Implementação paralela do Algoritmo Smith-Waterman utilizando threads POSIX

Gustavo Lermen, Daniela Saccol Peranconi, Gerson Geraldo H. Cavalheiro

Centro de Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
São Leopoldo - RS - Brasil

gustavo@prisma.unisinos.br, danielap@exatas.unisinos.br, gersonc@exatas.unisinos.br

Introdução

O estudo de seqüências de DNA é empregado em vários campos, como o estudo do genoma humano, a descoberta de causas de doenças e mesmo como prova em tribunais. Atualmente existe uma grande quantidade de bancos de dados públicos disponibilizando livremente seqüências de DNA para serem pesquisadas. A necessidade de um aumento na velocidade do tratamento de seqüências de DNA vem do crescimento exponencial destes bancos (todo ano o tamanho deles cresce em um fator entre 1.5 e 2 [SCH 02]) e da grande quantidade de cálculo envolvido [YAP 98].

Vários algoritmos para comparação e análise destas seqüências foram propostos, cada um com suas características específicas, alguns priorizando velocidade de execução, outros priorizando a precisão da análise [MOU 01]. O algoritmo Smith-Waterman é um dos algoritmos que prioriza a precisão da análise. Ele utiliza o método de programação dinâmica para encontrar alinhamentos locais entre seqüências de DNA. Este método garante que o alinhamento encontrado seja ótimo. Devido a ele priorizar a precisão da análise, seu tempo de resposta é mais alto que outros algoritmos, pois uma grande quantidade de cálculo é necessária. Este artigo apresenta uma solução concorrente para o algoritmo Smith-Waterman e sua implementação.

Caracterização do Problema

O algoritmo Smith-Waterman utiliza programação dinâmica como método computacional para encontrar alinhamentos locais ótimos entre duas seqüências de DNA. O alinhamento entre duas seqüências consiste em estabelecer uma correspondência entre os símbolos das duas, obedecendo a ordem dos símbolos, e dando prioridade a procurar fazer corresponder símbolos iguais. Um alinhamento encontrado contém caracteres que combinam (iguais), caracteres que não combinam e também espaços que foram eventualmente inseridos em ambas as seqüências. O alinhamento resultante faz com que o número de caracteres que combinam seja o maior possível.

É provado matematicamente que o método de programação dinâmica para alinhamento de seqüências produz um alinhamento ótimo entre duas seqüências [GUS 97]. Alinhamentos ótimos fornecem informações muito úteis para biólogos a respeito da relação das seqüências analisadas. A partir destas informações, pode-se prever informações estruturais, funcionais e evolucionarias das seqüências [GUS 97].

A partir do algoritmo de programação dinâmica pode-se obter alinhamentos globais e locais com pequenas mudanças no mesmo. Outra característica importante do método de programação dinâmica é que os alinhamentos obtidos dependem do sistema de pontuação utilizado na comparação de pares de caracteres e também da pontuação dada aos espaços inseridos. Para seqüências de proteínas, o sistema mais simples de comparação é baseado na identidade. Uma combinação em um alinhamento é pontuada somente se os caracteres sendo comparados são iguais.

Descrição do algoritmo

O algoritmo para calcular o alinhamento local ótimo entre as seqüências sendo analisadas consiste de duas partes: o cálculo da pontuação total indicando a similaridade entre as duas seqüências em analise e a identificação do(s) alinhamento(s) que levam a esta pontuação. Este trabalho se concentra na primeira parte do algoritmo pois ela é a mais cara computacionalmente. A idéia do algoritmo é chegar a solução final utilizando para isto soluções parciais obtidas de subseqüências menores calculadas anteriormente. A comparação de duas seqüências X e Y, utilizando este algoritmo, é mostrada na Tabela 1. As seqüências são colocadas na margem esquerda (Y) e na margem superior (X) da matriz de similaridades. Inicialmente a primeira linha e a primeira coluna da matriz são preenchidas com zeros demarcando o final de um alinhamento local. Os demais elementos da matriz são calculados considerando sua vizinhança. O valor S(i,j) é encontrado a partir dos seguintes valores: o elemento da esquerda mais a penalidade de espaço, o elemento da diagonal esquerda mais o valor da similaridade dos caracteres sendo analisados e o elemento acima mais a penalidade de espaço.

