Implementação e Avaliação de Desempenho do LCS Paralelo em *Cluster Multicore*

Alexandre M. Lauredo, Joyce S. de Mesquita, Leandro Santiago, Maria Clicia S. de Castro, Alexandre C. Sena, Leandro A. J. Marzulo

[1]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brazil

{alauredo,jmesquita,lsantiago,clicia,asena,leandro}@ime.uerj.br

Abstract. *Finding the longest common subsequence (LCS) is an important technique in DNA sequence alignment. Through dynamic programming it is possible to find the exact solution to the LCS, with space and time complexity of O($m \times n$), being m e n the sequence sizes. Parallel algorithms are essential, since large sequences require too much time and memory to be processed sequentially. Thus, the aim of this work is to implement and evaluate different parallel solutions for distributed memory machines, so that the amount of memory is equally divided among the various processing nodes.*

Resumo. *Encontrar a maior subsequência comum (LCS) entre duas sequências é uma técnica muito utilizada para o alinhamento de sequências de DNA. Através da programação dinâmica se consegue a solução exata para o LCS, com complexidade de tempo e espaço O($m \times n$), onde m e n são os tamanhos das sequências. Algoritmos paralelos são fundamentais, tanto pelo tempo de processamento, quanto pela quantidade de memória necessária para processar sequências grandes. Logo, o objetivo deste trabalho é implementar e avaliar quatro versões paralelas do LCS para máquinas com memória distribuída, otimizando o paralelismo entre nós e também dentro de cada nó e buscando uso homogêneo de memória nos nós de processamento.*

1. Introdução

O alinhamento de sequências é um problema extremamente importante para investigar a similaridade entre diferentes espécies, uma vez que uma grande similaridade entre sequências genéticas frequentemente resulta em grandes semelhanças na estrutura molecular e funcional. Nesse contexto, encontrar a maior subsequência comum (*LCS*) entre duas sequências é um método muito utilizado para alinhamento de sequências.

Para encontrar a solução exata para o problema *LCS*, a técnica de programação dinâmica é provavelmente a mais empregada. Ela utiliza uma matriz de pontos que, após ser preenchida, terá no elemento inferior da direita o tamanho da *LCS*. A maior subsequência comum pode ser encontrada percorrendo a matriz de pontos na direção contrária. Assumindo que m e n são os tamanhos das duas sequências, a complexidade de tempo e espaço da técnica de programação dinâmica é O($m \times n$) [Axelson-Fisk 2010].

Em função do crescimento dos bancos de dados de Genomas, como por exemplo o GenBank que contém atualmente aproximadamente $173.353.076$ sequências

[D. A. Benson and Sayers 2013] e principalmente dos tamanhos das cadeias de DNA, por exemplo, o genoma de uma simples drosófila (Drosophila melanogaster) é composto de $122.653.977$ pares de bases, enquanto o genoma humano é composto de $3,3 \times 10^9$ pares de bases [D. A. Benson and Sayers 2013]. Assim sendo, os algoritmos paralelos se tornaram fundamentais para tratar problemas complexos (comparar grandes cadeias de DNA).

A técnica de percorrer a matriz pela diagonal, conhecida como *wave-front* [Yap et al. 1998], é a mais usada na paralelização do *LCS* com programação dinâmica, pois ela aproveita a independência dos elementos em uma mesma diagonal. Nesse algoritmo, a complexidade para manter toda matriz em memória é O($m \times n$).

Em razão da grande quantidade de memória necessária para comparar cadeias grandes, o uso de uma arquitetura com memória distribuída é mais adequado por permitir a divisão da matriz entre os diversos nós do sistema. Por exemplo, apenas a matriz usada pelo algoritmo *LCS* para descobrir a sequência mais longa entre duas cadeias contendo 100.000 nucleotídeos, gastaria aproximadamente 37 GB de memória, considerando que cada elemento da matriz tenha um tamanho de 4 *bytes*, inviabilizando sua execução em apenas uma máquina *multicore*.

O objetivo deste trabalho é avaliar o desempenho do algoritmo paralelo *LCS*, baseado na técnica de *wavefront*, em um *cluster multicore*. Para isso, foram implementadas quatro versões paralelas usando as bibliotecas OpenMPI [openMPI 2014], Pthreads [Butenhof 1997], OpenMP [Dagum and Menon 1998] and TBB [Reinders 2007], com o objetivo de explorar tanto o paralelismo dentro dos nós, como também entre os nós, eficientemente. Outro objetivo, é mostrar que é possível executar o algoritmo *LCS* para cadeias grandes dividindo a quantidade de memória necessária para armazenar a solução eficientemente entre os recursos computacionais.

O restante deste trabalho está organizado da seguinte forma: (*i*) na Seção 2 são apresentadas a definição do LCS e suas aplicações; (*ii*) na Seção 3 são discutidas as soluções de paralelização adotadas; (*ii*) na Seção 4 é apresentada a avaliação experimental das soluções paralelas; (*iv*) na Seção 5 são discutidos os trabalhos relacionados; (*v*) e, por último, na Seção 6 são apresentadas as conclusões e trabalhos futuros.

