Proceedings of the XXVI Iberian Latin-American Congress on Computational Methods in Engineering – CILAMCE 2005
Brazilian Assoc. for Comp. Mechanics (ABMEC) & Latin American Assoc. of Comp. Methods in Engineering (AMC),
Guarapari, Espírito Santo, Brazil, 19th – 21st October 2005

Paper CIL 12-1028

ESTRATÉGIAS DE UTILIZAÇÃO EFICIENTE DA MEMÓRIA CACHE EM UM ALGORITMO PARALELO PARA AS EQUAÇÕES DE NAVIER STOKES UTILIZANDO MALHAS ESTRUTURADAS EM PLATAFORMAS COM MEMÓRIA DISTRIBUÍDA

João Paulo De Angeli
Alberto Ferreira de Souza
Andréa Maria Pedrosa Valli
jpda@inf.ufes.br
alberto@inf.ufes.br
avalli@inf.ufes.br
Departamento de Informática, Universidade Federal do Espírito Santo,
Vitória, Av. Fernando Ferrari, 514, Goiabeiras, 29075-910, Vitória, ES, Brasil

Neyval Costa Reis Jr.
neyval@ambiental.com.br
Departamento de Engenharia Ambiental, Universidade Federal do Espírito Santo,
Vitória, Av. Fernando Ferrari, 514, Goiabeiras, 29075-910, Vitória, ES, Brasil

Resumo. *Este trabalho investiga uma estratégia de utilização eficiente da memória cache em um algoritmo numérico para a simulação de escoamentos de fluidos incompressíveis, baseado no método de diferenças finitas, projetado para plataformas de processamento paralelo com memória distribuída, particularmente para clusters de estações de trabalho. Para melhorar os níveis de desempenho obtidos pelo algoritmo, foi utilizada uma estratégia de decomposição de domínio em dois níveis, onde o primeiro nível de decomposição divide o problema entre os processadores, e o segundo nível divide o problema dentro do mesmo processador. Neste trabalho, a implementação desta estratégia é efetuada através da execução de mais de um processo MPI por processador. Os resultados experimentais mostram que esta estratégia aumenta o speedup em mais de 48%. São propostas também técnicas para a redução do volume de comunicação entre os processadores.*

Palavras Chaves: *memória cache, decomposição de domínio, MPI, cluster de estações de trabalho, diferenças finitas.*

1. INTRODUÇÃO

Os níveis de memória *cache* são consideravelmente mais rápidos que a memória principal do sistema e são utilizados para manter cópias dos blocos mais freqüentemente utilizados da memória principal – podendo transferir estes dados com baixa latência para a CPU sempre que forem necessários. A velocidade de execução de um código está relacionada à eficiência com que este utiliza a hierarquia de memória. Idealmente, um código computacional deve ser capaz de acessar variáveis (ou blocos de memória) que estão armazenados em um dos níveis da memória *cache*. Infelizmente, aplicações de Mecânica dos Fluidos Computacional (MFC) utilizam estruturas de dados grandes demais para caberem totalmente nas memórias *cache*. Assim, durante a execução de um código típico de MFC, a CPU tem que "esperar" pela "chegada" de dados armazenados na relativamente "lenta" memória principal. Isso faz com

que apenas uma fração do desempenho máximo do processador seja realmente alcançada. A utilização eficiente da hierarquia de memória dos sistemas pode aumentar a velocidade de processamento em até uma ordem de grandeza (Kowarschik, 2000). Pesquisas na área de otimização de métodos numéricos, com base em um melhor uso da memória *cache*, começaram apenas recentemente e ainda continuam em desenvolvimento, como os trabalhos de Tomko e Abraham (1994), Douglas *et al*. (2000) e Tseng (2000).

Kowarschik *et al.* (2000) analisa diversas técnicas de otimização da *cache* para métodos iterativos aplicados a sistemas lineares de equações, em particular para o método *multigrid*. Foram apresentadas três técnicas de otimização da utilização da memória *cache* nas repetidas relaxações Gauss-Seidel com ordenação *red-black*: *Fusion Technique*, *Blocking Technique* e *Windshield Wiper Technique*. No entanto, as técnicas apresentadas de codificação podem ser utilizadas em outros métodos iterativos, uma vez que os núcleos computacionais nas técnicas iterativas são bastante similares. Foi concluído que a aplicação dessas técnicas pode manter em 25% do pico de desempenho independentemente do tamanho da malha, que seriam perdidos se não fosse levada em consideração a utilização eficiente da memória *cache*. Os resultados foram obtidos em máquinas *DEC Alpha*, *SUN Ultra* e *SGI Origin*.

Gullerud e Dodds (2001) desenvolveram um algoritmo para solução de sistemas lineares, para computadores paralelos de memória distribuída, baseados em decomposição de domínio, empregando uma estratégia mais eficiente de utilização da *cache*. Neste algoritmo, após a divisão dos subdomínios entre os processadores, as varreduras dos *loops* são efetuadas em blocos suficientemente pequenos para otimizar o acesso à *cache*. Essa decomposição em dois níveis é similar ao proposto por este trabalho, onde o primeiro nível de decomposição divide o problema entre os processadores, e o segundo nível divide o problema dentro do mesmo processador.

Takahashi (2003) implementou o algoritmo FFT (*Fast Fourier Transform*) paralelo tridimensional para clusters de PCs. Foi possível reduzir o número de *cache misses* do algoritmo convencional FFT tridimensional, através da utilização da decomposição em blocos. O algoritmo proposto é mais vantajoso em máquinas que possuam diferenças consideráveis entre a velocidade da *cache* e da memória principal. Neste caso, foi obtido um ganho de 30% no tempo de processamento em relação aos métodos convencionais.

