Um Modelo de Desempenho Multinível para Algoritmos Paralelos *Wavefront 2D* Clássicos

Alexandre M. Lauredo, Alexandre C. Sena, Maria Clicia S. de Castro, Leandro A. J. Marzulo, Tiago A. O. Alves

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

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

Abstract. *The Classic 2D-wavefront pattern is commonly used to parallelize dynamic programming algorithms, where data elements or cells are distributed on a matrix. Since elements in a diagonal are completely independent, they can be computed in parallel. In modern systems, such as multicore clusters, a multi-level parallelism approach is usually adopted and selecting the size of each task is of critical importance to maximize performance. On one hand, coarse-grained tasks will result in low concurrency, while fine-grained tasks increase overhead. In this work we analyse this trade-off, considering aspects such as the task sizes (in two levels) and memory limitations. As a result of this analysis, we propose a model that predicts which task size configurations would yield optimal performance, based on the aforementioned parameters. To evaluate the model accuracy, we use the classic Longest Common Subsequence problem (LCS), which is an important part of DNA sequence alignment.*

Resumo. *O padrão wavefront 2D clássico é comumente usado para paralelizar algoritmos de programação dinâmica, onde elementos ou células de dados são distribuídos em uma matriz. Como os elementos da diagonal são completamente independentes, eles podem ser computados em paralelo. Em sistemas modernos, como os clusters multicore, usualmente é adotada uma abordagem de paralelismo multinível. A seleção do tamanho de cada tarefa tem uma importância crítica para maximizar o desempenho. As tarefas com granularidade grossa resultam em baixa concorrência, enquanto as de granularidade fina podem produzir alta sobrecarga. Neste trabalho analisa-se este balanço (trade-off), considerando aspectos tais como o tamanho das tarefas (em dois níveis) e limitações de memória. Como resultado desta análise, é proposto um modelo que prediz quais configurações de tamanhos de tarefas produzem alto desempenho, com base nos parâmetros mencionados. Para avaliar a precisão do modelo é usado o algoritmo clássico para encontrar a maior subsequência comum (LCS), que é uma parte importante do alinhamento de sequências de DNA.*

1. Introdução

O padrão *wavefront* clássico [Anvik et al. 2002, Mohanty and Cole 2007] é uma das técnicas mais utilizadas para paralelizar algoritmos de programação dinâmica. Uma matriz multidimensional representa a relação de recorrência em algorimos de programação

dinâmica, onde todos os elementos na mesma diagonal são independentes e podem, portanto, ser calculados em paralelo. No entanto, para diminuir os custos de gerenciamento de *thread* e de sincronização, os elementos são normalmente organizados em blocos de granularidade grossa que seguem o mesmo padrão de dependência do problema original.

Em *clusters multicore* modernos, é comum adotar uma implementação híbrida que utiliza a biblioteca MPI para comunicação entre os nós e memória compartilhada dentro de cada nó. Nesta abordagem, é empregada uma matriz *wavefront* com divisão em dois níveis. A divisão interna é usada para distribuir o trabalho entre os *cores*, enquanto a divisão externa é usada para distribuir o trabalho entre máquinas do *cluster*. Neste contexto, uma divisão de dados eficiente é de extrema importância para se conseguir alto desempenho. Além disso, diversas aplicações *wavefront* são limitadas pela memória. Este fato deve ser observado ao se comprar/escolher um sistema para executar tais aplicações.

Este trabalho propõe um modelo de desempenho multinível para avaliar algoritmos paralelos *wavefront*, considerando as limitações de memória e custo de paralelização. Como estudo de caso, é implementada uma aplicação que calcula a maior subsequência comum (LCS) entre duas cadeias, sendo este algoritmo importante para aplicações bioinformática. A implementação é baseada na versão MPI descrita em [Chen et al. 2006] e estendida com o paralelismo interno, usando OpenMP e TBB.

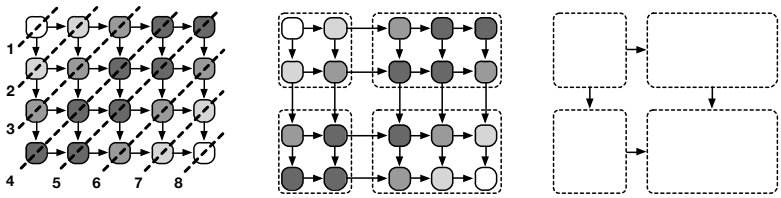
Os resultados mostram que o modelo de desempenho pode fornecer um limite superior realístico para o *speedup* potencial de um sistema específico, considerando a contenção de memória. Além disso, análises de execuções reais mostram que uma estratégia gulosa que inicia com tarefas de granularidade fina e que se tornam mais grossas seguindo o gradiente em cada passo, pode encontrar parâmetros de divisão de tarefa perto dos valores ótimos. Isto não só pode auxiliar os usuários a ajustarem suas aplicações, obtendo bom desempenho em seus sistemas, mas também fornecer a eles um conjunto de diretrizes para especificar um novo sistema ou atualizar um sistema existente.