2. *Longest Common Subsequence* (*LCS*)

Dada uma sequência de símbolos $S = \langle a_0, a_1, ..., a_n \rangle$, uma subsequência S' de S é obtida removendo zero ou mais símbolos de S. Por exemplo, dado $K = \langle a, b, c, d, e \rangle$, $K' = \langle b, d, e \rangle$ é uma subsequência de K. Uma subsequência comum máxima (*longest common subsequence*, ou *LCS*) de duas sequências X e Y é uma subsequência de ambos X e Y com maior comprimento.

O comprimento de uma subsequência comum máxima, a distância de Levenshtein, é usado para medir a diferença entre duas sequências. A *LCS* tem aplicações em diversas áreas, como na Biologia, Genética, Medicina e Linguística. Em Linguística, por exemplo, a *LCS* dá informações sobre similaridades entre palavras de idiomas distintos, já que a distância de Levenshtein entre palavras é o número mínimo de inserções, exclusões ou substituições de caracteres necessário para transformar uma palavra em outra. No ramo da Genética, a *LCS* é usada para encontrar similaridades entre proteínas ou moléculas de DNA, que são reprentados como cadeias de nucleotídeos.

O comprimento $c[m, n]$ de uma *LCS* de duas sequências $A = \langle a_0, a_1, ..., a_m \rangle$ e $B = \langle b_0, b_1, ..., b_n \rangle$ pode ser definido, recursivamente, da seguinte forma:

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } a_i = b_j, \\ max\{c[i-1, j], c[i, j-1]\} & \text{caso contrário.} \end{cases}$$

A partir dessa definição, um algoritmo de programação dinâmica pode ser derivado diretamente. Em [Wagner and Fischer 1974] uma implementação com programação dinâmica foi apresentada, com complexidade de tempo e espaço de $O(m \times n)$. Nessa versão, uma matriz de pontuação é preenchida, observando as relações descritas na definição recursiva, ou seja, cada elemento da matriz depende de seu vizinho de cima, da esquerda e do elemento na diagonal superior esquerda.

3. *LCS* Paralelo para *Clusters Multicore*

Nesta seção são descritas as quatro diferentes soluções paralelas do *LCS* implementadas para execução em *clusters multicore*. O paralelismo da aplicação é baseado na técnica de *wavefront*, que percorre a matriz pela diagonal [Yap et al. 1998]. O foco principal das soluções é promover uma divisão homogênea da matriz de pontuação entre os nós do *cluster*, com o objetivo de permitir a execução de alinhamento de sequências grandes. A primeira solução usa apenas troca de mensagens com MPI. As demais adotam uma abordagem híbrida, com uso de MPI para comunicação entre os nós e memória compartilhada dentro de cada nó, com uso de TBB, OpenMP ou Pthreads.

3.1. Solução com troca de mensagens - MPI

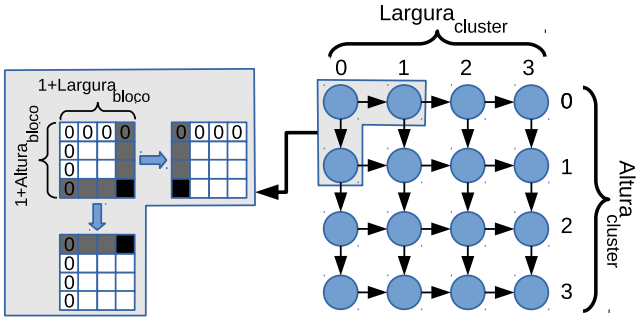


Figura 1. Padrão de comunicação do MPI-LCS.

A solução MPI do *LCS* é a base para todas as versões, inclusive as híbridas, apresentadas na Seção 3.2. Ela é fundamentada no fluxo de dados básico da aplicação. O padrão de dependências descrito na Seção 2 é mantido quando a matriz de pontuação é dividida em blocos. Dessa forma, o *cluster* pode ser visto como uma matriz virtual de nós, onde cada nó é responsável pela computação de um bloco da matriz de pontuação. A Figura 1 mostra a comunicação entre os nós do *cluster* no MPI-LCS. Cada nó processa a sua parte da matriz de pontuação e envia a última coluna para o vizinho da direita e a última linha para o vizinho de baixo. Note que a dependência com o elemento da diagonal é satisfeita transitivamente.