Silva (2003) apresentou uma revisão dos métodos disponíveis para otimização de acesso à *cache*, em uma abordagem similar a Kowarschik *et al*. (2000). A técnica foi aplicada ao algoritmo iterativo de Gauss-Seidel, obtendo melhoria de desempenho de 84%. Silva (2003) sugere, como trabalhos futuros, o desenvolvimento de modelos matemáticos para previsão de desempenho em algoritmos baseados na divisão em blocos, além de outras técnicas de distribuição de dados, permitindo aos programadores a avaliação das estratégias antes do desenvolvimento do código.

Nesse trabalho foi empregada uma estratégia de utilização mais eficiente da *cache* baseada em um algoritmo de decomposição de domínio que utiliza dois níveis de decomposição. No primeiro nível, o problema é dividido entre os processadores e, no segundo nível, o problema é dividido dentro de um mesmo processador. A implementação desta estratégia será efetuada através da execução de mais de um processo MPI por processador. A técnica baseia-se na divisão das estruturas de dados em blocos suficientemente pequenos para caber na memória *cache* do processador, favorecendo a reutilização de informações que estão armazenadas em *cache* e garantindo uma rápida disponibilidade dos dados ao processador. A idéia principal desta técnica consiste em definir o número de processos que cada processador deve executar com base nos requisitos de memória e tamanho da memória *cache* do processador, não havendo necessidade de nenhuma alteração significativa no código paralelo.

Este trabalho está organizado da seguinte maneira. Após a introdução, a Seção 2 apresenta rapidamente o algoritmo computacional implementado, além das equações

governantes dos problemas de MFC e as estratégias de paralelização empregadas. Na Seção 3 são apresentadas as estratégias de utilização eficiente da memória *cache*, apresentada a técnica de decomposição do problema entre os processadores e a técnica de divisão em um único processador. Em seguida são apresentados os experimentos computacionais e, por fim, as conclusões.

2. ALGORITMO COMPUTACIONAL

Os problemas tratados neste trabalho envolvem a simulação de escoamentos bidimensionais transientes de fluidos viscosos incompressíveis, governados pelas equações de conservação de massa e quantidade de movimento para fluidos Newtonianos, considerando viscosidade e densidade constante. As equações de Navier-Stokes e continuidade podem ser escritas como:

$$\frac{\partial u}{\partial t}+\frac{\partial p}{\partial x}=\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)-\frac{\partial (u^2)}{\partial x}-\frac{\partial (uv)}{\partial y}+g_x \tag{1}$$

$$\frac{\partial v}{\partial t}+\frac{\partial p}{\partial y}=\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)-\frac{\partial (uv)}{\partial x}-\frac{\partial (v^2)}{\partial y}+g_y \tag{2}$$

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \tag{3}$$

onde u e v são os componentes horizontal e vertical da velocidade, respectivamente, p é a pressão, gx e gy são os componentes da força de corpo, $Re=(\rho_\infty u_\infty L)/\mu$ é o número adimensional de Reynolds, ρ_∞, u_∞ e L são constantes escalares (ou seja: densidade do fluido, velocidade característica e comprimento característico, respectivamente) e μ é a viscosidade dinâmica. Para completar a formulação matemática do problema, são necessárias as condições iniciais e as condições de contorno.

O algoritmo desenvolvido é baseado no método das diferenças finitas, conforme a formulação proposta por Griebel, Dornseifer e Neunhoeffer (1998), que consiste na solução de um esquema implícito para pressão, através de sucessivas iterações de relaxação (SOR), e um esquema explícito para as velocidades com uma discretização no tempo de primeira ordem. Para evitar o surgimento de oscilações, foi usado um controle do tamanho do passo no tempo adaptável baseado na conhecida condição de Courant-Friedrichs-Lewy (CFL). Isso garante estabilidade no algoritmo numérico (Anderson, 1995).

A divisão da carga de trabalho entre os nós de processamento do cluster é feita através da técnica de bisecção das coordenadas do domínio (Streng, 1996). Nessa técnica o número de pontos do domínio é dividido igualmente entre os processadores, mas nenhum esforço é feito para se obter uma divisão do domínio que minimiza a comunicação entre os processadores. A troca de informação entre os subdomínios, ou processadores, é feita usando o padrão *Message Passing Interface* (MPI).

Para melhorar os níveis de desempenho obtidos pelo algoritmo, foram utilizadas técnicas para a redução do volume de comunicação entre processadores. Griebel, Dornseifer e Neunhoeffer (1998) propuseram que um passo da comunicação deveria ser executado para cada iteração SOR, de forma que os valores nos contornos sejam atualizados em todas as iterações SOR. Esse procedimento introduz uma quantidade significativa de comunicações, reduzindo a velocidade do processamento (De Angeli *et al.*, 2003). Sendo assim, neste trabalho é utilizado um procedimento alternativo, que executa cinco iterações SOR entre cada

comunicação. Isso reduz a quantidade de comunicação sem diminuir a velocidade de convergência.

Existem algumas tarefas no procedimento computacional que exigem comunicação entre todos os processadores, como por exemplo, o cálculo do tamanho do passo no tempo e o teste de convergência da solução no tempo. Neste trabalho, esta operação foi feita utilizando uma árvore binária, o que revelou ser uma estratégia de comunicação bastante eficiente quando comparada com a estrutura mestre-escravo e comunicação todos-para-todos (De Angeli *et al.*, 2003). Conforme o número de subdomínios aumenta, o número de mensagens se torna um importante fator limitante e tende a baixar a velocidade de processamento. De Angeli *et al.* (2003) observaram que a comunicação todos-para-todos revela a pior marca de eficiência paralela para números grandes de processadores, enquanto que a implementação mestre-escravo obtém um resultado intermediário.

