Framework para alinhamento de seqüências biológicas com o auxílio de programação concorrente

Gustavo Lermen Daniela Saccol Peranconi[1] Gerson Geraldo H. Cavalheiro[2]

Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
São Leopoldo – RS – Brasil
gustavo@prisma.unisinos.br, {danielap,gersonc}@exatas.unisinos.br

Resumo

Este artigo apresenta um framework para implementação de algoritmos de alinhamento de seqüências biológicas, com o diferencial de oferecer suporte à execução concorrente. O objetivo do trabalho é oferecer recursos computacionais que facilitem a extração de informações estruturais, funcionais e evolucionárias de pares de seqüências de DNA ou proteínas em ambiente de processamento paralelo. A avaliação dos resultados obtidos foi realizada através da implementação de algoritmos que utilizam o método de programação dinâmica e por uma análise de desempenho. O artigo é complementado por uma análise do uso do modelo de programação de Anahy e de seu núcleo executivo.

1. Introdução

O estudo de seqüências biológicas, em especial seqüências de DNA, é empregado em vários campos, como a pesquisa do genoma humano, a descoberta de causas de doenças e como prova em tribunais. Atualmente, existe uma grande quantidade de bancos de dados públicos disponibilizando livremente seqüências de DNA para serem analisadas. Porém, dada a quantidade de informações contidas nestas bases de dados, pouco avanço pode ser obtido sem o emprego de ferramentas computacionais especialmente desenvolvidas para este fim. Assim, surge a necessidade do desenvolvimento de softwares para análise de seqüências que garantam absorção de conhecimento sobre o DNA e suas aplicações.

A necessidade de um aumento na velocidade do tratamento de seqüências de DNA vem do crescimento exponencial dos bancos de dados públicos [19] e da grande quantidade de cálculo envolvido [21]. Vários algoritmos para comparação e análise destas seqüências foram propostos [16], sendo que os que empregam o método de programação dinâmica oferecem resultados mais precisos, a custo computacional elevado, em termos de tempo de processamento e consumo de memória. Exemplos de soluções empregando processamento paralelo são encontrados na literatura ([14],[1]).

Neste trabalho é proposto um *framework* para a implementação de algoritmos de alinhamento de DNA. Um dos diferenciais deste *framework* é a inclusão de uma estrutura para suportar a execução concorrente destes algoritmos. A estrutura adotada permite que a concorrência das operações de alinhamento seja descrita sem considerar as ferramentas utilizadas para a efetiva execução paralela destas. Esta característica viabiliza a extensão do *framework* a diferentes suportes ao paralelismo. No momento, o *framework* conta com duas opções de suporte à execução paralela: *threads* POSIX e Anahy [6][9], sendo dada uma atenção especial a Anahy, que permite introduzir um nível ainda maior de abstração na implementação de aplicações concorrentes.

A construção do *framework* foi baseada no estudo de algoritmos de alinhamento de seqüências, sendo modeladas as classes considerando características comuns entre os algoritmos. No atual estágio, foi dada maior atenção aos algoritmos que empregam o método de programação dinâmica.

O *framework* construído foi avaliado através de sua instanciação com a implementação de algoritmos de alinhamento de seqüências, os quais foram igualmente descritos de maneira a serem executados de forma con-

[1] Bolsista PROSUP/CAPES – Programa Interdisciplinar de Pós-Graduação em Computação Aplicada.
[2] Programa Interdisciplinar de Pós-Graduação em Computação Aplicada.
Este trabalho foi parcialmente financiado pelo CNPq (55.2196/02-9) e desenvolvido em colaboração com a HP Brasil P&D.

corrente. Uma avaliação de desempenho sobre arquiteturas SMP (*Symmetric MultiProcessors*) foi realizada.

A próxima seção apresenta os conceitos gerais sobre o alinhamento de seqüências e sobre o método de programação dinâmica. A seção 3 apresenta as principais características de Anahy. A seção 4 apresenta o *framework* proposto e a introdução do modelo da concorrência. As seções finais, 5 e 6, apresentam, respectivamente, uma avaliação do *framework*, através de resultados de desempenho obtidos e as conclusões deste trabalho.

2. Alinhamento de seqüências

Alinhar duas seqüências consiste em estabelecer uma correspondência entre seus símbolos, obedecendo a ordem destes, procurando fazer com que símbolos iguais nessas seqüências se correspondam. Um alinhamento encontrado contém caracteres que combinam (iguais) e caracteres que não combinam. Eventualmente, pode conter espaços (*gaps*) introduzidos em uma das seqüências sendo comparadas de forma a aumentar o número de caracteres que combinam. O alinhamento pode ser realizado tanto de forma global como local. No primeiro caso, é considerado o comprimento total das seqüências analisadas. No segundo, somente trechos das seqüências onde a similaridade é mais alta.