Algoritimo 1 Algoritmo para encontrar o tamanho de uma subsequência comum máxima entre duas sequências de forma paralela (com troca de mensagens)

Entrada: $Arquivo_A$ e $Arquivo_B$: Nomes dos arquivos com as sequências
tam_A e tam_B: Tamanho total das sequências
$Altura_{cluster}$ e $Largura_{cluster}$: as dimensões do *cluster* computacional
$rank$: identificador único processo MPI

Saída: Tamanho de uma subsequência comum máxima entre duas sequências

início

$i_{cluster} = rank/Alturacluster$
$j_{cluster} = rank\%Largura_{cluster}$

$Largura_{bloco} = tam_A/Largura_{cluster}$
$Altura_{bloco} = tam_B/Altura_{cluster}$

$Seq_A = l\hat{e}\,(tam_A, Largura_{bloco}, j_{cluster})$
$Seq_B = l\hat{e}\,(tam_B, Alturabloco, i_{cluster})$

$MP = Aloca\,(Altura_{bloco} + 1, Largura_{bloco} + 1)$

$MP[0][0..Largura_{bloco}] = 0$
se $(i_{cluster} > 0)$ **então**
$MP[0][0..Largura_{bloco}] = RecebaDe\,(i_{cluster} - 1, j_{cluster})$

fim

$MP[0..Altura_{bloco}][0] = 0$
se $(j_{cluster} > 0)$ **então**
$MP[0..Altura_{bloco}][0] = RecebaDe\,(i_{cluster}, j_{cluster} - 1)$

fim

$lcs = computaLCS\,(MP, Seq_A, Seq_B, Largura_{bloco}, Altura_{bloco})$

se $(i_{cluster} < Altura_{cluster})$ **então**
$EnviaPara\,(MP[Altura_{bloco}][0..Largura_{bloco}], i_{cluster} - 1, j_{cluster})$

fim

se $(j_{cluster} < Largura_{cluster})$ **então**
$EnviaPara\,(MP[0..Altura_{bloco}][Largura_{bloco}], i_{cluster}, j_{cluster} - 1)$

fim

$Libera\,(MP)$

se $(j_{cluster} == Largura_{cluster} - 1)$ *e* $(i_{cluster} == Altura_{cluster} - 1)$ **então**
$Imprime\,(lcs)$

fim

fim

O Algoritmo 1 descreve a execução realizada por cada processo MPI para o processamento paralelo do LCS. A entrada de cada processo é composta pelos nomes dos arquivos com as sequências e seus tamanhos, a altura e largura do *cluster* e o seu identificador único no MPI ($rank$). O processo calcula suas coordenadas dentro do *cluster*, com base no seu $rank$ e nas dimensões do *cluster* e faz a divisão da matriz de pontuação (MP), com base no tamanho das sequências e nas dimensões do *cluster*. Por simplicidade, no algoritmo apresentado, foi considerado que os tamanhos das sequências são divisíveis pelas dimensões do *cluster*. Na implementação real, se há resto nessa divisão, o bloco dos processos na última linha e última coluna do *cluster* realizam o trabalho extra. O processo, então, lê nos arquivos a porção das sequências pelas quais é responsável e aloca a matriz de pontuação.

Antes de iniciar a computação, cada processo precisa receber de seus vizinhos superior e esquerdo, respectivamente, a última linha e última coluna de suas matrizes de

pontuação. Elas são armazenadas, respectivamente, na primeira linha e primeira coluna da matriz de pontuação do próprio processo. É importante lembrar que nem todos os nós do *cluster* possuem as duas dependências. O nó $[0, 0]$ não depende de nenhum vizinho para iniciar sua computação. Já os nós da primeira linha e primeira coluna não dependem de seus vizinhos superior e esquerdo, respectivamente. Nesses casos, o recebimento de mensagem é substituído por uma inicialização (com zeros) da primeira linha e/ou coluna da matriz de pontuação.

Após o recebimento das dependências, o processo computa a *LCS* para a sua matriz de pontuação, como descrito na Seção 2 e, então, envia a última coluna da matriz para o seu vizinho da direita e a última linha para o vizinho de baixo, se houver.

3.2. Soluções Híbridas - MPI com Memória Compartilhada

Para explorar o paralelismo, dentro de uma máquina *multicore*, é mais adequado adotar o paradigma de memória compartilhada com execução *multithread*. Nessa abordagem é criado apenas um processo por nó, e a porção da matriz de pontuação do processo é subdividida em blocos a serem computados pelas *threads* instanciadas pela solução. Dessa forma, são evitados os envios de mensagens de linhas e colunas em um mesmo nó, pois cada *thread* pode acessar diretamente essa informação na matriz de pontuação compartilhada. Como visto na Seção 3.1, o MPI é usado para a comunicação entre nós em todas a versões híbridas apresentadas neste trabalho. Para a paralelização da execução dentro de um nó são usadas três alternativas: TBB, Pthreads e OpenMP.

3.2.1. MPI com TBB

Para a versão híbrida com TBB foi usado o conceito de *flow-graph* [TBB FlowGraph 2014], que permite descrever a aplicação como um grafo de fluxo de dados, onde os nós são as operações e as arestas representam as dependências entre elas. O código deve indicar a criação dos nós e arestas, associando a cada nó a tarefa a ser realizada. O grafo é gerado em tempo de execução conforme a descrição feita pelo usuário. Nós sem dependência devem receber uma mensagem de inicialização para disparar a execução no grafo.

A matriz de pontuação de um nó do *cluster* é dividada em blocos, com tamanho escolhido pelo usuário por passagem de parâmetro de linha de comando. Cada bloco é computado por uma tarefa do grafo de fluxo de dados. O TBB cria um número pequeno de *threads* trabalhadoras (no caso deste trabalho, igual ao número de *cores*) e possui um mecanismo de escalonamento com roubo de tarefas para balancear a execução das tarefas nas *threads* trabalhadoras. Assim, nessa implementação foi necessário descrever o mesmo padrão de dependências da versão MPI puro para execução em paralelo dentro de um nó do *cluster*.