Com o objetivo de melhorar a eficiência de utilização da memória *cache* dos processadores do cluster foram efetuadas divisões no domínio de forma que os dados de cada subdomínio fossem acomodados na memória *cache*. Isso garante que o processador não fique ocioso aguardando por informações provenientes da memória principal do computador, uma vez que essa é significativamente mais lenta que a memória *cache*. Na próxima seção, descrevemos as estratégias utilizadas para o uso mais eficiente da memória *cache*.

3. ESTRATÉGIAS DE UTILIZAÇÃO EFICIENTE DA MEMÓRIA *CACHE*

Esta seção descreve a implementação das estratégias de utilização eficiente da memória *cache*, especificando inicialmente os requisitos e o roteiro para a aplicação da técnica e em seguida detalhando alguns dos processos envolvidos: como a determinação do tamanho máximo do problema para que se obtenha uma taxa muito baixa de *cache miss;* como calcular teoricamente e experimentalmente as taxas de *cache miss* de escrita e leitura e como determinar o número de processos no qual fique garantido que a taxa de *cache miss* em cada um deles seja muito baixa. O termo *cache miss* é utilizado quando a memória *cache* não possui o bloco de dados que contenha a informação solicitada pelo processador ou por uma *cache* um nível acima na hierarquia. Também pode ocorrer *cache miss* quando alguma informação é escrita e o bloco destino não está presente em *cache*.

3.1. Requisitos e roteiro para a aplicação da técnica

Independentemente do algoritmo MFC, De Angeli (2005) mostrou que existem alguns requisitos para a aplicação da técnica proposta de utilização eficiente da memória *cache*: (1) o algoritmo deve ser iterativo, uma vez que o foco da eficiência é a reutilização dos dados; (2) o algoritmo deve ser paralelo, uma vez que a idéia é utilizar mais de um processo por processador, mesmo que seja apenas um processador. A técnica só apresenta benefícios se houver mais de uma iteração do método de convergência para cada comunicação entre os processos, visto que a técnica explora a reutilização dos dados; (3) é preciso conhecer o tamanho máximo do problema para que se tenha uma taxa de *cache miss* muito baixa. Isso pode ser feito por determinação teórica ou experimental.

Com os requisitos verificados, é necessário conhecer o número de processos (p) no qual fique garantido que a taxa de *cache miss* em cada um deles seja muito baixa. Para isto, p deve satisfazer as seguintes condições: (1) p deve ser um número inteiro igual ou maior que a razão entre a quantidade de pontos nodais (*QPN*) do problema a ser simulado pela quantidade de pontos nodais suportados (*QPNSC*) pela *cache*, garantindo assim, que os dados caibam na *cache*; (2) p deve ser um múltiplo do número de processadores utilizados na simulação, de tal

forma que cada processador possua o mesmo número de processos, proporcionando uma distribuição uniforme dos pontos nodais entre eles.

Para que as condições sobre o número de processos (p) sejam satisfeitas, temos que obedecer às relações:

$$p \geq \frac{QPN}{QPNSC} \quad e \quad p = k \times m \tag{4}$$

onde m é o número de processadores disponíveis para a execução da simulação, e k é um inteiro que representa a quantidade de processos por processador. Será verificado neste trabalho que existem perdas devido ao excesso de comunicação, ao se aumentar o número de processos por processador, portanto, deve ser utilizado o menor valor de k que satisfaça as condições acima.

3.2. Determinação do tamanho máximo do problema

Um dos objetivos deste trabalho é fazer com que haja o mínimo de *cache misses* no nível L2 de memória *cache*. Como o procedimento SOR do algoritmo é o que demanda maior esforço computacional, além de lidar com a maior parte da memória alocada para o processamento, é necessário saber o quanto de espaço de memória é utilizado neste procedimento. É preciso calcular também a utilização da memória pela função que testa a convergência do método SOR, pois este procedimento é o segundo na lista dos procedimentos que demandam mais computação, apesar de acessar praticamente a mesma região de memória que o SOR.

A matriz do sistema é armazenada por linha, onde a segunda linha da matriz é armazenada no vetor após a primeira linha, a terceira linha após a segunda e assim sucessivamente. Na implementação do método SOR são acessados três vetores, P, A e B, sendo que o vetor A é dividido em cinco partes, totalizando assim sete variáveis. Portanto, é necessário que existam sete variáveis para cada ponto nodal. A malha completa tem dimensão $N\times M$ (largura × altura), sendo que $(N\text{-}2)\times(M\text{-}2)$ são de pontos nodais úteis de processamento e o restante são para a comunicação com os outros subdomínios.

Ao acessar algum dado que não esteja na *cache* L2, ocorrerá *cache miss*, fazendo com que o nível 2 da *cache* busque não apenas o dado requisitado, mas sim uma quantidade de dados do tamanho de cada bloco da *cache* L2. Assim, mesmo que os vetores A e B não acessem dados nas fronteiras, é necessário contabilizar o espaço utilizado para os pontos nodais das fronteiras laterais. As fronteiras superior e inferior não são necessárias devido ao fato de nenhum dado das linhas superior e inferior serem acessados; nesse caso, o L2 não precisa armazenar nenhum bloco da *cache* para a primeira e a última linha.