Tabela 1- Matriz de similaridades gerada a partir das seqüências ATCAGATC e GTCAGTCA

	(X)	A	T	C	A	G	A	G	T	C
(Y)	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	1	0	1	0	0
T	0	0	1	0	0	0	0	0	2	0
C	0	0	0	2	0	0	0	0	0	3
A	0	1	0	0	3	1	1	1	1	1
G	0	0	0	0	1	4	2	2	2	2
T	0	0	1	0	1	2	3	1	3	1
C	0	0	0	2	1	2	1	2	1	4
A	0	1	0	0	3	2	3	1	1	2

Para o caso geral onde X= x1...xn e Y= y1...yn, para i de 1..n e j de 1..m, a matriz de similaridade S[n.m] é calculada a partir da equação [1], onde *gp* é a penalidade por um espaço inserido e *ss* é a pontuação pela substituição:

$$S[i,\ j] = \max\begin{Bmatrix} \max\{S[0,\ j]...S[i-1,\ j]\} + gp, \\ \max\{S[i,0]...S[i,\ j-1]\} + gp, \\ S[i-1,\ j-1] + ss(Xi, Yj), \\ 0 \end{Bmatrix} \qquad [1]$$

No exemplo mostrado a penalidade por espaço inserido (*gp*) utilizada foi de –2. A pontuação de substituição (*ss*) utilizada foi +1 se os caracteres sendo comparados são iguais e –1 caso contrário. Entretanto outros valores podem ser utilizados.

Seguindo estas recorrências, a matriz é preenchida a partir da célula mais superior à esquerda até a célula mais inferior à direita. Desta maneira, para obter o valor de cada célula [i,j] os valores das células [i-1,j], [i,j-1] e [i-1.j-1] devem ser conhecidos.

Depois de todos os valores da matriz de similaridades terem sido calculados, o alinhamento local ótimo é encontrado da seguinte maneira: a célula da matriz com a pontuação mais alta é encontrada e a partir desta célula o alinhamento é construído, sempre se escolhendo a célula vizinha ([i-1,j-1],[i-1,j] ou [i,j-1]) com o valor mais alto. O alinhamento local termina quando uma célula com o valor zero é encontrada.

O alinhamento local resultante da matriz da Tabela 1 é mostrado na Tabela 2.

Tabela 2 - Alinhamento local resultante da Tabela 1

A	**T**	**C**	**A**	**G**	**A**	**G**	**T**	**C**	
G	**T**	**C**	**A**	**G**	-	-	**T**	**C**	A

Modelo da Solução Concorrente

Uma versão paralela do algoritmo apresentado na seção anterior deve lidar com as dependências apresentadas e ao mesmo tempo deve tentar manter os processadores envolvidos no cálculo ocupados a maior parte possível do tempo. Dadas as dependências de dados do algoritmo, a matriz de similaridades pode ser preenchida linha a linha, coluna a coluna ou ainda pelas anti-diagonais. O problema das duas primeiras estratégias é que a maioria dos elementos em uma linha ou coluna da matriz depende diretamente dos outros elementos da mesma linha ou coluna. Desta maneira, as linhas ou colunas não podem ser calculadas em paralelo. A outra estratégia apresentada, que calcula a matriz por suas anti-diagonais não apresenta este problema, pois uma antidiagonal depende somente dos elementos das outras anti-diagonais previamente calculadas. Mesmo apresentando um possível paralelismo, esta estratégia também apresenta problemas para uma implementação paralela eficiente. Um dos problemas é que o tamanho das anti-diagonais varia durante o preenchimento da matriz, causando assim um trabalho não balanceado entre os processadores.

Outro problema desta estratégia está no número de elementos a serem calculados por cada processador. Se cada processador ficar responsável por calcular um elemento da matriz, o número de processadores necessários será muito grande se forem levados em consideração dados biológicos reais, causando desta maneira um *overhead* de comunicação muito elevado.

Uma solução para este problema é dividir a matriz de similaridades em blocos retangulares, como mostrado na Tabela 3. Neste exemplo o programa calcula o bloco 1, logo em seguida o bloco 2 e depois o bloco 5 é calculado visto que todos os elementos necessários para o inicio do seu cálculo já teriam sido calculados nos blocos 1 e 2. Toda a matriz de similaridades seria calculada desta forma [MAR 01]..

Se cada bloco tem "*q*" linhas e "*r*" colunas, então a computação de um dado bloco requer somente o segmento de linha imediatamente acima do bloco, o segmento de coluna imediatamente a esquerda e o elemento acima a esquerda, um total de $q + r + 1$ elementos. Por exemplo se cada bloco tem 4 linhas e 4 colunas, então cada bloco irá calcular 16 valores após ter recebido 9 valores de entrada.