Dentre as técnicas computacionais utilizadas para alinhar duas seqüências, destaca-se o método de programação dinâmica, que se caracteriza por prover o alinhamento ótimo [13] entre as duas seqüências. Com o emprego deste método, é possível obter tanto alinhamentos globais como locais, através de modificações no algoritmo básico. Uma característica importante, é que os alinhamentos obtidos dependem do sistema de pontuação utilizado na comparação de pares de símbolos e também da pontuação dada aos espaços inseridos.

O algoritmo básico utilizado para calcular o alinhamento entre duas seqüências é composto por dois passos: (*i*) cálculo da pontuação total indicando a similaridade entre as duas seqüências em análise e (*ii*) identificação do(s) alinhamento(s) que levam a esta pontuação. A idéia do algoritmo é avançar até a solução final obtendo soluções parciais a cada passo.

A comparação de duas seqüências X e Y, utilizando este método para obtenção de um alinhamento local, é mostrada na Figura 1. O comprimento destas seqüências é de n = 9 para X e m = 8 para Y. 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. No caso de um alinhamento global, a primeira linha e coluna devem possuir outros valores iniciais [17]. Os demais elementos são calculados considerando sua vizinhança. Um valor S(i,j) pertencente à matriz é calculado a partir de seus vizinhos: S(i-1,j) mais a penalidade por espaço inserido, S(i-1,j-1) mais o valor da similaridade dos caracteres sendo analisados e S(i,j-1) mais a penalidade por espaço inserido.

	(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

Figura 1. Matriz de similaridades gerada a partir de duas seqüências.

Para o caso geral onde $X=x_1...x_n$ e $Y=y_1...y_m$, para i=1...n e j=1...m, a matriz de similaridades é calculada a partir da equação (1), onde *gp* é um índice de penalidade por um espaço inserido e *ss* é a pontuação pela substituição:

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

Com a matriz de similaridades completamente preenchida, o primeiro passo do algoritmo é encerrado. O segundo passo consiste em uma operação para encontrar o alinhamento propriamente dito entre as duas seqüências. Partindo-se da célula da matriz com a pontuação mais alta, o alinhamento é construído escolhendo-se sempre a célula vizinha com o valor mais alto. O alinhamento local termina quando uma célula com o valor zero é encontrada.

As vantagens dos algoritmos que implementam o modelo de programação dinâmica estão associadas à possibilidade de obtenção de alinhamentos ótimos entre duas seqüências. Porém, o algoritmo necessita de grande quantidade de recursos computacionais para atender seus requisitos de processamento e memória, uma vez que estes são proporcionais ao produto do número de símbolos de cada seqüência. As implementações paralelas de tais algoritmos devem, portanto, utilizar eficientemente os recursos computacionais da arquitetura de suporte à execução.

3. Anahy

Anahy [6][9] é um ambiente de processamento para exploração de alto desempenho em aglomerados de computadores e arquiteturas SMP, estando uma versão operacional disponível para esta última classe de arquiteturas. A principal característica de Anahy é dissociar a

descrição da concorrência da aplicação da forma como o paralelismo da arquitetura é explorado. Esta característica é buscada considerando que a concorrência natural que pode ser explorada em uma aplicação é, em geral, superior à capacidade de execução paralela das arquiteturas disponíveis atualmente.

A proposta de Anahy é de oferecer uma abstração de programação de alto nível onde seja possível explicitar as atividades concorrentes de um programa (tarefas), assumindo a responsabilidade de explorar os recursos de processamento (processadores e memória) disponíveis. Assim, o ambiente propõe uma interface de programação baseada em um modelo de fluxo de dados e um núcleo executivo dotado de uma política de escalonamento explorando a criação preguiçosa de *threads* ([4], [5], [7], [12]).

3.1 Arquitetura virtual

A arquitetura virtual definida para execução de programas Anahy conta com um conjunto de processadores compartilhando acesso a um espaço de endereçamento compartilhado. Uma máquina virtual é instanciada com esta arquitetura para oferecer suporte à execução de um único programa Anahy. Durante o tempo em que a máquina virtual estiver ativa, cada processador virtual (PV) pode se encontrar ou processando uma unidade de trabalho gerada pelo programa em execução ou aguardando que um trabalho lhe seja atribuído. O conjunto de instruções suportado pelos PVs é o mesmo que o oferecido pelas linguagens imperativas seqüenciais tradicionais, tendo sido acrescentadas duas novas instruções para acesso ao espaço de endereçamento compartilhado: uma para efetuar a leitura e outra para a escrita de dados necessários/produzidos para uma tarefa.