O restante deste trabalho está organizado da seguinte forma: (*i*) Seção 2 descreve como é o padrão *wavefront 2D* clássico e explica o algoritmo LCS; (*ii*) Seção 3 discute a concorrência no padrão *wavefront* e apresenta o modelo de desempenho; (*iii*) Seção 4 descreve a implementação híbrida do LCS; (*iv*) Seção 5 apresenta os resultados comparando o desempenho previsto pelo modelo com a execução real e discute como a estratégia gulosa poderia ser usada para selecionar os parâmetros de divisão de tarefas; (*v*) Seção 6 aborda alguns trabalhos relacionados; (*vi*) Seção 7 conclui e discute os trabalhos futuros.

2. O Padrão *Wavefront 2D* Clássico

O padrão *wavefront* [Anvik et al. 2002] é amplamente adotado para paralelizar algoritmos de programação dinâmica que podem ser representados por uma relação de recorrência multidimensional. No *wavefront* é utilizada uma matriz multidimensional, e seus valores são computados de um canto da grade do problema ao canto oposto, atravessando a matriz diagonalmente, como pode ser visto na Figura 1(a). Consequentemente, todos os elementos em uma diagonal são completamente independentes entre si e podem ser computados concorrentemente, permitindo o uso de computação paralela [Mohanty and Cole 2007].

Uma aplicação clássica que emprega o padrão paralelo *wavefront* é a Longest Common Subsequence (LCS). Dada uma sequência de símbolos $S = \langle a_0, a_1, ..., a_n \rangle$,



(a) padrão (b) divisão em blocos (c) comunicação

Figura 1. *Wavefront* 2D e divisão em blocos para engrossar o grão.

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 LCS de X e Y é uma subsequência que aparece em X e Y e tem o maior tamanho. O comprimento $M[m, n]$ de uma LCS de sequências $A = \langle a_1, a_2, ..., a_m \rangle$ e $B = \langle b_1, b_2, ..., b_n \rangle$ pode ser definido, recursivamente, como:

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

Desta definição, proposta em [Wagner and Fischer 1974], pode ser diretamente derivado um algoritmo de programação dinâmica. Além disso, pode ser aplicado o padrão *wavefront*. Isto porque cada elemento $M[i, j]$ da matriz de valores depende de seus vizinhos superior ($M[i-1, j]$), esquerdo ($M[i, j-1]$) e superior-esquerdo ($M[i-1, j-1]$).

O alinhamento de sequências é um mecanismo extremamente importante para investigar a similaridade entre as espécies. Com Frequência, sequências genéticas semelhantes são relacionadas a grandes similaridades entre estruturas moleculares ou funcionais. Neste contexto, encontrar a maior subsequência comum (LCS) entre duas sequências é uma tarefa essencial no alinhamento de sequências de DNA.

Em razão do enorme crescimento dos bancos de dados de Genomas, como o GenBank, e do tamanho das cadeias de DNA, por exemplo o genoma de um simples Drosophila é composto de $122.653.977$ pares de base, enquanto o genoma humano é formado por $3,3 \times 10^9$ pares de base [D. A. Benson and Sayers 2013], algoritmos paralelos são extremamente necessários para comparar grandes sequências. Devido a grande quantidade de memória necessária para comparar grandes cadeias, uma arquitetura de memória distribuída é mais adequada, pois permite a divisão da matriz entre os vários nós do sistema.

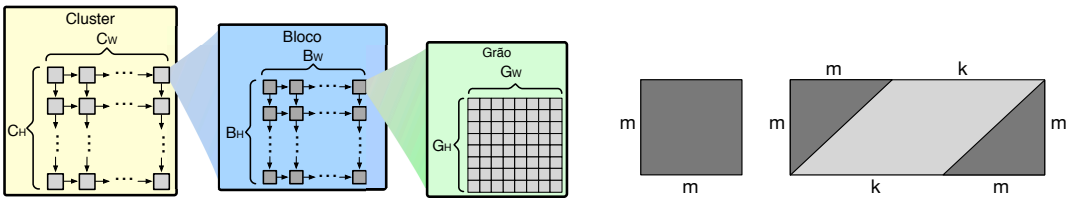
Uma implementação multinível do LCS, usando a técnica *wavefront*, não somente permite executar mais rápidamente, mas também avaliar sequências maiores. Além disso, como o LCS é uma aplicação limitada pela memória, ela se torna uma boa candidata para avaliar o modelo de desempenho proposto.

3. Análise de Concorrência do *Wavefront*

Esta seção analisa brevemente os *speedups* potenciais que podem ser obtidos em uma implementação paralela *wavefront* para *clusters multicore*. Em um cenário real, a criação de uma tarefa para computar somente um elemento da matriz *wavefront* (M) pode gerar muita sobrecarga. Por isto, uma solução comum é dividir a matriz em blocos, como pode ser visto na Figura 1(b). O padrão de comunicação observado entre os blocos é o mesmo que o do *wavefront* original, como mostrado na Figura 1(c). Este comportamento fractal

torna mais fácil para os desenvolvedores escolher o tamanho dos blocos que melhor se ajustam aos seus sistemas, sem se preocupar em reescrever o código. Essa subdivisão de blocos pode ainda ocorrer em múltiplos níveis para se ajustar à arquitetura do sistema.