3.2.2. MPI com Pthreads

Para a versão Pthreads foi adotada uma abordagem semelhante a do TBB, mas de forma manual. A biblioteca Pthreads é de baixo nível, contendo apenas um conjunto de funções para criação, sincronização e gerência de *threads*. Assim como no TBB, a porção da

matriz de pontuação alocada a um nó do *cluster* foi dividida em blocos. Entretanto, como em Pthreads não há um mecanismo de escalonamento de tarefas, é criada uma *thread* em cada bloco.

Para implementar a sincronização fina orientada por dependências de dados, como no TBB e MPI, foi criada uma matriz de semáforos de contagem, onde cada semáforo é responsável por permitir a execução de uma *thread*. O semáforo do bloco $[0, 0]$ é iniciado com 2, os semáforos dos blocos $[0, 1..n-1]$ e $[1..n-1, 0]$ são iniciados com 1 e os semáforos dos blocos $[1..n-1, 1..n-1]$ são iniciados com 0. O valor do semáforo de cada bloco indica quantas liberações de execução a *thread* que computa cada bloco recebeu, sendo que cada *thread* necessita de 2 liberações, uma do vizinho de cima e outra do vizinho da esquerda, quando houver.

A dinâmica de execução é simples: cada *thread* de coordenada (i, j) realiza duas operações `wait(semaforo[i,j])`, para aguardar as duas liberações no seu semáforo. Quando as liberações forem feitas pelos vizinhos, a *thread* computa e sinaliza seus vizinhos de baixo e da direita, com as operações `signal(semaforo[i+1,j])` e `signal(semaforo[i,j+1])`, respectivamente. Note, portanto, que a inicialização com 2 no semáforo do bloco $[0, 0]$ permite que a *thread* correspondente execute sem receber liberações; que a inicialização com 1 nos semáforos dos blocos $[0, 1..n-1]$ e $[1..n-1, 0]$ permite que as *threads* correspondentes executem com apenas uma liberação e que as demais *threads* só executem após duas liberações.

3.2.3. MPI com OpenMP

O OpenMP oferece diretivas simples de paralelização, especialmente para aplicações que seguem o modelo *fork-join*. Embora seja possível adotar a mesma abordagem de paralelização da solução Pthreads no OpenMP, ela eliminaria o sentido de usar essa biblioteca. Desta forma, a solução OpenMP adota um padrão *wavefront* por blocos na paralelização. A matriz de pontuação do nó é dividida em blocos e a matriz de blocos é percorrida com dois laços aninhados: o laço externo percorre as diagonais da matriz de blocos e o interno percorre os elementos de uma diagonal. O laço interno é paralelizado usando a diretiva `parallel for` do OpenMP. Os elementos de um bloco são percorridos de forma tradicional.

A abordagem OpenMP, embora seja de fácil entendimento, adiciona uma sincronização global (barreira) ao final da computação de cada diagonal, o que pode ser um fator limitante na escalabilidade, conforme observado em [Alves et al. 2013].

4. Análise Experimental

Esta seção apresenta uma série de experimentos para avaliar o algoritmo proposto, usando diferentes modelos de programação, nas máquinas *multicore*. O objetivo dos experimentos é avaliar o algoritmo *LCS* baseado na técnica de *wavefront* explorando ao máximo o paralelismo dentro dos nós e também entre os nós. Além disso, o experimento mostra a viabilidade de executar a comparação de cadeias cada vez maiores a medida que o número de nós aumenta.

Todos os experimentos foram realizados em um *cluster* de 40 nós, onde cada

nó é uma máquina com processador Intel®core i5-3330 CPU (3.00GHz) com 8GB de memória DDR3-1333MHz, interconectados por uma rede ethernet de 100Mbs. Um dos nós do *cluster* é também um servidor de arquivos NFS, usado para armazenar os arquivos executáveis e os arquivos de entrada.

4.1. Comparação de Desempenho

Para avaliar o desempenho do *LCS* paralelo, cada uma das quatro versões implementadas, conforme descrito na Seção 3, foi executada para 5 pares de sequências com tamanhos de 40 mil, 80 mil, 150 mil, 200 mil e 230 mil, utilizando, respectivamente, toda a memória disponível em 1, 4, 16, 32 e 40 nós do *Cluster*. A versão MPI puro foi executada sempre com uma quantidade de processos igual a quantidade de nós multiplicada por 4, que é a quantidade de núcleos de cada nó do *cluster*. Por sua vez, as outras versões foram executadas com a quantidade de processos MPI igual ao número de nós e dentro de cada nó foram criadas 4 *threads*. Os resultados podem ser observados na Figura 2. Cada cenário foi executado 10 vezes e a média e desvio padrão calculados. Para sequências pequenas, como o tempo de execução é muito baixo (para matrizes de 40x40 mil, o tempo sequencial é de 6,8s), e diminui ainda mais com a paralelização, o desvio padrão chegou a 12%, em cenários com muitos núcleos. Para as sequências grandes, o desvio ficou entre 0 e 2%.