3.2 Fluxo de dados

A definição do modelo de programação concorrente de Anahy condiz com as seguintes considerações sobre a execução de programas seqüenciais:

- **Execução determinista**: para um determinado conjunto de dados de entrada, o resultado da computação é sempre o mesmo.
- **Efeito colateral**: a execução de uma instrução modifica um dado em memória. Uma posição de memória alterada por uma instrução pode ser entrada de uma outra instrução.
- **Expressão implícita de concorrência**: um certo nível de concorrência pode ser explorado pela análise da dependência de dados entre as instruções.
- **Sincronização implícita**: a seqüência de execução é definida pelo fluxo de execução único previsto pelo paradigma. Tal seqüência define uma ordem lexicográfica para execução das instruções limitando a concorrência entre estas. Esta ordem deve ser garantida para correta execução do programa.

Seguindo as premissas acima, o modelo de programação de Anahy propõe uma estrutura baseada em grafo de tarefas para controle da concorrência de execução. Este grafo é construído e mantido em tempo de execução, sendo composto por nodos, representando as tarefas (unidades de trabalho), e arcos direcionados, representando o fluxo de dados entre as tarefas. A análise deste grafo possibilita identificar as seqüências de tarefas que podem ser executadas de forma concorrente e quais possuem dependências entre si.

A interface de programação de Anahy encapsula tarefas em um nível de abstração mais alto, oferecendo uma interface de programação baseada em operações do tipo *fork/join*, de forma semelhante as tradicionais bibliotecas de programação com *threads* baseadas no padrão POSIX. A operação de *fork* tem por objetivo construir uma nova *thread* recebendo para tanto dois parâmetros: uma função F, para ser executada pela nova *thread* e um conjunto de dados necessários à execução de F. O retorno a uma chamada *fork* consiste em um identificador para a nova *thread* criada.

A sincronização com o término de uma *thread* é possível através de uma chamada à operação *join*, informando o identificador da *thread* a ser sincronizada. Esta operação permite que a *thread* realizando a sincronização bloqueie aguardando o término da *thread* sincronizada de forma a obter os dados de retorno. Note-se que, embora no nível de programação, as operações *fork/join* manipulem *threads* (*threads* Anahy), no contexto do núcleo executivo estas mesmas operações permitem identificar as tarefas definidas pelo programa em execução e as relações de dependências entre estas. Assim, as tarefas são implicitamente definidas da seguinte forma:

- No momento em que um *fork* é lançado, duas novas tarefas são criadas. A primeira, é definida no contexto da nova *thread* criada e possui como dados de entrada os argumentos passados para a função definida para ser executada pela *thread*. A segunda, é criada no contexto da *thread* original, tendo como entrada a própria memória local a *thread* atualizada até o momento em que o *fork* foi realizado e o identificador da nova *thread* criada.
- No momento em que um *join* é realizado, uma nova tarefa é criada: a *thread* que executa uma operação de *join* termina a execução de uma tarefa, criando uma nova a partir da instrução que segue (considerando a ordem lexicográfica das instruções) a chamada ao *join*. Esta nova tarefa possui como entrada a memória local a *thread* atualizada até o momento da invocação ao operador *join* e os dados retornados pela *thread* sincronizada.

Considerando a forma como as operações *fork/join* são suportadas pelo ambiente Anahy, os acessos à memória compartilhada são realizados implicitamente. Também se salienta que não existe outra forma de passagem de dados entre tarefas que não seja através dos parâmetros passados para *threads* e de seus eventuais dados de retorno.

4. *Framework*

Em vista das questões a respeito do alinhamento de seqüências traçadas nas seções anteriores, em especial tratando-se da diversidade de variantes de algoritmos de alinhamento e da necessidade de introdução de concorrência, a idéia de uma estrutura comum para a análise de seqüências torna-se desejável. A estrutura proposta neste trabalho é um *framework* [10] sobre o qual possam ser implementados algoritmos de alinhamento de seqüências com suporte para execução concorrente destes algoritmos.

Das características de um *framework*, destaca-se a inversão de controle [11]. Enquanto que na programação tradicional o programador desenvolve o código especificando o fluxo de controle a ser seguido durante a execução, o uso de uma estrutura de *framework* possibilita o desenvolvimento de componentes a serem empregados pela execução de um fluxo já definido.

4.1 Composição do *framework*

O *framework* apresentado neste trabalho foi implementado em C++. Suas principais classes são descritas brevemente na seqüência e apresentadas no diagrama de classes da Figura 2.

A classe abstrata `CDynamicProgramming` implementa métodos genéricos para os cálculos realizados e define a estrutura utilizada para introduzir o método de programação dinâmica. Embora a implementação destes métodos seja de responsabilidade do programador, a sua invocação é realizada pelo *framework*. As classes `CSmithWaterman` e `CNeedlemanWunsch` implementam, respectivamente, os algoritmos de alinhamento local e global, sendo ambas especializações de `CDynamicProgramming`.