Em *clusters multicore* são muito adotadas as implementações híbridas, que usam memória distribuída para comunicação entre nós (MPI) e memória compartilhada (threads) dentro de cada nó. Logo, é necessária uma abordagem em dois níveis composta por paralelismo interno (*multicore*) e externo (*cluster*). O *speedup* potencial interno é limitado pelo número de *cores* da máquina, enquanto o *speedup* potencial externo é limitado pelo total de nós. A Figura 2(a) mostra a divisão em dois níveis da matriz de valores para o processamento paralelo em *clusters multicore*. O primeiro nível é denominado *Cluster*, que é uma matriz de $C_H \times C_W$ Blocos, sendo C_H e C_W a altura e largura do *Cluster*, respectivamente. Os Blocos são as unidades de trabalho dos processos MPI. No segundo nível, cada Bloco é uma matriz de $B_H \times B_W$ grãos, sendo B_H e B_W a altura e largura do Bloco, respectivamente. Finalmente, os grãos são as unidades de trabalho das *threads*, cada uma representando uma fatia real da matriz de valores a ser computada sequencialmente. A altura e a largura do grão são G_H e G_W, respectivamente.



(a) níveis (b) geometria

Figura 2. Divisão em níveis e formato da matriz.

Dada uma matriz *wavefront* M composta por $m \times n$ elementos, a complexidade do tempo de execução serial é $O(m \times n)$. Por outro lado, o tempo de execução paralelo em um nível é limitado pelo caminho crítico do grafo de dependência *wavefront* [Arora et al. 1998] (Figura 1(a)). Este caminho vai de $M[0,0]$ a $M[m-1, n-1]$ com comprimento $m + n - 1$. Considerando que todos os blocos têm aproximadamente o mesmo tamanho e que o mesmo se aplica aos grãos, todos os caminhos têm o mesmo tempo de execução. Portanto, a Equação 1 representa o *Speedup* Potencial Interno (S_I), enquanto a Equação 2 representa o *Speedup* Potencial Externo (S_E).

$$S_I = \frac{B_H \times B_W}{B_H + B_W - 1} \qquad (1) \qquad\qquad S_E = \frac{C_H \times C_W}{C_H + C_W - 1} \qquad (2)$$

Para calcular o *Speedup* Potencial (S) é necessário considerar o número de nós N (máquinas em um *cluster*) e o número de processadores P (*cores* na máquina). Isto porque, eles limitam o *speedup* potencial máximo que pode ser atingido em cada nível. Assim, o *speedup* potencial é um produto dos *Speedups* Potenciais Interno e Externo, considerando os limites mencionados anteriormente, como pode ser visto na Equação 3. É importante destacar que o *Speedup* Potencial máximo é um limite superior quando não são considerados os custos de comunicação para o *Speedup* Externo e nem os custos de

gerenciamento e sincronização para o *Speedup* Interno.

$$S = \min\{\min\{S_I, P\} \times S_E, P \times N\} \tag{3}$$

O *Speedup* Potencial máximo que pode ser obtido em um nível depende da geometria da matriz *wavefront*. Por exemplo, considere duas matrizes *wavefront* M e M' ambas com 16 elementos, mas com dimensões 4×4 e 2×8, respectivamente. Ambas matrizes podem ser vistas como quadriláteros com a mesma área (16), que pode ser relacionada ao tempo de execução sequencial. Por outro lado, o tempo paralelo (tempo do caminho crítico) é diferente (7 para M e 9 para M'), podendo ser relacionado com o perímetro (ele é na verdade metade do perímetro). Portanto, o *Speedup* Potencial de M (≈ 2.29) é melhor que o *Speedup* Potencial de M' (≈ 1.78). De fato, dadas duas matrizes com o mesmo número de elementos (quadriláteros com a mesma área) o *Speedup* Potencial máximo é atingido em uma matriz quadrada (que tem o menor perímetro). Isto também é conhecido como o Problema Isoperimétrico, apresentado e provado em [Eggleston 1967].

Por outro lado, dadas duas matrizes com a mesma altura e larguras diferentes (áreas diferentes), o melhor *Speedup* Potencial (relação entre área e perímetro) é atingido pela matriz mais larga. A Figura 2(b) mostra uma matriz quadrada e uma retangular, com a mesma altura (m) e larguras diferentes (m e $m + k$, respectivamente). A Equação 4 mostra que, neste cenário, a matriz retangular fornece o melhor *Speedup* Potencial que a matriz quadrada. O lado esquerdo da equação representa o *Speedup* Potencial para a matriz quadrada, enquanto o lado direito mostra o *Speedup* Potencial para a matriz retangular.

$$\frac{m^2}{2 \times m - 1} < \frac{m^2 + k \times m}{(2 \times m + k - 1}, (m \geq 1 \wedge k \geq 1) \tag{4}$$

Uma razão para fixar a altura da matriz *wavefront* é quando se deseja executar a aplicação em um *cluster* e evitar o *overhead* de comunicação, criando um processo por máquina e dando uma linha de blocos para cada um daqueles processos (como pode ser visto na Seção 4). Neste caso, para aumentar o *Speedup* Potencial seria necessário ter mais blocos na horizontal (aumentar C_W), resultando numa matriz retangular, como mostrado na Figura 2(b). Como uma consequência, a largura do bloco (B_W) se tornaria menor, incorrendo em mais comunicação entre blocos, de acordo com o padrão *wavefront*.

A diferença de desempenho entre a memória e o processador pode ter um grande impacto na execução, especialmente quando considera as limitações de memória das aplicações [Sena et al. 2011]. Assim, um *speedup* teórico que não considera os custos de acesso à memória produz valores não realistas. Ao analisar como o gargalo da memória pode afetar o desempenho de uma aplicação *wavefront* é possível fornecer um cálculo aproximado dos custos dos acessos a memória e adicioná-los a equação do *Speedup* Potencial. Diferentemente do *speedup* teórico (Equação 3), limitações de memória são altamente dependentes das características da aplicação e da máquina.