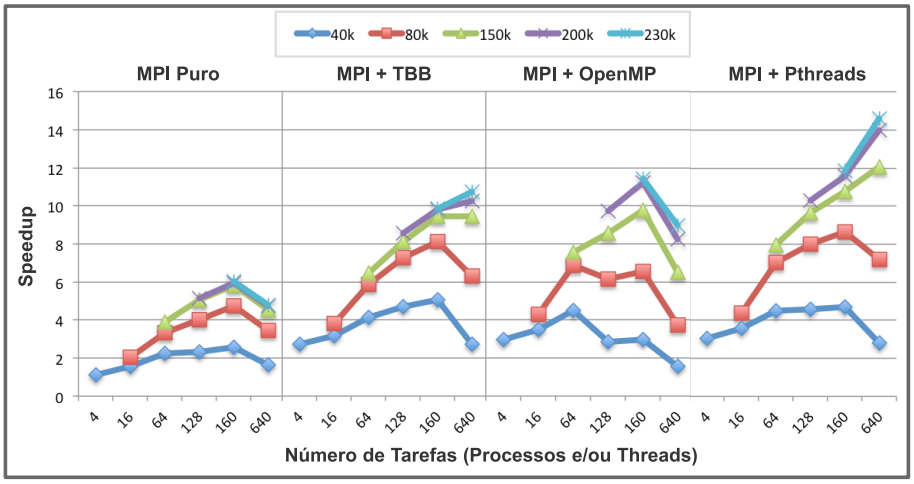


Figura 2. Desempenho do *LCS* num *cluster*. Cada quadro mostra os resultados para uma solução diferente: MPI puro, MPI+TBB, MPI+OpenMP e MPI+Pthreads. O eixo x mostra a quantidade de tarefas usadas. Para a versão MPI puro cada tarefa é um processo, e nas demais versões cada processo cria 4 *threads* ($processos = tarefas/4$). O eixo y mostra o *speedup* obtido. Cada linha representa a medição para a *LCS* entre um par de sequências de um determinado tamanho (40 mil, 80 mil, 150 mil, 200 mil e 230 mil caracteres).

Como esperado, todas as versões conseguiram executar cadeias maiores a medida que o número de nós (e consequentemente memória) aumentou. Por exemplo, enquanto o par de cadeias de tamanho 40 mil pode ser executado em todas as configurações de nós, a cadeia de 230 mil só pode ser executada utilizando todos os 40 nós do *cluster*, para que a matriz coubesse na memória. Isso mostra que é possível executar cadeias cada vez maiores, desde que existam mais nós disponíveis. A piora de desempenho apresentada

para algumas execuções na última parte do gráfico (quando há um aumento de 160 para 640 tarefas) em todas versões, está explicada mais adiante.

A versão MPI puro foi a que obteve o pior desempenho por ser a menos eficiente no paralelismo interno aos nós. Ao criar 4 processos em cada nó, as comunicações internas aumentaram a sobrecarga, obtendo o pior *speedup*. Cabe ressaltar, que para calcular o *speedup* foi necessário o tempo sequencial para cada uma das execuções. Porém, como não foi possível realizar este cálculo para cadeias com mais de 40 mil caracteres pela limitação da memória de um único nó (apenas 8 GB), foi feita uma interpolação utilizando valores reais para cadeias menores e calculada uma previsão do tempo sequencial.

Entre as três versões que otimizam a comunicação e, principalmente, o paralelismo interno, MPI+TBB foi a que obteve o pior desempenho, quando consideramos apenas o *speedup* máximo obtido. Entretanto, sua escalabilidade foi bem superior a versão MPI+OpenMP, principalmente para as cadeias de tamanhos 40 mil e 80 mil. Repare que enquanto na versão MPI+TBB o ganho de desempenho foi aproximadamente constante a medida que a quantidade de nós aumentava, na versão MPI+OpenMP há uma queda de desempenho a partir da execução com 128 tarefas. O motivo dessa queda é a diminuição da granularidade das tarefas de cada nó, o que faz com que o custo de comunicação passe a ter uma sobrecarga maior, anulando o benefício de se executar mais tarefas menores em uma quantidade maior de nós. Além disso, como cada nó possui apenas quatro núcleos de processamento, o custo da barreira entre cada diagonal no OpenMP não chegou a ser um fator limitante no desempenho. Para nós com mais núcleos de processamento, o OpenMP perde para o TBB, como observado em [Alves et al. 2013].

O melhor desempenho foi obtido pela versão MPI + Pthreads, atingindo um *speedup* máximo de aproximadamente $14,6$ e, também, conseguindo um crescimento constante, com o aumento do número de nós.