A diversidade de formatos de armazenamento de seqüências também foi considerada na especificação do *framework*. `CSequence` e `CSequenceParser` oferecem funcionalidades para introdução de recursos para manipulação de seqüências em memória e em disco. O atual suporte é oferecido para o formato FASTA [16]. O armazenamento dos alinhamentos obtidos é realizado pela classe `CAlignment`, que conta com métodos para obtenção da pontuação do alinhamento resultante e outras informações estatísticas, como as porcentagens de *gaps*, de nucleotídeos idênticos e de similaridade entre as seqüências.

`CMTSmithWaterman` e `CMTNeedlemanWunsch` contêm as implementações concorrentes dos algoritmos de alinhamento local e global Estas classes utilizam serviços especificados a partir da classe abstrata `CMatrixBlock`, que define a interface de serviço ao suporte à concorrência, seguindo o modelo de concorrência adotado pelo *framework*. De forma pontual, esta classe define dois métodos abstratos. O primeiro permite a especificação do cálculo a ser realizado. O segundo é utilizado para viabilizar a sincronização entre os cálculos realizados.

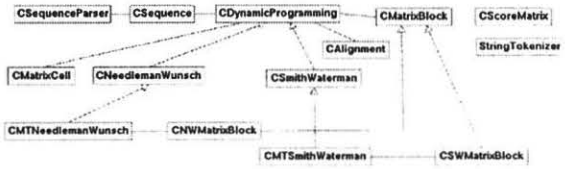

Figura 2. Diagrama de classes do framework.

4.2 Concorrência no *framework*

A concorrência foi introduzida no *framework* para permitir a diminuição do tempo de resposta dos algoritmos implementados e facilitar o desenvolvimento de algoritmos concorrentes de alinhamento de seqüências. O modelo de concorrência adotado segue o padrão *create/join* e é definido pela classe abstrata `CMatrixBlock`. Neste modelo de concorrência, uma tarefa é criada para realizar uma computação, tendo como entrada um conjunto de dados e produzindo, ao final, novos dados como resultado. As relações de ordem entre as tarefas, produção e consumo de dados, permitem realizar o controle da troca de dados entre as tarefas (controle de precedência de execução). O *framework* deve, portanto, disponibilizar funcionalidades para permitir a especificação tanto do cálculo a ser realizado como das dependências de dados existentes entre estes.

Desta forma, as relações de dependências entre os cálculos na programação dinâmica foram consideradas sob diferentes óticas. Dadas as dependências de dados entre os cálculos, a matriz de similaridades pode ser preenchida (*i*) linha a linha, (*ii*) coluna a coluna ou, ainda, (*iii*) pelas antidiagonais ([2][14]). 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 devido ao grande número de sincronizações. A terceira estratégia, cálculo da matriz por suas antidiagonais, não apresenta este problema, pois uma antidiagonal depende somente dos elementos das outras antidiagonais previamente calculadas. Mesmo

apresentando um grau de concorrência mais elevado, esta estratégia apresenta problemas para uma implementação paralela eficiente. Um dos problemas é que o tamanho das antidiagonais varia durante o preenchimento da matriz, causando, assim, uma não uniformidade no número de tarefas durante a execução do programa. Outro problema desta estratégia está na definição da granulosidade. Se para cada célula da matriz for criada uma tarefa, o número total de tarefas criadas será muito elevado, proporcional ao produto do tamanho das seqüências. Em dados biológicos reais, o número de tarefas pode vir a ser grande o suficiente para que o sobrecusto (*overhead*) de sincronização entre estas sobreponha o potencial ganho de uma execução paralela.

Uma solução ao problema de granulosidade é dividir a matriz de similaridades em blocos retangulares de elementos, como mostrado na Figura 3. Neste exemplo, um programa em execução inicia pelo cálculo do bloco 1, para, em seguida, calcular os blocos 2 e 5, visto que as dependências foram resolvidas pelo cálculo do bloco 1. Toda a matriz de similaridades é calculada desta forma [14]. Se cada bloco tem q linhas e r colunas, então a computação de um dado bloco requer o segmento de linha imediatamente acima do bloco, o segmento de coluna imediatamente à esquerda e o elemento acima à esquerda, um total de $q + r + 1$ elementos, mais dois vetores de q e r elementos, contendo os maiores elementos já encontrados nas respectivas linhas e colunas. Por exemplo, se cada bloco tem 4 linhas e 4 colunas, então cada bloco irá calcular 16 valores após ter recebido 17 valores de entrada.