Tabela 3 - Divisão da matriz de similaridades em blocos

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

Esta estratégia ainda apresenta uma dependência de dados, o que prejudica o paralelismo do programa. Entretanto, dado o tamanho dos problemas atuais e as máquinas paralelas utilizadas atualmente, isso não é um fator limitante [MAR 01].

O problema do balanceamento de carga pode ser resolvido atribuindo cada linha de blocos, ou tira, para o mesmo processador. Isto faz com que os processadores mantenham-se ocupados durante o cálculo dos elementos da matriz.

Outra solução é criar dinamicamente threads para tratar cada bloco, alocando-as aos processadores à medida que as dependências sejam resolvidas. Desta maneira, em uma primeira etapa, as threads que efetuam os cálculos de todos os blocos são criadas e à medida que suas dependências são calculadas, sua execução é iniciada. Utilizando esta estratégia, o *overhead* de criar as threads é eliminado visto que elas já foram criadas e estão apenas aguardando por suas dependências. Esta estratégia pode ser implementada com a utilização da biblioteca *Anahy* [CAV 03].

Implementação

Nesta seção são apresentadas duas soluções para a implementação. A solução A com a utilização de *pthreads* e a solução B, que utiliza a biblioteca *Anahy*.

Solução A: implementação com C++ e *threads* POSIX (*pthreads)*. A estratégia utilizada foi a divisão da matriz de similaridades em blocos de "*q"* linhas e "*r"* colunas. Nesta estratégia, para cada bloco é criada uma *thread* que calcula todos os elementos do bloco. A questão das dependências entre os blocos foi resolvida através do uso de variáveis de condição. Desta maneira, quando todos os elementos de um bloco são calculados, uma variável de condição é utilizada para indicar aos blocos vizinhos que o cálculo de seus elementos já pode ser iniciado. O uso de variáveis de condição se faz necessário pois no modelo *pthreads*, uma *thread* não pode executar um *join* em mais de uma *thread*.

Solução B: utilização da biblioteca *Anahy*. Esta estratégia dispensa o uso de variáveis de condição para tratar a dependência entre os blocos. As dependências podem ser resolvidas apenas com o uso das primitivas *athread_create* e *athread_join*.

Conclusão

Este artigo apresentou um modelo para a solução concorrente do algoritmo Smith-Waterman e sua respectiva implementação utilizando *threads* POSIX.

O objetivo deste trabalho foi utilizar *threads* POSIX para diminuir o tempo de execução na comparação de duas seqüências, sem penalizar com isto a precisão do resultado.

Os trabalhos atuais são de avaliação de desempenho das estratégias propostas em arquiteturas SMP (*Symetric Multiprocessing*).

Referências

[CAV 03] CAVALHEIRO, G. G. H., Real, Lucas C. Villa , Dall'agnol, Evandro C. **Uma biblioteca de Processos Leves para a Implementação de Aplicações Altamente Paralelas.** IV Workshop de Sistemas Computacionais de Alto Desempenho. São Paulo, 2003.

[GUS 97] GUSFIELD, Dan. **Algorithms on Strings, Trees and Sequences**: Computer Science and Molecular Biology. Cambridge University: New York, 1997. 534p.

[MAR 01] MARTINS, W. S. et al. **A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison**. Pacific Simposium on Biocomputing, 6:311-322, 2001.

[MOU 01] MOUNT, David W. **Bioinformatics**: Sequence and Genome Analysis. Cold Spring Harbor Laboratory: New York, 2001. 564p.

[PEA 92] PEARSON, W. R., Miller, W. **Dynamic Programming algorithms for biological sequence comparison**. Methods in Enzymology, 210:575-601, 1992.

[SCH 02] SCHMIDT, Bertil; Schimmler, Manfred. **Massively Parrallel Solutions for Molecular Sequence Analysis.** Proceedings of the International Parallel and Distributed Processing Symposium. IEEE, 2002.

[SMI 81] SMITH, Temple F.; Waterman, Michael S. **Identification of Common Molecular Subsequences**. Journal of Molecular Biology, 147:195-197, 1981.

[YAP 98] YAP, Tieng K.; Frieder, Ophir and Martino, Robert L. **Parallel Computation in Biological Sequence Analysis.** IEEE Transactions on Parallel and Distributed Systems, 9:3 283-294, 1998.