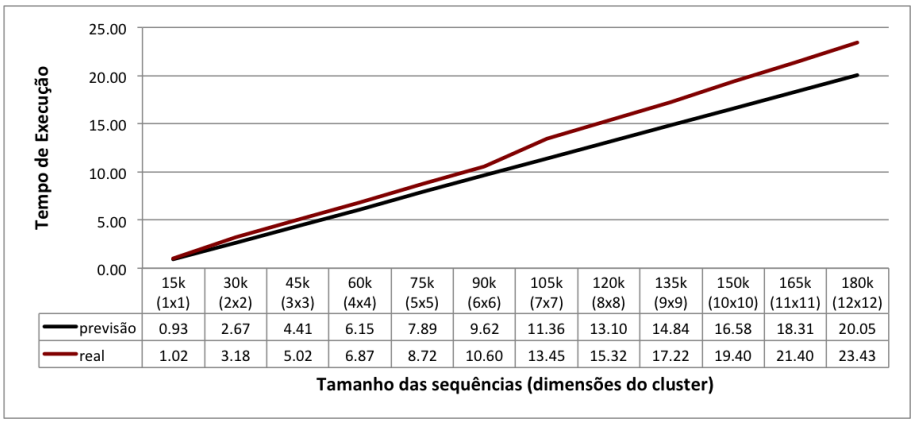
Diferentemente de todas as outras, as execuções com 640 tarefas (último ponto do gráfico da Figura 2) cria uma quantidade de processos maior do que a quantidade de *cores* totais disponíveis, que é de 160. O objetivo dessa abordagem é aumentar o paralelismo interno a cada nó, aumentando a quantidade de tarefas a serem processadas por cada *core*, diminuindo a ociosidade. Essa técnica obteve um excelente resultado para as versões MPI+TBB e MPI+Pthreads para cadeias grandes. Entretanto, tanto para a versão MPI puro como também para a versão MPI+OpenMP, os resultados foram sempre piores do que a execução com um processo para cada *core*, em virtude do aumento do custo de comunicação causado pela maior quantidade de processos e a granularidade mais fina das tarefas.

4.2. Previsão do Tempo de Execução com a Versão MPI Puro

Observando o padrão de comunicação do LCS e dado que as tarefas são balanceadas, é possível calcular uma previsão do tempo de execução paralelo com a versão MPI puro através do tamanho do maior caminho no grafo do bloco $[0, 0]$ ao bloco $[n-1, n-1]$. Neste caso, todos os caminhos possuem o mesmo tamanho de $C = Largura_{cluster} + Altura_{cluster} - 1$. Apenas com a medição do tempo T_{bloco} para computar um bloco (executado previamente na versão sequencial), onde o bloco é o pedaço da matriz atribuido a cada processo, é possível prever o tempo total de execução da versão MPI puro (T_{total}), com base em C e T_{bloco} com a fórmula: $T_{total} = T_{bloco} \times C$. Nesta previsão,

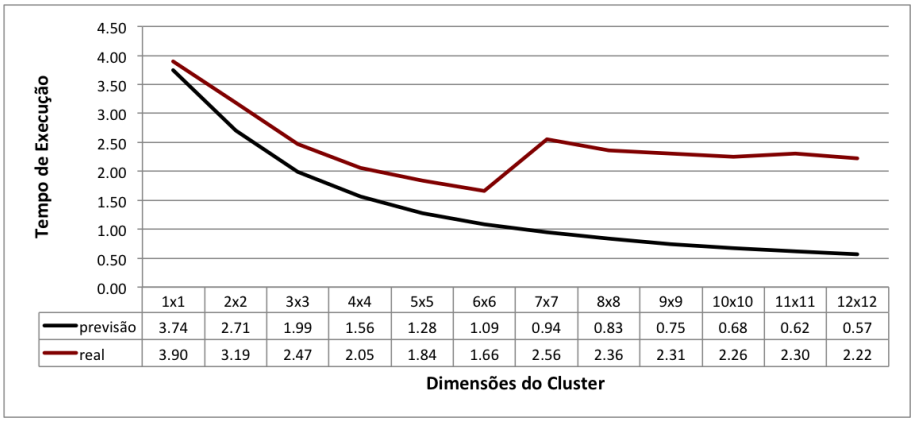
não estão considerados os custos de troca de mensagem, de maneira que este tempo pode ser considerado como um limite inferior do tempo de execução. O objetivo desta análise é avaliar a eficiência dos algoritmos implementados, comparando com o potencial máximo de paralelismo da estratégia utilizada (*wavefront*).

Vale lembrar, que essa estimativa é válida quando existe garantia de processadores disponíveis no cluster para executar cada processo, tão logo eles recebam a liberação de seus vizinhos. Nas versões híbridas, a fórmula se mantém, bastando usar como T_{bloco} o tempo de computação de um nó de forma paralela com memória compartilhada. O tempo de execução dentro de um bloco usando memória compartilhada não é trivial de estimar, pois o número de subblocos, em geral, é muito maior que o número de núcleos, podendo gerar espera na execução. Caso houvesse sempre a garantia de disponibilidade de núcleos, a mesma fómula poderia ser usada, pois a versão TBB e a versão Pthreads usam o mesmo grafo e na versão OpenMP, o número de diagonais é igual ao tamanho do caminho C, e cada diagonal é computada em paralelo.