No *framework* construído, são utilizadas as estratégias de divisão da matriz de similaridades em blocos e a execução paralela do cálculo dos elementos de cada bloco, respeitando a ordem especificada pelas dependências de dados. Na implementação realizada, cada instância de `CMatrixBlock` representa um fluxo de execução na aplicação, definindo, também, a granulosidade da mesma. A idéia geral é fazer com que cada bloco da matriz de similaridades seja calculado de forma concorrente, sempre respeitando as dependências existentes no cálculo dos valores da matriz.

A solução concorrente proposta faz uso de *threads* para realizar a computação de um bloco. *Threads* são, assim, criadas dinamicamente, sendo ativadas no momento em que suas 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. Uma primeira vantagem desta estratégia é de possibilitar a descrição da concorrência natural da aplicação, não considerando os recursos físicos disponíveis. Outra vantagem é possibilitar a distribuição das *threads* entre os recursos de processamento de forma dinâmica, evitando um indesejável desbalanceamento de carga advindo do número variável de *threads* ativas durante a execução.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figura 3. Divisão em blocos da matriz.

A Figura 4 apresenta o esquema de execução das *threads* adotado para a solução concorrente da aplicação. Nesta figura são apresentadas seis *threads*, responsáveis pelo cálculo de seis dos blocos da matriz apresentada na Figura 3. As arestas direcionadas indicam as dependências de dados entre as *threads*, restringindo o paralelismo na execução. Estas arestas encontram-se anotadas com os dados transferidos de uma *thread* à outra. A execução inicia pelo cálculo do bloco 1 e, quando este termina o cálculo de seus elementos, inicia-se o cálculo dos blocos 2 e 5, dado que as dependências encontram-se satisfeitas. O cálculo do bloco 6 tem sua execução iniciada somente quando os blocos 2 e 5 estiverem com todos os seus elementos calculados. A execução prossegue com as demais *threads* não representadas na figura.

4.3 Algoritmos implementados

Com o propósito de analisar a estrutura oferecida pelo *framework*, foram implementados, utilizando as classes disponibilizadas, os algoritmos Needleman-Wunsch [17] (alinhamento global) e Smith-Waterman [20] (alinhamento local), ambos baseados no método de programação dinâmica, tendo sido implementados de forma seqüencial e concorrente. Nas versões concorrentes, a matriz de similaridades foi dividida em blocos parametrizáveis de q linhas e r colunas, permitindo avaliar o impacto do tamanho do grão. Nesta estratégia cada bloco da matriz é calculado por uma *thread*. Duas versões para o suporte à execução concorrente foram implementadas (extensões à `CMatrixBlock`): uma utilizando *threads* POSIX e outra utilizando Anahy.

5. Análise da instanciação do *framework*

Com o intuito de analisar o comportamento de execução do *framework*, em especial do seu suporte à concorrência, foi realizada uma série de experimentos sobre uma arquitetura bi-processada (2 XEON 2.8GHz, 1 GB RAM, Gobo Linux, kernel 2.4.21). O algoritmo utilizado nos testes apresentados é o algoritmo Smith-Waterman para realizar o alinhamento local entre duas seqüências. Em um primeiro momento foi realizada uma validação das implementações, comparando os alinhamentos obtidos com os fornecidos por outros softwares (Emboss [18] e JAligner). Nesta análise, os

alinhamentos obtidos foram idênticos, aferindo confiabilidade aos resultados. A segunda bateria de experimentos buscou dados de desempenho do suporte à concorrência oferecido pelo *framework*, em especial de sua implementação com Anahy. Levando em conta a arquitetura utilizada, mais precisamente a quantidade de memória disponível, foi possível realizar o alinhamento de 3 genes do *Mycoplasma genitalium* com 3 genes do *Mycoplasma pneumoniae*, totalizando uma matriz de similaridades com 16.027.506 elementos.

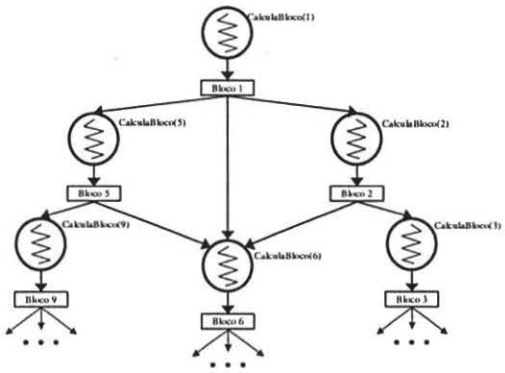

Figura 4. Esquema de execução concorrente.

Os resultados de desempenho apresentados referem-se ao tempo de execução da aplicação, nas suas diferentes versões, em segundos. Cada resultado apresentado consiste na média de 100 execuções com o desvio padrão inferior a 10%.

