    /*
     * A logica abaixo impede que surjam novos alinhamentos durante a
     * execucao do algoritmo, continuando somente os que iniciam-se
     * no canto superior esquerdo da matriz (primeira celula).
     * Isto porque o objetivo eh reconstruir o alinhamento indicado
     * por hardware, que inicia-se na primeira celula
     */
    //se negativo entao 0 (alinhamento local),
    if( ad_t2[bc_inv_i_mais_1] < 0 ) {
        ad_t2[bc_inv_i_mais_1] = 0;
        vd_t2[bc_inv_i_mais_1]=0;
    } else { //senao checar origem da celula

        //Testa as tres origens possiveis da celula,
        //se vem da celula diagonal mas vd daquela zerado,
        //ou seja, nao pertencia a alinhamento, zera BIT_DIAG
        //deste vd, impedindo novos alinhamentos
        if( (vd_t2[bc_inv_i_mais_1] & BIT_DIAG) &&
                (vd_t0[bc_inv_i_mais_1-1]==0) )
            vd_t2[bc_inv_i_mais_1]=vd_t2[bc_inv_i_mais_1]&(~BIT_DIAG);

        if( (vd_t2[bc_inv_i_mais_1] & BIT_COL) &&
                (vd_t1[bc_inv_i_mais_1-1]==0) )
```

```
            vd_t2[bc_inv_i_mais_1]=vd_t2[bc_inv_i_mais_1]&(~BIT_COL );

         if( (vd_t2[bc_inv_i_mais_1] & BIT_LIN) &&
                  (vd_t1[bc_inv_i_mais_1]  ==0) )
            vd_t2[bc_inv_i_mais_1]=vd_t2[bc_inv_i_mais_1]&(~BIT_LIN );

         //Se passou pelas checagens, entao esta ad tem
         //pelo menos 1 celula != 0. Sinaliza isso
         if( vd_t2[bc_inv_i_mais_1] ) { flag_ad_zerada=FALSE;
            //inclui na matriz de alinhamentos, zero-based
            m_alin_inv   [bbd_inv_i][bc_inv_i] = ad_t2[bc_inv_i_mais_1];
            m_alin_inv_vd[bbd_inv_i][bc_inv_i] = vd_t2[bc_inv_i_mais_1];
            //calcula os maiores indices da matriz invertida
            if( bc_inv_i  > m_alin_inv_max_bc )
               m_alin_inv_max_bc  = bc_inv_i;
            if( bbd_inv_i > m_alin_inv_max_bbd )
               m_alin_inv_max_bbd = bbd_inv_i;
         }
      }
#ifdef DEBUG
//       printf( "%d ", ad_t2[bc_inv_i_mais_1] );
#endif
   }

#ifdef DEBUG
//       printf( "\n" );
#endif

   if( flag_ad_zerada ) {
      //Na segunda antidiagonal sucessiva zerada, retorna FALSE pois
      //nenhum outro alinhamento pode surgir a partir daqui,
      if( ++num_ad_zeradas_sucessivas == 2 ) return FALSE;
   }
   else num_ad_zeradas_sucessivas = 0;

   return TRUE;
}

/*
 * Percorre os alinhamentos otimos terminados neste maximo (bbd_inv_i,
 * bc_inv_i), fazendo 2 tarefas:
 * - montando al_l1, al_l2 e al_score, vetores para exibicao do
 *   alinhamento, preenchidos dos indices al_inicio ate 2*MAX_BASES_ALIN-
```

```
 *   Ao termino do percorrimento de cada alinhamento, este eh impreso
 * - preenchendo matriz m_alin com os alinhamentos em ordem normal.
 *
 * Retorna o numero de alinhamentos encontrados e reconstruidos
 */
int
m_alin_inv_para_m_alin()
{
   //Armazena alinhamento sendo percorrido
   static char      al_l1   [2*MAX_BASES_ALIN];
   static char al_l2[2*MAX_BASES_ALIN];  //bases e '-'
   static int       al_score[2*MAX_BASES_ALIN];  //scores acumulados

   //indices para as sequencias em ordem normal (nao invertidas)
   int bbd_i, bc_i;
   for( bbd_i=0;   bbd_i <= m_alin_inv_max_bbd; bbd_i++ )
      for( bc_i=0; bc_i  <= m_alin_inv_max_bc;  bc_i++ )
         m_alin[bbd_i][bc_i] = -1;   //indica posicao sem uso