As variáveis armazenadas nos vetores são do tipo *double*, que ocupam 64 bits (8 bytes) cada uma. Então, do vetor P são acessados $8\times M\times N$ bytes, incluem os elementos das fronteiras, pois o cálculo de P necessita dos valores de P vizinhos. Mas em relação aos valores do vetor B, as fronteiras não são necessárias para o cálculo de P, dessa forma, são acessados $8\times(M\text{-}2)\times N$ bytes. A primeira e a última linha da malha não são acessadas, enquanto que as fronteiras laterais são consideradas devido à forma de armazenamento da malha na memória, linha após linha. Cada elemento da fronteira lateral ocupa um mesmo bloco de *cache* que alguns elementos internos a malha. Com isso, ao ler o valor de um desses elementos internos, necessariamente o valor do elemento da fronteira ocupará espaço em *cache*. O vetor A possui as mesmas particularidades do vetor B, exceto por ser composto por

cinco partes, então, são acessados 8×5×(*M*-2)×*N* bytes. Portanto, o total de memória (vetores *P*, *A* e *B*) acessada pelo procedimento SOR é dado pela expressão abaixo:

$$Acessado = 8 \times \left[M \times N + 6 \times \left(M - 2 \right) \times N \right] \tag{5}$$

Nas máquinas utilizadas neste trabalho, o nível 2 da *cache* possui 256KB, sendo esse espaço utilizado tanto para dados quanto para instruções. Dessa forma, é necessário garantir que o espaço acessado pelo SOR seja menor que o tamanho da *cache* L2. Lembre-se que a expressão acima fornece somente o espaço utilizado nos vetores acessados pelo SOR, não considerando as variáveis e constantes utilizadas por esse procedimento. Para uma malha de 64×64 pontos, a utilização total da memória é de 233,03 Kbytes, o que representa uma utilização de 90,64% da *cache*. Sendo assim, a taxa de *cache miss* começa a crescer após o tamanho 64×64 (4096 pontos nodais), como será verificado nos experimentos numéricos.

3.3. Determinação das taxas de *cache miss* de escrita e leitura

Para a medição de *cache misses*, é possível utilizar o software Valgrind (Valgrind, 2004), uma ferramenta de depuração que emula uma CPU x86. Neste trabalho, a taxa de *cache miss* também foi medida teoricamente e o resultado comparado com os resultados obtidos usando o software Valgrind. Os procedimentos SOR e cálculo de resíduo são os que demandam maior esforço computacional, acessando repetidamente uma grande região de memória, por isso, influenciam fortemente a taxa de *cache miss* global. Esta taxa pode ser calculada tanto para leitura quanto para escrita de dados, conforme as expressões a seguir:

$$Taxa\ de\ miss\ de\ escrita = \frac{número\ de\ misses\ ao\ escrever}{número\ de\ escritas\ de\ dados} \tag{6}$$

$$Taxa\ de\ miss\ de\ leitura = \frac{número\ de\ misses\ ao\ ler}{número\ de\ leituras\ de\ dados} \tag{7}$$

Os procedimentos SOR e cálculo de resíduo não apresentam taxa de *cache miss* de escrita. A escrita em um determinado elemento do vetor *P*, que consiste no valor da pressão no centro de cada elemento, é a única do procedimento SOR com relevância em se tratando de *cache misses* de escrita, diferentemente do procedimento de cálculo de resíduo, que não faz nenhuma gravação repetidamente. Como a escrita de um elemento de *P* é precedida da leitura de seu próprio valor antigo, uma cópia da região de memória destino da escrita já está na memória *cache*. Desta forma, a taxa de *cache miss* de escrita é zero.

O total de *cache misses* de leitura pode ser quantificado teoricamente pela leitura dos vetores *B*, *A* e *P*. Para os problemas pequenos, onde todos os dados acessados cabem na *cache* L2, o número de *cache misses* é próximo de zero. Só não é zero devido aos dados acessados na primeira vez da execução do SOR ainda não estarem na *cache*. No entanto, para cada repetição, os dados ainda permanecerão em *cache*, não ocorrendo *cache misses* adicionais. Para os problemas maiores, cada execução do SOR, mesmo que consecutiva, não terá todos os dados disponíveis em *cache*, havendo um acréscimo da taxa de *cache miss*. Sempre que houver um *cache miss*, um bloco de 64 bytes da memória principal da máquina substituirá um bloco de *cache* que contém algum dado que futuramente será requisitado pelo próprio SOR, desencadeando uma seqüência de *cache misses*.

Considerando as particularidades de cada um dos vetores, é possível mostrar que o total de *cache misses* de leitura para o procedimento SOR é a razão entre a quantidade de bytes acessados (Eq. (5)) e o tamanho de cada bloco de cache (64 bytes). Maiores detalhes são apresentados por De Angeli (2005). A expressão resultante é:

$$L2d_{mrd} = \frac{M \times N + 6 \times (M-2) \times N}{8} \tag{8}$$

onde $L2d_{mrd}$ é o número de *cache misses* ao ler dados da *cache* L2. Esse valor é aproximado devido a alguns fatores como, por exemplo, o fato do início dos vetores, quando estão na *cache*, não estarem alinhados exatamente com o início de um dos blocos da *cache*, ou o fato de instruções e outras variáveis do procedimento ocuparem algum espaço no L2.