O primeiro conjunto de resultados a ser discutido encontra-se na Tabela 1, que apresenta os tempos obtidos pela execução de diferentes versões da aplicação na arquitetura utilizada. Salienta-se que, os resultados apresentados nesta tabela, referem-se a execuções não paralelas, ou seja, exploram apenas um dos processadores da arquitetura anteriormente descrita. Estes resultados permitem analisar alguns dos custos associados à execução destas diferentes versões.

Em um primeiro momento foi coletado o tempo de execução S da versão seqüencial (cálculo por elementos) da aplicação, o qual corresponde à implementação direta do algoritmo de programação dinâmica. Neste caso, o algoritmo inicia percorrendo toda a matriz de similaridades, da esquerda para a direita e de cima para baixo, realizando o cálculo elemento por elemento, varrendo a matriz linha a linha. Como define o método de programação dinâmica, cada elemento é calculado considerando os valores de seus vizinhos (norte, oeste e noroeste) e os valores máximos encontrados em cada linha e coluna. O tempo obtido para esta versão seqüencial foi de S = 17,03 s.

Tabela 1. Tempos para diferentes versões da aplicação em execução seqüencial.

Versões da aplicação	Medida	Tempo (s)
Seqüencial - cálculo por elementos	S	17,03
Concorrente Anahy	T_1	6,06
Seqüencial - cálculo por blocos	T_s	6,02

O segundo experimento realizado refere-se à análise dos tempos obtidos pela execução da implementação concorrente sobre recursos de processamento não paralelos. Esta medida encontra-se referenciada na Tabela 1 como T_1, tendo sido aferido como T_1 = 6,06 s. Salienta-se que, diferentemente da versão seqüencial, a versão concorrente realiza as operações por blocos de elementos de forma a aumentar a granulosidade das tarefas de cálculo (no caso dos dados apresentados nesta tabela, os blocos possuem tamanho de 15x15 elementos). Somente foi possível coletar o tempo T_1 por ter sido utilizado como mecanismo de suporte à concorrência a ferramenta Anahy.

No caso da medida T_1, a máquina virtual de Anahy foi configurada com apenas 1 processador virtual. Desta forma, as tarefas concorrentes definidas pelo programa em execução são executadas em um fluxo único. Assim, embora seja agregado ao tempo de execução todo custo adicional envolvido na gestão da concorrência pelo ambiente Anahy, não é explorado o ganho que pode ser obtido na execução paralela do algoritmo. Portanto, a diferença entre os tempos S e T_1 representa o *overhead* da descrição e da gestão da concorrência.

Chama atenção que a versão concorrente sendo executada de forma não paralela possui um tempo de execução inferior à versão seqüencial. Este fato advém das diferentes formas adotadas pelos algoritmos para realização dos cálculos. A versão seqüencial realiza o cálculo elemento por elemento, pesquisando cada linha e coluna para encontrar o maior elemento a cada elemento calculado. A versão concorrente realiza o cálculo por blocos, utilizando como entrada um vetor com os máximos valores de linha e coluna já encontrados, evitando um processo de busca mais demorado.

Uma nova versão seqüencial foi composta para igualmente realizar o cálculo por blocos, recebendo como entrada um vetor com os máximos já encontrados em cada linha e coluna. O tempo obtido para esta nova versão (para o mesmo tamanho de bloco) foi de T_s = 6,02 s. Portanto coerente com a expectativa de que existe a introdução de *overhead* para obter uma versão concorrente de um algoritmo seqüencial. A conclusão da análise dos tempos apresentados indica que o algoritmo concorrente pode ser suportado sem a introdução de sobrecusto significativo na execução. Além disso, Anahy

não introduz um grande custo de gerência da execução das atividades concorrentes.

Na seqüência foram tomados os tempos com execuções paralelas do algoritmo concorrente, sobre os dois processadores da arquitetura SMP disponível, considerando diferentes tamanhos de blocos. Para tanto, o *framework* foi estendido oferecendo dois suportes à concorrência: *threads* POSIX e Anahy, possibilitando a obtenção das versões concorrentes implementadas, pelas classes `CMTPosixSmithWaterman` e `CMTAnahySmithWaterman`, respectivamente. Destaca-se que estas versões possuem estrutura e comportamento idênticos. Os tempos medidos para as execuções paralelas encontram-se na Tabela 2. Nesta tabela, a primeira coluna identifica o número de blocos utilizado no experimento, refletindo o número de tarefas concorrentes (*threads*) e a granulosidade destas. A segunda coluna relata os tempos obtidos pela execução do programa com suporte de *threads* POSIX. As colunas restantes referem-se à execução do programa com suporte de Anahy tendo sua máquina virtual configurada com diferentes números de processadores virtuais.