	15k (1x1)	30k (2x2)	45k (3x3)	60k (4x4)	75k (5x5)	90k (6x6)	105k (7x7)	120k (8x8)	135k (9x9)	150k (10x10)	165k (11x11)	180k (12x12)
previsão	0.93	2.67	4.41	6.15	7.89	9.62	11.36	13.10	14.84	16.58	18.31	20.05
real	1.02	3.18	5.02	6.87	8.72	10.60	13.45	15.32	17.22	19.40	21.40	23.43

Figura 3. Avaliação da precisão da previsão de tempo de execução pelo cálculo do caminho crítico com bloco de tamanho fixo (15k x 15k), a medida que o número de *cores* aumenta.



	1x1	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10	11x11	12x12
previsão	3.74	2.71	1.99	1.56	1.28	1.09	0.94	0.83	0.75	0.68	0.62	0.57
real	3.90	3.19	2.47	2.05	1.84	1.66	2.56	2.36	2.31	2.26	2.30	2.22

Figura 4. Avaliação da precisão da previsão de tempo de execução pelo cálculo do caminho crítico com sequências de tamanho fixo (30k) e aumentando o número de *cores*.

A previsão em questão não considera os custos de criação e gerência de processos e nem o custo de troca de mensagens. As figuras 3 e 4 comprovam a precisão da fórmula com dois experimentos: (*i*) no primeiro (resultados na Figura 3) é fixado o tamanho de bloco (15k x 15k) e é aumentado o tamanho das sequências de forma quadrática para ocupar mais núcleos de processamento; (*ii*) no segundo (resultados na Figura 4) é fixado o tamanho das sequências (30k x 30k) e diminuido o tamanho dos blocos de forma quadrática para ocupar mais núcleos de processamento. As figuras mostram os tempos real e previsto para cada cenário. Em ambos os casos é possível ver que a medida em que são utilizados mais núcleos, o tempo real se afasta do tempo previsto. O aumento do tempo é justamente o custo da solução paralela (troca de mensagens, criação e gerencia de processos, entre outros).

É possível observar, que as soluções propostas se aproximam do tempo previsto, o que sugere que o paralelismo máximo possível está sendo explorado. No caso da Figura 4, há uma perda brusca de desempenho à partir da execução com 49 núcleos (7x7). Isso ocorre pois, neste experimento, a matriz possui um tamanho fixo e é particionada em blocos cada vez menores. Assim, a perda de desempenho aumenta muito em função da granularidade muito fina dos processos junto com o aumento da comunicação. Assumindo que um dos objetivos do trabalho é mostrar a viabilidade de se executar cadeias grandes é esperado que a granularidade dos processos seja grossa.

5. Trabalhos Relacionados

Os algoritmos para encontrar uma subsequência comum máxima podem ser classificados como exatos ou heurísticos. Os algoritmos de alinhamento de sequência exatos usam programação dinâmica, onde um problema é resolvido utilizando soluções de subproblemas [Dasgupta et al. 2006]. Isto ocorre através do processo de preenchimento de uma matriz contendo todos os possíveis alinhamentos entre duas ou mais sequências, o que garante a solução ótima. Por sua vez, algoritmos heurísticos utilizam técnicas para evitar o uso excessivo de processamento e memória, mas não garantem a solução ótima.

Entre os algoritmos exatos existem dois principais. O primeiro é conhecido como Needleman-Wunsch [Needleman and Wunsch 1970], desenvolvido na década de 70, que foi o primeiro algoritmo de alinhamento global criado. O segundo, conhecido como Smith-Waterman [Smith and Waterman 1981], desenvolvido na década de 80, é uma variação do algoritmo NW (Needleman-Wunsch) utilizado para alinhamento local.

Chen, Yu e Len (2006), em seu artigo, implementaram o algoritmo de Needleman-Wunsch utilizando a técnica de paralelismo *wavefront* para *clusters* utilizando MPI. A estratégia adotada para paralelização foi subdividir a matriz de comparações em colunas de blocos, onde cada processador é responsável por computar uma linha de blocos. Aplicando esta técnica de paralelismo, afirmaram que a complexidade de tempo do algoritmo foi reduzida para $O(n)$ quando utilizados n processadores. Porém, um grave problema deste algoritmo é juntar todos os blocos de uma mesma linha em um único processador, o que pode extrapolar a memória do processador, principalmente para matrizes grandes.

Existem muitos outros trabalhos no campo da Bioinformática em relação à paralelização de algoritmos de alinhamento de sequências. Nestas implementações diversas estratégias e arquiteturas computacionais são utilizadas, mas em grande maioria, o método de preenchimento da ma-

triz de comparações que oferece maior vantagem em uma paralelização é o *wavefront* [Yap et al. 1998, Alves et al. 2003, Batista and Magalhaes 2006, Cehn and Schmidt 2003, Chen et al. 2006, Martins et al. 2001, Seguel and Torres 2011, Naveed et al. 2005, Steinfadt et al. 2006, Rajko and Aluru 2004].