Para calcular o número de acessos de leitura à *cache* L2 realizado pelo procedimento SOR, foi examinado o código em C do algoritmo traduzido para *assembly*. Pode-se mostrar que o número de leitura de dados em uma iteração SOR é dado por:

$$D_{rd} = 18 \times (M-2) \times (N-2) + 12 \times (M-2) + 32 \tag{9}$$

onde D_{rd} é o número de leituras de dados. É importante notar que este número de acessos é função do código gerado pelo compilador utilizado, outro compilador poderia gerar um código *assembly* com um número ligeiramente maior ou menor de acessos à memória.

A taxa de *cache miss* de leitura no procedimento SOR é dada pela razão entre as quantidades dadas nas Eq. (8) e Eq. (9):

$$taxa\ L2d_{mrd} = \frac{M \times N + 6 \times (M-2) \times N}{8 \times [18 \times (M-2) \times (N-2) + 12 \times (M-2) + 32]} \tag{10}$$

onde a *taxa* $L2d_{mrd}$ é a taxa de *cache miss* de leitura na *cache* L2 no procedimento SOR. A comparação entre a taxa de *cache miss* teórica obtida e a experimental utilizando o software Valgrind será feita nos experimentos numéricos apresentados neste trabalho. Mas já é possível garantir, através da Eq. (10), que a taxa de *cache miss* de leitura na *cache* L2 é de aproximadamente 5±0,5%, para os problemas que acessam uma região de memória maior do que a *cache* L2.

3.4. Mais de um processo em um único processador

Como mencionado anteriormente, a estratégia de utilização mais eficiente da memória *cache* empregada neste trabalho baseia-se na decomposição de domínios em dois níveis, onde o primeiro nível de decomposição divide o problema entre os processadores, e o segundo nível dentro do mesmo processador. Neste trabalho a implementação desta estratégia é efetuada através da execução de mais de um processo MPI por CPU. A idéia deste método é transformar problemas maiores em problemas suficientemente pequenos para que a taxa de *cache misses* de leitura permaneça próxima de zero.

Se o algoritmo paralelo já tiver sido desenvolvido para ser executado distribuidamente em vários processadores, não será necessário alteração alguma no código fonte do simulador. Basta dividir o problema, não no número exato de processadores, mas em um número onde cada parte seja igual ou menor ao número de pontos nodais para o qual a taxa de *cache miss* é

aproximadamente zero. É importante garantir que o número de partes seja múltiplo do número de processadores para balanceamento de carga entre os processadores.

4. RESULTADOS COMPUTACIONAIS

Para verificar a eficiência da estratégia de utilização da memória *cache* sugerida neste trabalho, foi simulado o problema da cavidade com cobertura deslizante, Figura 1, que consiste de um recipiente quadrado com lados de tamanhos iguais a 1 metro, e preenchido com um fluido. A cobertura do recipiente se move com velocidade constante, causando movimentação do fluido. Condições de contorno de não deslizamento são impostas em todos os quatro segmentos das fronteiras, isto é, velocidade na fronteira superior igual à velocidade da cobertura do compartimento (u = 1,0 e v = 0,0) e velocidade nula nas outras três fronteiras (u = 0,0 e v = 0,0). Para o exemplo simulado o número de Reynolds é igual a 10000. O problema da cavidade representa um escoamento que parte de uma condição de repouso, sendo gradualmente acelerado até uma condição de regime permanente, isto é, a solução torna-se estacionária no tempo a partir de um determinado número de iterações. É importante enfatizar que as simulações executadas com apenas um processo são de fato, um pouco diferentes daquelas executadas com dois ou mais processos em um ou mais processadores, visto que um código verdadeiramente seqüencial foi utilizado no caso de um único processo em um processador. Além disso, essas simulações utilizam o algoritmo sem rotinas de comunicação.

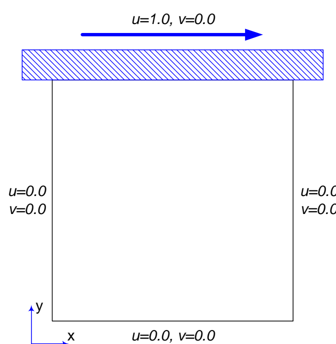

Figura 1: Representação do problema da cavidade com cobertura deslizante.

Os testes computacionais foram executados no cluster Enterprise do Laboratório de Computação de Alto Desempenho do Centro Tecnológico da Universidade Federal do Espírito Santo (LCAD – CT – UFES), um ambiente paralelo dedicado, composto por 64 nós de processamento e um host (máquina de gerenciamento). Todos utilizam o sistema operacional Linux Red Hat 7.1 (kernel 2.4.20) e interligados através de dois switches Fast Ethernet de 48 portas de 100Mb/s cada. Dentre as diversas ferramentas de programação disponíveis no Enterprise, foram utilizadas neste trabalho: o compilador da linguagem C gcc (GNU C Compiler, http://gcc.gnu.org), versão 2.96-112.7.1; a biblioteca (LAM-MPI, 2003) de comunicação entre os processadores; e o software Valgrind (Valgrind, 2004). O processador de cada nó é o AMD ATHLON XP 1800+. Todos os testes foram executados utilizando máquinas dedicadas, ou seja, não havia nenhum outro processo em execução nos processadores. E ainda, para garantir uma maior precisão dos resultados, todos os testes foram executados pelo menos duas vezes, com intuito de selecionar o resultado com menor tempo entre eles. Entretanto, não houve divergência significativa nos tempos de processamento em cada uma das repetidas execuções.