Primeiro, assuma que a memória principal é o gargalo e que os dados em *cache* não causam contenção. Com isto em mente, é necessário encontrar quantas instruções são necessárias para computar um elemento na matriz *wavefront* e qual porção destas instruções acessam a memória principal. Existem pelo menos dois acessos por elemento,

uma leitura (falha compulsória) e uma escrita (*write back*). Mesmo se houver mais leituras nesta matriz, a localidade dos dados evita acessos extras a memória principal. Se são utilizadas estruturas de dados auxiliares além da matriz *wavefront*, deve-se verificar se elas adicionam acessos significativos a memória principal.

Considere que uma *thread* não pode realizar acessos a memória simultâneos no mesmo *core* e assuma que o número de instruções por ciclo (IPC) da aplicação é igual a 1 (em um superescalar). A Equação 5 mostra o número máximo de operações de memória (*bottleneck* da memória) que podem ser realizados simultaneamente se a aplicação é composta por 100% de acessos a memória (M_b), onde M_t é a taxa de transferência da memóra (em MT/s), M_c é o número de canais no barramento da memória e P_f é a frequência do processador (em MHz). Note que se IPC é diferente de um, é necessário dividir M_b pelo IPC. Neste trabalho assume-se que o IPC é igual a um e isto é válido para determinar o limite superior para nosso estudo de caso.

A Equação 6 mostra a razão (M_{ratio}) entre o número de instruções de acesso à memória (M_i) e o total de instruções (T_i) por elemento da matriz *wavefront*. Essa informação pode ser obtida através de ferramentas de *profiling*, como a Intel ®VTune ™ou Valgrind. Finalmente, a Equação 7 mostra o *Speedup* Interno máximo (S_{Imax}) que pode ser atingido, considerando o gargalo da memória da aplicação.

$$M_b = \frac{M_t \times M_c}{P_f} \quad (5) \qquad M_{ratio} = \frac{M_i}{T_i} \quad (6) \qquad S_{Imax} = \frac{M_b}{M_{ratio}} \quad (7)$$

Com essa informação, é necessário atualizar o *Speedup* Potencial (Equação 3) para incluir S_{Imax}. A Equação 8 mostra que se S_{Imax} é menor que o número real de *cores* do processador (P) e o *Speedup* Interno (S_I), ele limita o *Speedup* Potencial.

$$S = \min\{\min\{S_I, P, S_{Imax}\} \times S_E, P \times N\} \quad (8)$$

4. LCS Paralelo Híbrido para *Clusters Multicore*

Esta seção descreve duas implementações híbridas paralelas do algoritmo LCS que utilizam MPI para comunicação entre os nós e OpenMP ou Intel®Thread Building Blocks (TBB) para comunicação dentro de cada nó. Estas implementações são utilizadas como estudo de caso para analisar o modelo de desempenho proposto.

A implementação MPI é baseada em [Chen et al. 2006] e utiliza comunicação assíncrona somente de baixo para cima. Os processos MPI são organizados em um vetor, onde cada um é responsável por computar uma linha de blocos da matriz. A divisão vertical da matriz em blocos é apenas virtual, uma vez que não há necessidade de comunicação MPI na direção horizontal, dado que a linha inteira de blocos é computada pelo mesmo processo. Cada bloco libera a computação do vizinho de baixo (através de uma mensagem MPI) e do vizinho da direita (sem a necessidade de enviar mensagem), como pode ser visto na Figura 3. A largura do bloco virtual (B_W) é passada como parâmetro.

O trabalho realizado por cada processo MPI pode ser também paralelizado, utilizando memória compartilhada. Dessa maneira, somente um processo MPI por máquina é usado, enquanto múltiplas *threads* são empregadas. A ideia é dividir o bloco MPI em uma matriz de grãos que possam ser processados utilizando OpenMP ou TBB. No OpenMP, a computação de um bloco é realizada com dois laços aninhados. O laço de fora percorre

as diagonais da matriz de grãos sequencialmente, enquanto o laço de dentro computa os grãos da diagonal em paralelo, como pode ser visto na Figura 1(a). Por outro lado, no TBB, o conceito de *Flow Graph* é usado para implementar o padrão *wavefront*. Nas duas versões, as dimensões do grão (G_W e G_H) são iguais e passadas como parâmetros.

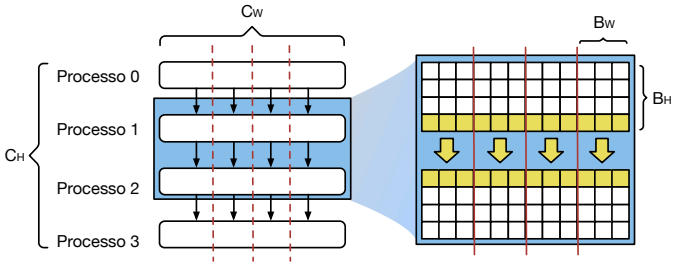


Figura 3. Solução paralela para memória distribuída.

Foram implementadas duas aplicações, usando os conceitos mencionados: **OMP+MPI** que utiliza uma abordagem de memória distribuída com comunicação assíncrona combinada com OpenMP para o paralelismo dentro de cada nó; **TBB+MPI** que utiliza uma abordagem de memória distribuída com comunicação assíncrona combinada com TBB para o paralelismo dentro de cada nó.