Na versão POSIX, cada uma das atividades concorrentes definida pelo programa é transformada em uma unidade de cálculo paralelo, ou seja, em uma *thread* POSIX. Estas *threads* são, portanto, criadas e destruídas à medida que o programa evolui, existindo, então, a possibilidade de diversas *threads* estarem ativas em um determinado instante de tempo. No caso do estudo realizado, até 15 *threads* ativas quando da realização do cálculo dividindo-se a matriz em 15x15 blocos (durante o cálculo da maior antidiagonal da matriz). Os tempos obtidos nesta versão indicam que um certo grau de paralelismo é necessário para tirar proveito do hardware (bi-processado) disponível. Porém, o número de atividades paralelas definidas pelo programa deve ser limitado à capacidade efetiva da máquina.

Um melhor mapeamento da concorrência da aplicação no paralelismo da arquitetura (*cf.* Tabela 2), foi obtido com o uso de Anahy como suporte à execução. Neste caso, o número de atividades em execução simultânea é limitado pela capacidade de processamento da máquina virtual, a qual é determinada pelo número de processadores virtuais empregados. Assim, embora possam ocorrer situações onde diversas *threads* definidas pela aplicação estejam aptas a executar, o número de atividades que ocorrem efetivamente em paralelo está limitado ao número de PVs disponíveis, reduzindo custos advindos da gestão da concorrência definida pela aplicação. Os tempos obtidos reforçam a idéia de que Anahy introduz pouco sobrecusto frente a uma execução seqüencial da aplicação, permitindo ainda, embora modesto, algum ganho de desempenho. A expectativa é que Anahy ofereça um maior conforto de programação, uma vez que, face ao uso de *threads* POSIX, permite uma maior liberdade para o programador definir a granulosidade de seu programa em função da concorrência natural da aplicação.

Tabela 2. Tempos para execuções de versões concorrentes da aplicação.

Número de blocos	POSIX	Anahy					
		1 PV	2 PVs	5 PVs	10 PVs	15 PVs	20 PVs
2x2	14,97	7,19	7,36	7,04	7,02	7,04	7,04
3x3	12,74	6,45	6,64	6,04	6,04	6,04	6,04
4x4	11,83	6,18	6,31	5,73	5,71	5,72	5,72
5x5	11,51	6,07	6,22	5,74	5,60	5,60	5,61
10x10	11,18	6,04	6,49	6,02	5,59	5,53	5,54
15x15	11,16	6,06	6,43	6,16	5,69	5,64	5,59

Os resultados alcançados até o momento mostram que a estrutura adotada para descrição da concorrência de algoritmos baseados em programação dinâmica para alinhamento de seqüências é adequada, permitindo a utilização de diferentes ferramentas de programação para suportar sua execução. A implementação realizada com Anahy tem um interesse especial, pois comprova resultados de desempenho de trabalhos já coletados em outras frentes de trabalho da equipe [9] e valida o esforço de desenvolvimento deste ambiente.

6. Conclusão

No presente trabalho tratou-se da crescente disponibilização de seqüências de DNA em bases de seqüências públicas e da necessidade de tratá-las precisa e rápidamente. A ferramenta implementada e apresentada neste artigo oferece facilidades para introdução de algoritmos de alinhamento de seqüências em uma estrutura de software pré-concebida. Esta ferramenta é disponibilizada sob a forma de um *framework*, representando o comportamento de um conjunto de algoritmos de alinhamento de seqüências. A avaliação deste *framework* se deu pela sua extensão, implementando algoritmos de alinhamento que empregam o método de programação dinâmica.

O *framework* desenvolvido se caracteriza por oferecer recursos para o processamento concorrente dos algoritmos implementados. Tais recursos foram introduzidos considerando a alta carga computacional gerada pela execução dos algoritmos de alinhamento. O modelo de concorrência adotado foi retirado do modelo de programação de Anahy [6], tendo sido implementado dois suportes à execução concorrente em arquitetura SMP. Análises de desempenho realizadas encorajam a continuidade dos trabalhos nesta linha.

Neste trabalho também foi possível analisar tanto o modelo de programação proposto por Anahy [6] como seu núcleo executivo [9]. Os resultados obtidos compro-

vam outros anteriormente alcançados ([15], [3]) em demais trabalhos da equipe.

Dentre as possibilidades para extensões deste trabalho, destaca-se a implementação de novos algoritmos de alinhamento e o tratamento de questões de limite de memória. Em médio prazo, trabalhos em desenvolvimento oferecerão suporte à execução do *framework* em aglomerados de computadores. Este suporte está sendo buscado tanto através do desenvolvimento de classes próprias ao *framework*, como no desenvolvimento de Anahy, que objetiva uma versão operacional para aglomerados ([8]). Desta forma, a comunidade científica poderá contar com recursos para exploração de processamento de alto desempenho para pesquisa em seqüências biológicas.