O método para obtenção de um maior desempenho pela melhor utilização da memória *cache* apresentado neste trabalho, é aplicado em situações adversas, tanto para casos onde se espera um melhor rendimento, tanto para situações onde o rendimento pode piorar pela utilização incorreta do método, quer dizer, quando não for respeitado a Eq. (4). Para avaliar os métodos de algoritmos paralelos, foram adotados alguns parâmetros de medida de desempenho dos algoritmos para processamento paralelo. Os métodos de avaliação adotados foram: *speedup* e eficiência paralela. O *speedup* é o número de vezes que um processo paralelo fica mais rápido em relação à velocidade de processamento utilizando apenas um processador. A eficiência paralela é a razão entre o *speedup* e o número de processadores, dessa forma, é possível conhecer o quanto dos recursos disponíveis foi utilizado (Minty *et al.*, 1999).

O primeiro experimento tem o objetivo de verificar, para o problema serial da cavidade, a variação da taxa de *cache miss* de leitura de dados em função do número de pontos nodais úteis da matriz, Figura 2. A quantidade de pontos nodais no problema testado está entre 400 e 7225. Pode-se observar que a taxa de *cache miss* começa a crescer após o tamanho de 64×64, ou seja, 4096 pontos nodais. Além disso, a *cache* L1 tem um número elevado de *cache misses* mesmo em problemas pequenos, devido ao seu tamanho reduzido em relação ao *cache* L2. Este gráfico foi construído a partir de dados gerados pelo software Valgrind.

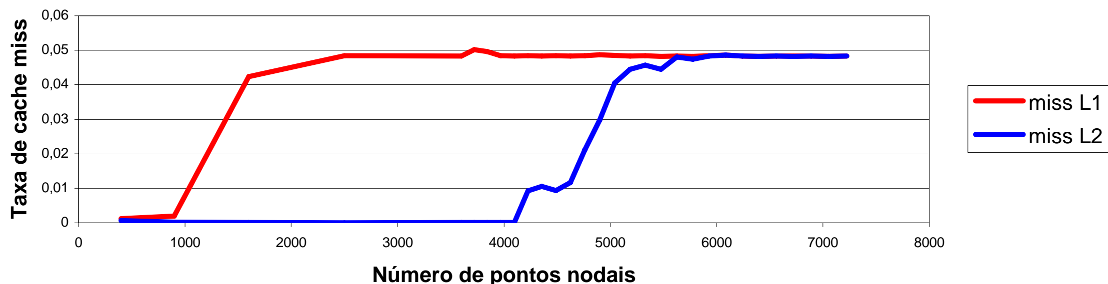

Figura 2: Variação da taxa de *cache miss* de leitura em função do número de pontos nodais.

Foi realizada uma análise para comparar as taxas de *cache miss* de leitura teóricas, Eq. (10), e as coletadas experimentalmente utilizando o software Valgrind (Figura 2). Tabela 1 mostra o número de leituras e o número de *cache misses* em L2 em função dos valores T e H, que representam o número de pontos nodais na largura e altura, respectivamente, excluindo os elementos das fronteias ($T = N$-2 e $H = M$-2), para as taxas teóricas e experimentais. É importante ressaltar que a taxa de *cache miss* teórica apresentada é para apenas uma iteração do procedimento SOR, enquanto que a taxa experimental é para uma simulação completa. Isso explica a pequena divergência de valores, mas confirma que o procedimento SOR é de grande relevância para todo o algoritmo. De acordo com a Figura 2 e a Tabela 1, independentemente do tamanho do problema (nos casos maiores onde ocorrem *cache misses* de leitura em L2), a taxa é de aproximadamente 5% de *cache misses* nas leituras pelo SOR. É importante notar que a taxa de *cache miss* não cresce além de 5% - este comportamento deve-se ao modo como é calculada a pressão em cada um dos pontos nodais da matriz.

O segundo experimento avalia a variação da taxa de *cache miss* em função do número de pontos nodais variando o número de processos em um único processador. A Figura 3 mostra como é possível "atrasar" o crescimento da taxa de *cache miss* em função do aumento do número de pontos nodais. Nesse gráfico, é apresentado o resultado da execução do algoritmo para malhas de 400 a 40000 pontos nodais, que formam um domínio que é dividido em apenas um processador. O domínio foi dividido de modo que o processador recebesse 1, 2, 3,

4 ou 5 subdomínios, sendo cada um tratado por um processo MPI (o MPI foi informado para dividir a tarefa em um número de processos igual a 1, 2, 3, 4 e 5). Com o uso de um simples parâmetro da linha de comando de execução dos experimentos é possível reduzir significativamente a taxa de *cache miss* em L2. É importante lembrar que os dados de origem para construção do gráfico foram gerados pelo software Valgrind que mede a taxa de *cache miss* sem levar em consideração a existência de um outro processo em execução. Considerando que esse método dispara mais de um processo em um único processador, existirão *cache misses* que não são contabilizados pelo Valgrind. Por exemplo, quando ocorrem as trocas de contexto entre os processos.