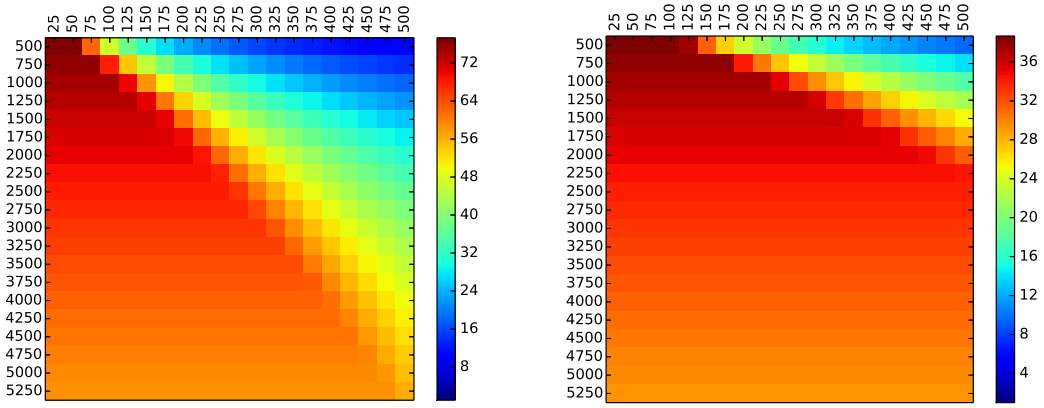
5. Análise Experimental

Esta seção apresenta uma série de experimentos para avaliar o desempenho do algoritmo LCS *wavefront* variando a granularidade das tarefas, considerando aspectos como o gargalo de memória e sobrecarga na utilização da memória distribuída. Inicialmente é realizado um *profiling* da aplicação LCS para que se estime a utilização da memória. Em seguida, é apresentada uma análise dos *speedups* teóricos considerando a configuração de um *cluster* real. Além disso, é realizada uma avaliação dos custos de usar memória compartilhada em uma máquina real e como isto impacta no *speedup* potencial. Por último, são apresentados os *speedups* reais para cada uma das versões híbridas apresentadas, seguida de uma discussão sobre como escolher as melhores configurações de bloco e grão.

Todos os experimentos foram realizados em um *cluster* composto de 10 nós, onde cada máquina possui dois processadores Intel®Xeon-E5345, com 4 *cores* cada um, velocidade de $2,33$ GHz, 8GB DDR2 667MT/s com dois canais, interconectados através de uma rede ethernet Gigabit. Nos experimentos foram considerados cenários com 10 processos MPI, um por máquina, e 4 e 8 *threads*.

Com base nas análises apresentadas na Seção 3, mapas de calor dos *spedups* potencias (teóricos) (Equação 3) são apresentados nas Figuras 4(a) e 4(b), considerando um *cluster* composto de 10 nós com 8 e 4 *cores*, respectivamente. A coordenada x mostra o tamanho de G_W, enquanto que a coordenada y o tamanho de B_W.

Os *speedups* teóricos aumentam à medida que a granularidade das tarefas diminui, o que significa menores valores para G_W e B_W. Assim, o melhor *speedup* teórico é alcançado na primeira célula do mapa de calor (500×25). É importante destacar que G_W define a granularidade das tarefas do nível mais interno (grão), enquanto que B_W define a granularidade das tarefas do nível externo (bloco).



(a) 10 nós com 8 *cores* (b) 10 nós com 4 *cores*

Figura 4. *Speedups* potenciais (teóricos).

Embora os mapas de calor com 4 e 8 *cores* sejam diferentes, o comportamento é similar, o que significa que o *speedup* aumenta a medida que se move para o lado esquerdo superior da matriz. Entretanto, o *speedup* teórico com 8 *cores* é, em geral, duas vezes melhor do que com 4 *cores*, o que era esperado, uma vez que os *speedup* internos teóricos (Equação 1) resultaram exatamente na quantidade de *cores*, 8 e 4, respectivamente.

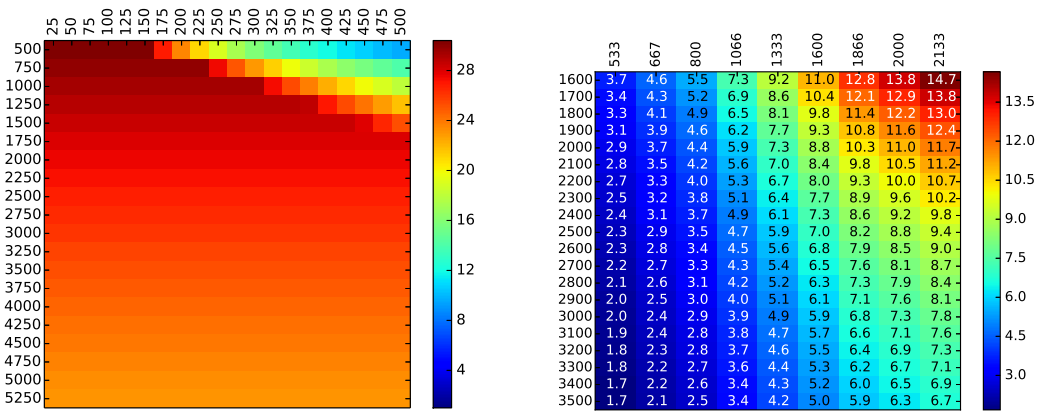
Portanto, para um *cluster* sem limitações no acesso a memória e nenhum custo de paralelização, o melhor seria escolher os menores valores possíveis para G_W e B_W. Entretanto, como tal cenário não existe, a seguir são analisados o impacto de tais custos e limitações no *speedup* teórico e, mais importante, a escolha de G_W e B_W que maximize o *speedup*.