6. Conclusões e Trabalhos Futuros

Nesse artigo foi avaliado o desempenho do algoritmo paralelo *LCS* baseado na técnica de *wavefront*, num *cluster multicore* com 40 máquinas. Para isso foram implementadas quatro versões paralelas usando as bibliotecas OpenMPI, Pthreads, OpenMP e TBB. A análise de desempenho foi realizada com diferentes cenários, onde foram variados os tamanhos das sequências de entrada e a quantidade de processadores.

Os resultados obtidos mostraram que é possível executar o algoritmo *LCS* para cadeias grandes distribuindo a quantidade de memória necessária para encontrar a solução. O melhor desempenho foi obtido com a versão MPI+Pthreads para uma cadeia de 230 mil caracteres, onde o *speedup* máximo foi de aproximadamente 14,6 e com crescimento constante. A versão com *speedup* mais baixo foi a implementada com MPI puro, que não aproveita o compartilhamento interno aos *cores*.

Nos experimentos realizados, os nós ficam ociosos a maior parte do tempo, pois cada nó só computa um bloco da matriz (ou 4 blocos, na versão MPI puro). Na teoria, essa ociosidade pode ser eliminada com matrizes com mais blocos, de forma que cada nó trabalhe múltiplas vezes. Em uma matriz de 40x1000 blocos executada em um *cluster* de 40 nós, por exemplo, cada nó i ($0 \leq i < 1000$) trabalharia $1000 - i$ vezes. O problema é que os custos de troca de mensagem para matrizes com muitos blocos tornam essa prática proibitiva no *cluster* usado no experimento. A rede de baixa velocidade e falta de um sistema de arquivos distribuído são possíveis causas desse problema. Além disso, para uma avaliação mais profunda da abordagem híbrida (TBB, Pthreads ou OpenMP) é necessário realizar experimentos em um *cluster* com nós que possuam uma maior quantidade de núcleos de processamento. Investigar estes fatores e avaliar as soluções com outras configurações de matrizes é objeto de trabalhos futuros.

Agradecimentos

À FAPERJ e ao CNPq pelo apoio dado aos autores deste trabalho.

Referências

Alves, C. E. R., Cáceres, E. N., Dehne, F., , and Song, S. W. (2003). A parallel wavefront algorithm for efficient biological sequence comparison. In *Lecture Notes in Computer Science*, pages 249–258.

Alves, T. A., Marzulo, L. A. J., and Franca, F. M. G. (2013). Unleashing parallelism in longest common subsequence using dataflow. In *4th Workshop on Applications for Multi-Core Architectures*.

Axelson-Fisk, M. (2010). *Comparative Gene Finding: models, algorithms and implementation*. Springer.

Batista, R. and Magalhaes, A. (2006). An exact and parallel strategy for local biological sequence alignment in user-restricted memory space. In *Cluster 2006*, pages 1–10.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Cehn, C. and Schmidt, B. (2003). Computing large-scale alignments on a multi-cluster. In *Cluster 2003*, pages 38–45.

Chen, Y., Yu, S., and Leng, M. (2006). Parallel sequence alignment algorithm for clustering system. In *Shangai. Knowledge Enterprise: Intelligent Strategies in Product Design, Manufacturing, and Management*, pages 311–321.

D. A. Benson, M. Cavanaugh, K. C. I. K.-M. D. J. L. J. O. and Sayers, E. W. (2013). Genbank. *Nucleic Acids Research*, 41:36–42.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

Dasgupta, S., Papadimitriou, C., and Vazirani, U. (2006). *Algorithms*. McGraw-Hill.

Martins, W., Cuvillo, J., Useche, F., Theobald, K., and Gao, G. (2001). A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *PSB proceedings*, pages 311–322.

Naveed, T., Siddiqui, S., and Ahmed, S. (2005). Parallel needleman-wunsch algorithm for grid. In *Proceedings of the PAK-US International Symposium on High Capacity Optical Networks and Enabling Technologies*, pages 19–21.

Needleman, S. B. and Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol*, 48(3):443–453.

openMPI (último acesso 31/07/2014). open MPI. `http://www.open-mpi.org/`.

Rajko, S. and Aluru, S. (2004). Space and time optimal parallel sequence alignments. *IEEE Trans. Parallel Distrib*, 15(12):1070–1081.

Reinders, J. (2007). *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly.

Seguel, J. and Torres, C. (2011). Parallelization of needleman-wunsch string alignment method. In *BIOCOMP, 2011*, pages 239–244.

Smith, T. F. and Waterman, M. (1981). Identification of common molecular subsequences. *J. Mol. Biol*, 147(1):195–197.

Steinfadt, U., Scherger, M., and Baker, J. W. (2006). A local sequence alignment algorithm using an associative model of parallel computation. In *Proc. of IASTED Computational and Systems Biology*, pages 38–43.

TBB FlowGraph (último acesso 31/07/2014). TBB FlowGraph. `http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *J. ACM*, 21(1):168–173.

Yap, T. K., Frieder, O., and Martino, R. L. (1998). Parallel computation in biological sequence analysis. *IEEE Trans. Parallel Distrib*, 9(3):283–294.