   //Cada ponto de maximo na matriz invertida (termino de alinhamnento)
   //eh inicio de alinhamento na matriz normal. Percorrer cada
   //alinhamento para exibicao
   printf( "\n" );
   int bbd_inv_i, bc_inv_i, alin_n = 1;
   for( bbd_inv_i=0;   bbd_inv_i <= m_alin_inv_max_bbd; bbd_inv_i++ )
      for( bc_inv_i=0; bc_inv_i  <= m_alin_inv_max_bc;  bc_inv_i++ )
         if( m_alin_inv[bbd_inv_i][bc_inv_i] == maximo ) {
            //ultimas bases do alinhamento colocadas em al_l1 e al_l2
            al_l1[2*MAX_BASES_ALIN-1] = sc_inv[bc_inv_i];
            al_l2[2*MAX_BASES_ALIN-1] = sbd_inv[bbd_inv_i];
            //percorre recursivamente o alinhamento, montando al_l1,
            //al_l2, al_score e m_alin
            int al_inicio = percorre_alinhamento( al_l1, al_l2,
                  al_score, bbd_inv_i, bc_inv_i, 2*MAX_BASES_ALIN-1);
            //imprime cabecalho para o alinhamento
            printf( "\nAlinhamento %d:", alin_n++ );
            printf( "\n %d bases da sequencia de consulta       "
                    "(%d a %d)",
                    bc_inv_i+1,  bc_corte-bc_inv_i, bc_corte );
            printf( "\n %d bases da sequencia do banco de dados "
                    "(%d a %d)",
                    bbd_inv_i+1, bbd_corte-bbd_inv_i, bbd_corte );
            //alinhamento percorrido, al_l1, al_l2 e al_score
```

```
            //preenchidos  com um alinhamento
            imprime_alinhamento( al_l1, al_l2, al_score, al_inicio );
        }

    return alin_n-1;  //desfaz ultimo incremento
}

/*
 * Percorre o alinhamento indicado pelo parametro recursivamente, do
 * final para o inicio, executando 2 funcoes:
 * - armazena alinhamento na forma de 2 linhas (uma acima da outra) em
 *   al_l1 e al_l2, imprimindo o alinhamento total quando chegar ao
 *   inicio deste
 * - copia as bases participantes do alinhamento para a matriz m_alin
 *
 * al_i: indice para construcao de al_l1, al_l2 e al_score, ou seja da
 * visualizacao do alinhamento em 2 linhas com score abaixo. No momento
 * da chamada "externa" da  funcao, fora da recursividade, al_i deve ser
 * iniciado para 2*MAX_BASES_ALIN que eh o numero maximo de posicoes
 * necessarias para representar uma sequencia em um alinhamento,
 * considerando a possibilidade de GAPs. Ex do resultado esperado:
 * A G C - A T C - A T A  - al_l1 (linha 1)
 * A G C T A T C C A T A  - al_l2 (linha 2)
 * ---------------------
 * 1 2 3 1 2 3 4 2 3 4 5  - al_scores
 *
 */
int
percorre_alinhamento( char al_l1[2*MAX_BASES_ALIN],
    char al_l2[MAX_BASES_ALIN],
    int al_score[MAX_BASES_ALIN], int bbd, int bc, int al_i )
{
    //calcula o score da posicao corrente no alinhamento normal baseado
    //no score do invertido
    int score;
    score = m_alin[maximos_max_bbd-bbd][maximos_max_bc-bc] =
    maximo-m_alin_inv[bbd-1][bc-1];

    al_score[al_i--] = score;
    if( m_alin_inv_vd[bbd-1][bc-1] & BIT_DIAG ) {
        bc--; bbd--;  //bases alinhadas
        al_l1[al_i] = sc_inv[bc]; al_l2[al_i] = sbd_inv[bbd];
        return percorre_alinhamento(al_l1,al_l2,al_score,bbd,bc,al_i);
```

```
    }
    if( m_alin_inv_vd[bbd-1][bc-1] & BIT_LIN  ) {
        bbd--; //base com gap
        al_l1[al_i] = '-'; al_l2[al_i] = sbd_inv[bbd];
        return percorre_alinhamento(al_l1,al_l2,al_score,bbd,bc,al_i);
    }
    if( m_alin_inv_vd[bbd-1][bc-1] & BIT_COL  ) {
        bc--; //base com gap
        al_l1[al_i] = sc_inv[bc]; al_l2[al_i] = '-';
        return percorre_alinhamento(al_l1,al_l2,al_score,bbd,bc,al_i);
    }