Tabela 1: Comparativo entre as taxas de *cache miss* teóricas e experimentais

			Teórico			Experimental
T	*H*	*T*×*H*	Número de leituras	Número de *cache misses* L2	Taxa de *cache miss* L2	Taxa de *cache miss* L2 (valgrind)
75	75	5625	102.182	5.072	4,96%	4,81%
76	76	5776	104.912	5.207	4,96%	4,75%
77	77	5929	107.678	5.342	4,96%	4,84%
78	78	6084	110.480	5.480	4,96%	4,86%
79	79	6241	113.318	5.619	4,96%	4,83%
80	80	6400	116.192	5.761	4,96%	4,82%
81	81	6561	119.102	5.903	4,96%	4,83%
82	82	6724	122.048	6.048	4,96%	4,83%
83	83	6889	125.030	6.194	4,95%	4,83%
84	84	7056	128.048	6.343	4,95%	4,82%
85	85	7225	131.102	6.492	4,95%	4,83%

Na Figura 3 onde a taxa de *cache miss* é aproximadamente zero nos exemplos com mais de um processo, essa taxa, na realidade, deveria ser maior que zero. A cada troca de contexto, o processo que retoma o domínio sobre o processador, não mais encontra seus dados na *cache* L2, ocorrendo assim, *cache misses*. As trocas de contexto não são tão agravantes, pois ocorrem, aproximadamente, a cada 10ms (OLIVEIRA; CARISSINI; TOSCANI, 2001). Nesse intervalo de tempo é possível executar até 33 iterações SOR em uma malha (subdomínio) que caiba na cache L2 (De Angeli, 2005). Essa quantidade de iterações é mais do que suficiente para executar as cinco iterações SOR entre os ciclos de comunicações adotadas neste trabalho. Nas situações onde um processo aguarda por informações provenientes de outros processos, a troca de contexto pode ocorrer antes de completar o tempo de 10ms, evitando assim, a ociosidade do processador.

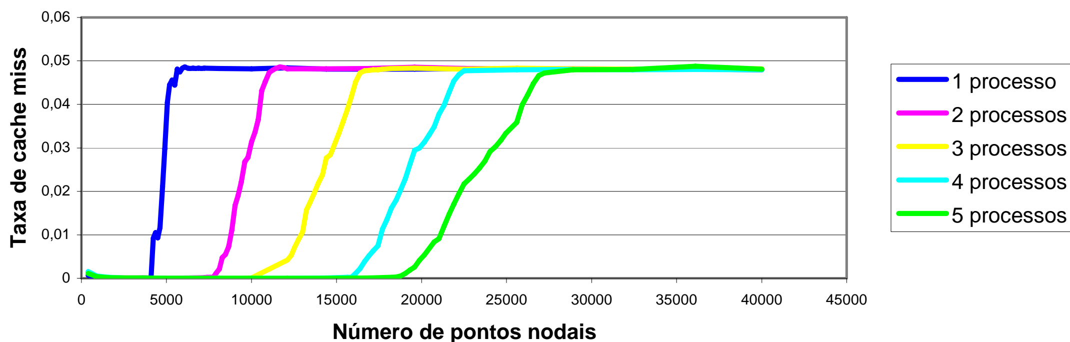

Figura 3: Taxa de *cache miss* em função do número de pontos nodais variando o número de processos em um único processador.

No próximo experimento são apresentadas as variações de tempo, da velocidade de processamento e do número de iterações SOR globais ocasionadas pelo uso de um ou mais processos computando subdomínios específicos em cada processador do cluster. Além disso, são apresentadas as taxas de ganho no tempo e na velocidade de processamento, além da taxa de aumento do número de iterações SOR global para o problema em diversos processadores. A Figura 4 mostra a diminuição da velocidade de processamento com o aumento do número de pontos nodais em uma máquina com um processador. No caso serial (1 processo), pode-se notar que o maior decréscimo ocorre após os 4096 pontos nodais úteis (64×64 pontos nodais, sem considerar os pontos nodais das fronteiras). Para os problemas maiores, a velocidade de processamento fica em torno de 125 MFLOP/s e a taxa de *cache miss* é de aproximadamente 5%, Figura 3.

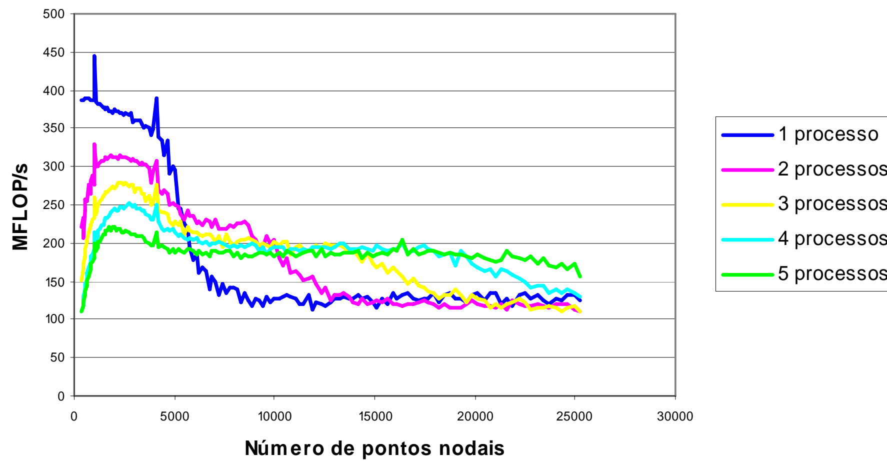

Figura 4: Número total de operações de ponto flutuante por segundo em função do número de pontos nodais

Nas matrizes menores que 64×64 pontos nodais, a velocidade de processamento diminui com o aumento do número de processos no mesmo processador. Isso ocorre devido ao fato de que, quanto maior o número de processos, maior é a quantidade total de comunicações MPI entre eles - o custo da comunicação reduz o tempo empregado em processamento "útil", reduzindo significativamente o número de FLOP/s obtido. Os exemplos com 2, 3, 4 e 5 processos executados no mesmo processador possuem velocidade de processamento maior que no caso serial (1 processo) para os tamanhos entre 5184 e 9216 pontos nodais. Isso ocorre por causa do excesso de *cache misses* do caso serial. Nessa faixa de tamanho do domínio as informações dos exemplos com dois até cinco processos são comportadas pela *cache* L2 (ou pelo menos a taxa de *cache miss* ainda é baixa). Pode-se observar que o exemplo com dois processos continua mais eficiente do que o com cinco, devido ao menor número de comunicação.