Para avaliar o gargalo de memória, o *Valgrind* foi usado para mensurar a aplicação LCS e os resultados mostraram que 11 instruções são necessárias por elemento, onde 5 das quais são de memória, mas somente 2 acessam a memória principal, como discutido na Seção 3. Assim, LCS tem $M_{ratio} = 0.1818$ e $M_b = 0.5718$, resultando em $S_{Imax} = 3.1452$. Isto significa que os 8 *cores* disponíveis não serão eficientemente utilizados, como mostra a Figura 5(a), que é igual para 8 e 4 *cores*. A razão para este comportamento é simples: embora os nós com mais *cores* forneçam melhores *speedups* teóricos, na prática os acessos à memória limitam o *speedup* interno. Além disso, embora o comportamento deste novo *speedup* teórico seja o mesmo observado na Figura 4, os valores do *speedup* reduziram para menos da metade dos valores originais, no cenário com 8 *cores*.

Estes resultados são de extrema importância na hora de se comprar (ou escolher) um sistema com a melhor relação entre custo/desempenho para uma aplicação *wavefront*. A Figura 5(b) apresenta os valores de S_{Imax} para processadores com diferentes frequências e taxas de transferência de memória, em sistemas com 2 canais de acesso à memória. Uma boa opção para maximizar o desempenho é escolher a frequência do computador e quantidade de *cores* após calcular S_{Imax}, que será maior caso sejam escolhidas memórias com altas taxas de transferência e placas mãe com maior número de canais.

Embora o gargalo de memória limite o *speedup* teórico, o comportamento do mapa de calor é o mesmo, ou seja, o *speedup* aumenta à medida que os valores G_W e B_W diminuem (granularidade fina). Entretanto, em um cenário real, o aumento de tarefas de

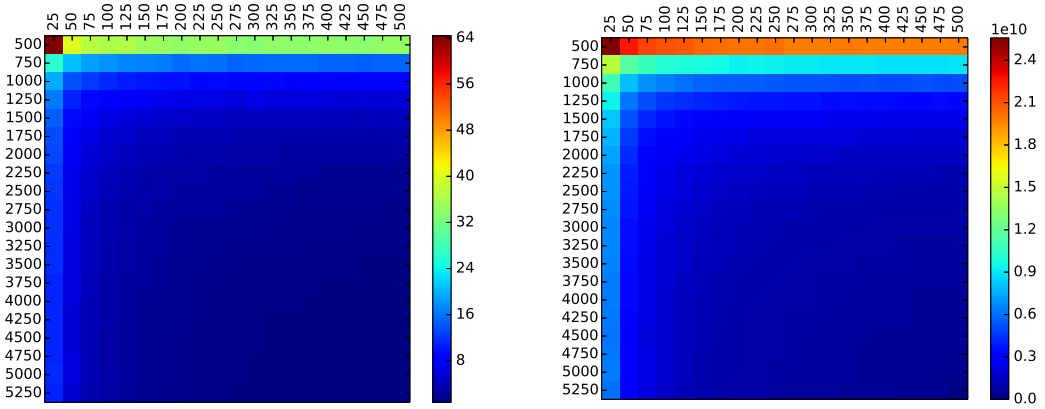
granularidade cada vez mais fina, provavelmente, aumentará a sobrecarga. A seguir, é analisado o impacto do custo de gerenciamento com o aumento no número de tarefas. Para medir a sobrecarga do gerenciamento de tarefas em um ambiente de memória compartilhada, uma aplicação com padrão *wavefront* foi executada, contendo apenas os laços e diretivas do OpenMP. Esta aplicação é basicamente um esqueleto do padrão *wavefront*.



(a) 10 nós com 8 *cores* e 10 nós com 4 *cores* (b) processador $\times$ memória em uma máquina

Figura 5. *Speedups* teóricos com limitações de memória

O tempo de execução dessa aplicação (em ms), variando os valores de G_W e B_W, pode ser visto na Figura 6(a). No pior caso a sobrecarga foi $\approx 0,5\%$ do tempo de execução da aplicação. Entretanto, o gerenciamento de tarefas do OpenMP inclui um número significativo de operações de memória, como apresentado na Figura 6(b). Estas operações de memória são realizadas em estruturas de dados pequenas que provavelmente permanecem na *cache* durante toda execução do programa e, em aplicações reais, elas podem aumentar a contenção nos acesso a *cache*, piorando o tempo de execução.