    //este codigo somente sera alcancado na primeira base de cada
    //alinhamento, com o vetor de direcoes zerado
    //(~BIT_DIAG & ~BIT_COL & ~BIT_LIN)
    return al_i+1;  //desfaz ultimo decremento
}


/*************************************
 * FUNCOES AUXILIARES
 *************************************/
void
le_sequencias()
{
    FILE *fd = fopen( ARQ_SEQUENCIAS, "r" );
    if( ! fd ) {
        printf( "Erro ! Nao pude abrir %s", ARQ_SEQUENCIAS );
        abort();
    }

    fgets( sc, MAX_BASES, fd );
    sc_tam = strlen(sc);
    //remove possivel \n
    if( sc[sc_tam-1]=='\n' ) sc_tam--;

    fgets( sbd, MAX_BASES, fd );
    sbd_tam = strlen(sbd);
    if( sbd[sbd_tam-1]=='\n') sbd_tam--;
}

void
corta_e_inverte_sequencias()
```

```
{
    int i;
     for( i = 0; i < sc_inv_tam;  i++ )  sc_inv  [i]  = sc [bc_corte -i];
     for( i = 0; i < sbd_inv_tam; i++ )  sbd_inv [i]  = sbd[bbd_corte-i];

#ifdef DEBUG
    sc_inv[sc_inv_tam] = '\0'; sbd_inv[sbd_inv_tam] = '\0';
    printf( "\n\nSequencia consulta invertida (tamanho %d): %s",
                sc_inv_tam, sc_inv );
    printf( "\nSequencia banco de dados invertida (tamanho: %d): %s",
                sbd_inv_tam, sbd_inv );
#endif
}

/*
 * Percorre matriz de alinhamentos invertidos
 * e calcula o maximo e a posicao
 */
void
calcula_maximo_matriz()
{
    maximos_max_bbd = maximos_max_bc = maximo = 0;
    int bbd, bc;
    for( bbd=0; bbd <= m_alin_inv_max_bbd; bbd++ )
        for( bc=0; bc <= m_alin_inv_max_bc; bc++ ) {
            if( m_alin_inv[bbd][bc] > maximo )
                maximo = m_alin_inv[bbd][bc];
            if( m_alin_inv[bbd][bc] == maximo ) {
                //recebem o valor somado de um pois matriz eh
                //zero-based mas bases nao
                if( bbd > maximos_max_bbd ) maximos_max_bbd = bbd;
                if( bc  > maximos_max_bc  ) maximos_max_bc  = bc;
            }
        }
}

int
calcula_digitos( int numero )
{
    int digitos;
    for( digitos=1; (numero=numero/10) >= 1; digitos++ );

    return digitos;
```

```
}

/*
 * Imprime alinhamento no formato:
 *
 * A G C - A T C - A T A  - al_l1 (linha 1)
 * A G C T A T C C A T A  - al_l2 (linha 2)
 * -----------------------
 * 1 2 3 1 2 3 4 2 3 4 5  - arg3 al_scores (scores)
 *
 * vetores vao de al_i_inicio ateh 2*MAX_BASES, pois gerados invertidos
 */
void
imprime_alinhamento( char *al_l1, char *al_l2, unsigned *al_score,
                     unsigned al_i_inicio )
{
    //calcula espacos necessarios para alinhar exibicao, baseado no
    //numero de digitos do score
    //espacos entre bases p alinhar scores
    char espacos_bases[11] = "          ";
    espacos_bases[ maximo_num_dig ] = '\0';  //se 1 digito, sem espacos


    unsigned al_i;
    //imprime sequencia de consulta (primeira linha)
    printf( "\n\n " );
    for( al_i=2*MAX_BASES_ALIN-1; al_i >= al_i_inicio; al_i-- )
        printf( "%s%c", espacos_bases, al_l1[al_i] );
    //imprime sequencia do banco de dados (segunda linha)
    printf( "\n " );
    for( al_i=2*MAX_BASES_ALIN-1; al_i >= al_i_inicio; al_i-- )
        printf( "%s%c", espacos_bases, al_l2[al_i] );
    //imprime linha dividindo sequencias do score acumulado '-----'
    unsigned i;
    printf( "\n " );
    for( al_i=2*MAX_BASES_ALIN-1; al_i >= al_i_inicio; al_i-- )
        for( i=0; i<=maximo_num_dig; i++)
            printf( "-" );
    printf( "\n " );
    //imprime score acumulado
    for( al_i=2*MAX_BASES_ALIN-1; al_i >= al_i_inicio; al_i-- )
        printf( "%*d", maximo_num_dig+1, al_score[al_i] );
    printf( "\n" );
```