Referências

[1] Almeida Jr, N.G., Alves, C. E. R., Caceres, N. E. e Song, S. W. "Comparison of Genomes using High-Performance Parallel Computing", *Proc. of 15th Symp. on Comp. Archit. and High Perf. Comp.*, São Paulo, 2003.

[2] Alves, C. E. R., Caceres, E. N., Dehne, F. e Song, S. W. "A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison", *Proc. of Inter. Conf. on Comput. Science and its Applic.*, Montreal, May 18-21, Lecture Notes in Computer Science, 2003, 2668:249-258.

[3] Benitez, E. D., Dall'Agnol, E. C., Villa Real, L. C., Cardozo Jr, M. A. e Cavalheiro, G. G. H. "Avaliação de desempenho de Anahy em aplicações paralelas", *Anais WPerformance*, Salvador, 2004. (A ser publicado)

[4] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H e Zhou, Y. "Cilk: an efficient multithreaded runtime system", *ACM SIG-PLAN Notices*, 30(8):207-216, Agosto, 1995.

[5] Cavalheiro, G. G. H. "Introdução à Programação Paralela e Distribuída", *Anais da 1ª. Escola Regional de Alto Desempenho*, Gramado, janeiro, 2001, pp. 35-74.

[6] Cavalheiro, G. G. H., Dall´Agnol, E. C. e Villa Real, L. C. "Uma biblioteca de Processos Leves para a Implementação de Aplicações Altamente Paralelas", *Anais do IV Workshop de Sistemas Computacionais de Alto Desempenho*, São Paulo, 2003.

[7] Cavalheiro, G. G. H., Denneulin, Y. e J.-L. Roch. "A General Modular Specification for Distributed Schedulers", *Lecture Notes in Computer Science*, 1470:373-??, 1998.

[8] Dall'Agnol, E. C. e Cavalheiro, G. G. H. "Biblioteca de comunicação com mensagens ativas", *Anais da 4ª. Escola Regional de Alto Desempenho*, Pelotas, 2003.

[9] Dall'Agnol, E. C., Villa Real, L. C., Benitez, E. D. e Cavalheiro, G. G. H. "Portabilidade na programação para o processamento de alto desempenho", *Anais do IV Workshop de Sistemas Computacionais de Alto Desempenho*, São Paulo, 2003.

[10] Fayad, M. A. et al. "*Building Application Frameworks Object Oriented Foundations of Framework Design*", John Wiley & Sons: New York, 1999. 688 p.

[11] Fayad, M. e Schmidt, D. C. "Object Oriented Application Frameworks", *Comm. of the ACM*. 10(40), Outubro, 1997.

[12] Gallilée, F., Roch, J.-L., Cavalheiro, G. G. H. e Doreille, M. "Athapascan-1: On-Line Building Data Flow Graph in a Parallel Language", *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (PACT98), IEEE Computer Society Press, Paris, outubro, 1998, pp. 88-95.

[13] Gusfield, D. "*Algorithms on strings, trees and sequences: computer science and molecular biology*", Cambridge University: New York, 1997, 534p.

[14] Martins, W. S., Del Cuvillo, J. B., Useche, F. J., Theobald, K. B. e Gao, G. R. "A multithreaded parallel implementation of a dynamic programming algorithm for Sequence Comparison", *Proc. of Pacific Symposium on Biocomputing*, 6:311-322, 2001.

[15] Moschetta, E., Osório, F. S. e Cavalheiro, G. G. H. "Reconhecimento de imagens em aplicações críticas", *Anais do Workshop de Sistemas Computacionais de Alto Desempenho*, Vitória, 2002.

[16] Mount, D. W. "*Bioinformatics: sequence and genome analysis*", Cold Spring Harbor Laboratory, New York, 2001, 564 p.

[17] Needleman S. B. e Wunsch C. D. "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *Journal of Molecular Biology*. Março, 48(3):443-53, 1970.

[18] Rice, P., Longden, I. e Bleasby, A. "EMBOSS: The European Molecular Biology Open software Suite". *Trends in Genetics*. Junho, 16(6):276-277, 2000.

[19] Schmidt, B., Schröder, H. e Schimmler, M. "Massively parallel solutions for molecular sequence Analysis", *Proc. of the International Parallel and Distributed Processing Symposium*, IEEE, 2002.

[20] Smith, T. F. e Waterman, M. "Identification of common molecular subsequences", *Journal of Molecular Biology*, 147:195-197, 1981.

[21] Yap, T. K., Frieder, O. e Martino, R. L. "Parallel computation in biological sequence analysis", *IEEE Trans. on Par. and Distrib. Syst.*, 9(3):283-294, 1998.