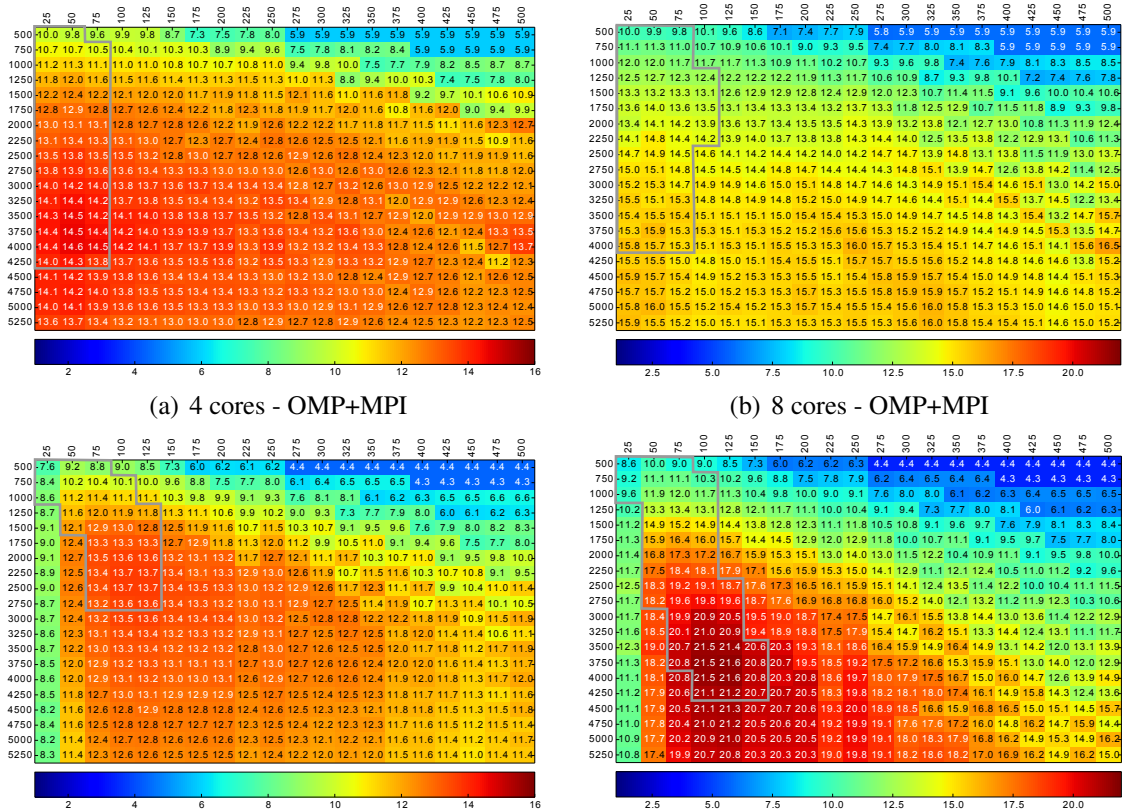
(a) Tempo (ms) (b) Operações de Memória

Figura 6. Custos do OpenMP.

Como esperado, contrariamente ao comportamento de todos os outros mapas de calor, a medida que os valores de G_W e B_W diminuem o tempo de execução cresce, aumentando a sobrecarga (Figura 6(a)). Este mesmo comportamento foi observado na

quantidade de operações de memória, onde o maior número de operações de memória foi obtido com $G_W = 25$ e $B_W = 500$. Este experimento claramente mostra o efeito colateral de aumentar-se a quantidade de tarefas de granularidade fina.

Finalmente, a Figura 7 apresenta mapas de calor com *speedups* reais para as aplicações LCS OMP+MPI e TBB+MPI, em 10 nós com 4 e 8 cores, ao alinhar cadeias de DNA compostas de $130k$ pares de base. Cada cenário, variando os valores de G_W e B_W, foi executado 20 vezes e o tempo médio foi utilizado para calcular o *speedup*. O desvio padrão foi calculado demonstrando valores desprezíveis. Repare que as escalas para 8 e 4 *cores* são diferentes para melhorar a visualização do comportamento dos *speedups*. Para as duas versões híbridas, foram usados 10 processos MPI (1 por máquina), enquanto que a quantidade de *threads* por nó é igual a 4 (Figuras 7(a) e 7(c)) e 8 (Figuras 7(b) e 7(d)).



(a) 4 cores - OMP+MPI

(b) 8 cores - OMP+MPI

(c) 4 cores - TBB+MPI

(d) 8 cores - TBB+MPI

Figura 7. *Speedups* para LCS híbrido.

Diferentemente de todos os outros mapas de calor, os melhores *speedups* estão localizados no centro esquerdo do mapa de calor, ao invés de estar localizado na parte superior esquerda. Este comportamento representa o balanceamento entre o paralelismo potencial e o custo de gerenciamento. Além disso, enquanto o limite superior representado pelo *speedup* teórico com contenção de memória (Equação 8) foi 28, o *speedup* máximo para as versões MPI+OMP e MPI+TBB foram $16,5$ e $21,6$, respectivamente. Os maiores custos de gerenciamento e, principalmente, a sincronização do OpenMP prejudicaram bastante o desempenho, enquanto a habilidade de descrever a aplicação como um *flow graph* do TBB, diminuiu as barreiras, permitindo um desempenho melhor.