```
}

void
imprime_m_alin_inv( )
{
   int bbd_nd = calcula_digitos( bbd_corte ), bbd,bc;

   printf( "\n\nRegiao do alinhamento solicitado abaixo, "
           "sequencias invertidas" );
   printf( "\n %d bases da sequencia de consulta        (%d a %d)",
            m_alin_inv_max_bc+1, bc_corte, bc_corte-m_alin_inv_max_bc );
   printf( "\n %d bases da sequencia do banco de dados (%d a %d)",
            m_alin_inv_max_bbd+1, bbd_corte,
            bbd_corte-m_alin_inv_max_bbd );
   printf( "\n Maximo: %d. Maiores indices onde ocorre maximo: "
           "base de consulta %d, base do banco de dados %d",
          maximo, maximos_max_bc, maximos_max_bbd );

   //imprime sequencia de consulta
   printf( "\n\n%*s  %d %d ... %d", bbd_nd+maximo_num_dig, "",
         bc_corte, bc_corte-1,
         bc_corte-m_alin_inv_max_bc );  //numero das bases de consulta
   printf( "\n%*s  ", bbd_nd, "" ); imprime_seq_espacada( sc_inv, 0,
         m_alin_inv_max_bc, maximo_num_dig );  //seq de consulta
   //imprime linha a a linha a matriz com os alinhamentos
   for( bbd=0; bbd <= m_alin_inv_max_bbd; bbd++ ) {
      //imprime a base bd desta linha
      printf( "\n%*d %c ", bbd_nd, bbd_corte-bbd, sbd_inv[bbd] );
      //percorer as colunas (bc) desta linha
      for( bc=0; bc <= m_alin_inv_max_bc; bc++ )
         if( m_alin_inv_vd[bbd][bc] != 0 )
            printf( "%*d ", maximo_num_dig, m_alin_inv[bbd][bc] );
         else printf( "%*s ", maximo_num_dig, "" );
   } printf( "\n" );
}

/*
 * Conta com o fato do tamanho do alinhamento ser
 * o mesmo, invertido ou na ordem normal
 */
void
imprime_m_alin()
{
```

```
    printf( "\n %d bases da sequencia de consulta        (%d a %d)",
          maximos_max_bc+1, bc_corte-maximos_max_bc, bc_corte );
    printf( "\n %d bases da sequencia do banco de dados (%d a %d)\n",
          maximos_max_bbd+1, bbd_corte-maximos_max_bbd, bbd_corte );

    //imprime sequencia de consulta
    int  bc  = bc_corte-maximos_max_bc;
    int bbd = bbd_corte-maximos_max_bbd;
    int bbd_nd = calcula_digitos( bbd_corte );
    //numero das bases de consulta
    printf( "\n%*s  %d %d ... %d",
          bbd_nd+maximo_num_dig, "", bc, bc+1, bc_corte );
    printf( "\n%*s  ", bbd_nd, "" );
    imprime_seq_espacada( sc, bc, bc_corte, maximo_num_dig );

    //os alinhamentos inciam no termino menos o numero maximo de bases
    //participantes nos alinhamentos otimos
    int m_alin_bbd, m_alin_bc;
    for( m_alin_bbd = 0; m_alin_bbd <= maximos_max_bbd;
          m_alin_bbd++, bbd++ ) {
          //imprime a base bd desta linha
       printf( "\n%*d %c ", bbd_nd, bbd, sbd[bbd] );
       for( m_alin_bc = 0; m_alin_bc <= maximos_max_bc; m_alin_bc++ )
          if( (m_alin[m_alin_bbd][m_alin_bc] != -1) )
             printf( "%*d ", maximo_num_dig,
                   m_alin[m_alin_bbd][m_alin_bc] );
          else printf( "%*s ", maximo_num_dig, "" );
    }
    printf( "\n\n" );
}
```