Os *Speedups* teóricos que consideram o gargalo de memória sugerem que utilizar mais do que 4 *cores* não aumenta o desempenho ($S_{Imax} = 3.1452$). Isto somente acontece se a implementação paralela estiver aproveitando o desempenho máximo do sistema ou se ela mantiver a pressão no barramento de memória no mesmo nível do que a versão sequencial. No caso da versão MPI+OMP esta suposição é verdadeira, visto que nenhuma melhora significativa foi alcançada na execução com 8 *cores* (figuras 7(a) e 7(b)). Por outro lado, na versão MPI+TBB foram observadas melhoras significativas utilizando 8 *cores* (figura 7(c) e 7(d)). Isto sugere que os custos do TBB não aumentam a pressão no barramento, podendo inclusive aumentar a quantidade de instruções que não acessam a memória, diminuindo M_{ratio} e aumentando S_{Imax} (Equações 6 e 7).

Outro aspecto interessante do mapa de calor das aplicações híbridas LCS é mostrar que várias configurações de G_W e B_W resultaram em valores de *speedups* próximos do ótimo e, mais ainda, relativamente perto uns dos outros. Assim, pode ser possível encontrar configurações de bloco e grão próximas do valor ótimo sem a necessidade de executar todas as 400 combinações. Assumindo que a especificação de todos os custos de gerenciamento, comunicação e de contenção da memória é dependente da arquitetura, como analisado neste trabalho, uma estratégia gulosa pode ser usada para achar uma boa configuração. Começando com os menores tamanhos de grão e bloco (25 e 500) e iterativamente testando os vizinhos, seguindo o gradiente a cada passo. Na Figura 7, o caminho da estratégia gulosa em cada cenário é destacado com uma linha cinza sólida. A melhor solução foi encontrada em três dos quatro cenários. No caso da versão MPI+OMP com 8 *cores* um ótimo local foi encontrado, sendo apenas 4% pior do que a melhor solução.

6. Trabalhos Relacionados

Um *framework* baseado em aprendizagem de máquina para aplicações *wavefront* foi apresentado em [Mohanty and Cole 2007]. Ele facilita o particionamento dos dados em sistemas compostos de CPUs *multicore* com múltiplos aceleradores GPU. Não são apresentadas análises do comportamento da aplicação *wavefront*.

A perda de desempenho na execução de matrizes 3D em sistemas *multicore* foi observada em [Sena et al. 2011]. Os autores, através da avaliação de desempenho da aplicação RTM usando a ferramenta Intel VTune, detectaram que a principal razão para não se obter *speedup* linear dentro das máquinas *multicore* foi o gargalo da memória. Embora, as técnicas de *tiling* melhorem os acessos a memória, a diferença de velocidade entre memória e processador prejudica o barramento da memória e aumenta as falhas da *cache*, ocorrendo perda de desempenho, especialmente quando todos os *cores* são usados.

Uma implementação MPI do algorimo LCS baseada em Needleman-Wunsch foi apresentada em [Chen et al. 2006] que, para evitar as comunicações horizontais, divide a matriz em colunas virtuais de blocos e cada processador é responsável por computar uma ou mais linhas de blocos. Entretanto, os experimentos foram executados em um *cluster* de máquinas com um único *core* com somente 256MB de memória cada, limitando o tamanho das sequências. Além disso, este trabalho não fornece análise de desempenho teórica e não avalia a relação entre tamanho de blocos e desempenho.

7. Conclusões e Trabalhos Futuros

Wavefront é uma técnica importante para paralelizar algoritmos de programação dinâmica.

Entretanto, encontrar uma divisão de dados para executar em *clusters multicore* não é uma tarefa fácil, que depende não somente do padrão de paralelismo, mas também da arquitetura alvo e dos custos de paralelização.

Este trabalho propôs e analisou um modelo de desempenho multinível para aplicações *wavefront*. Os resultados mostraram que ele fornece limites superiores aproximados do *speedup*, considerando a contenção de memória. Mais importante, ele pode não só ser usado para ajudar escolher ou comprar um *cluster* mais adequado para a aplicação alvo, mas principalmente, avaliar se o desempenho da aplicação *wavefront* está adequado. Além disso, uma estratégia gulosa pode ser usada para descobrir divisões de dados próximas do ótimo, permitindo uma execução eficiente.

Como trabalhos futuros, para entender melhor as implicações do modelo proposto, incluindo as análises dos custos de paralelização e também o gargalo de memória, será necessário executar o estudo de caso LCS em diferentes sistemas de computadores que permitam avaliar, por exemplo, quão próximo de S_{imax} é possível chegar. Tais sistemas devem ter uma quantidade grande de *cores*, memórias mais rápidas e com mais canais. Outro ponto importante a ser avaliado é a influência do tamanho da cache no modelo.

Agradecimentos

À FAPERJ (processo E-26/203.537/2015), ao CNPq, à CAPES e ao NCC/UNESP pelo apoio aos autores deste trabalho.

Referências

Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J., Bromling, S., and Tan, K. (2002). Generating parallel programs from the wavefront design pattern. In *Proceedings International Parallel and Distributed Processing Symposium., IPDPS 2002*, pages 8 pp–.

Arora, N. S., Blumofe, R. D., and Plaxton, C. G. (1998). Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, New York, NY, USA. ACM.

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.

Eggleston, H. G. (1967). The isoperimetric problem. In Press, P., editor, *Exploring University Mathematics*, volume 1, chapter 7. Pergamon Press.

Mohanty, S. and Cole, M. (2007). Autotuning wavefront applications for multicore multi-gpu hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 1:1–1:9, New York, NY, USA. ACM.

Sena, A. C., Nascimento, A. P., Boeres, C., Rebello, V., and Bulcao, A. (2011). An approach to optimise the execution of rtm algorithm in multicore machines. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 403–410.

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