A partir de 9216 pontos nodais, as informações em cada uma das partes no exemplo com dois processos tornam-se maiores do que a *cache* L2, em conseqüência disso, sua velocidade de cálculo diminui a ponto de ficar pior que o caso serial. Isso se deve ao fato de que, mesmo com os dois exemplos tendo aproximadamente 5% de *cache misses*, o exemplo com dois processos inclui o custo da comunicação. A partir desse ponto, o exemplo com três processos tem a maior velocidade de processamento e isso se mantém até o tamanho de 12996 pontos nodais. Em seguida, o exemplo com quatro processos torna-se o mais eficiente até o tamanho de 18769 pontos nodais. Por fim, o exemplo com cinco processos é o único que ainda tem seus dados comportados pela *cache* L2 e, em conseqüência disso, é o que tem o maior número

de FLOP/s. Pode-se notar que o exemplo com cinco processos mantém um nível de FLOP/s praticamente constante em quase todos os tamanhos. Isto se deve ao fato de que, independentemente do número de pontos nodais testados, os dados acessados pelo procedimento SOR em cada um dos subdomínios são suportados pela *cache*, dessa forma, não há aumento da taxa de *cache miss*, o que implicaria na diminuição do nível de FLOP/s.

A Figura 5a mostra o tempo total para o processamento de cada um dos cinco exemplos anteriores. Pode-se notar que, mesmo nas matrizes maiores, onde o exemplo com cinco processos tem o maior nível de FLOP/s (de acordo com Figura 6a), o tempo de processamento é significativamente maior do que o serial. Isso ocorre porque um problema com um número maior de divisões de domínio requer um número maior de iterações globais SOR até a convergência. A propagação de informações de um subdomínio será atrasada na medida em que se aumenta o número de subdomínios na malha, uma vez que serão processadas cinco iterações SOR para cada ciclo de comunicação. Assim, mesmo um número maior de FLOP/s não garante um menor tempo de execução para os casos com maior número de processos. Pela Figura 5b, que apresenta o *speedup* pelo número de pontos nodais da malha dos cinco exemplos, é possível notar que o exemplo com cinco processos, em momento nenhum, é mais rápido que o serial. Essa degradação é causada principalmente pela grande quantidade de comunicação entre os processos, mesmo sendo no mesmo processador, e pelo retardo na convergência. Na Figura 6a, que apresenta a taxa de ganho de FLOP/s dos exemplos com mais de um processo no processador em relação ao exemplo puramente serial, é possível notar a queda da taxa máxima de ganho com o aumento do número de processos no processador. Essa redução no ganho se deve principalmente ao aumento do número de comunicações nos exemplos com um maior número de processos. A causa do retardo na convergência pode ser visualizada na Figura 6b que compara o total de iterações SOR em cada um dos cinco exemplos.

Uma forma de atenuar a diminuição do ganho utilizando uma quantidade grande de processos por processador é garantir que haja o menor número possível de comunicação entre processos em processadores distintos. A Figura 7 apresenta uma representação esquemática desta situação, indicando que mesmo com um número elevado de processos, o número de trocas de mensagens pelos processadores pode ser reduzido. Nos dois exemplos da Figura 7 existem 112 canais de comunicação entre os processos. No exemplo (a), somente 8 são entre processadores distintos, enquanto que no exemplo (b) são 64 canais de comunicação entre processadores distintos. Portanto, o baixo tráfego de rede pode fazer com que a utilização de vários processos por processador tenha uma grande quantidade de FLOP/s, uma vez que, iterações SOR são realizadas com dados na *cache*, e ainda, com um baixo custo de tráfego rede.

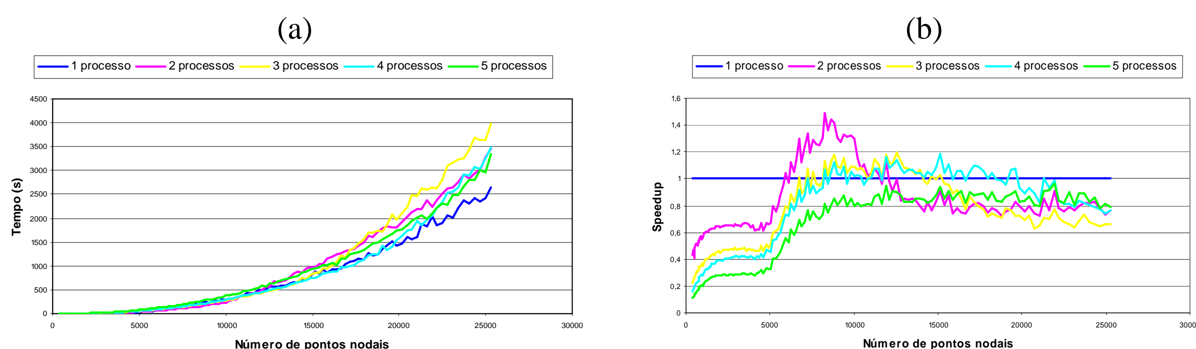

Figura 5: (a) Tempo de processamento e (b) *Speedup* em função do número de pontos nodais apresentados para um processo serial, e para processos paralelos executados em uma única máquina.

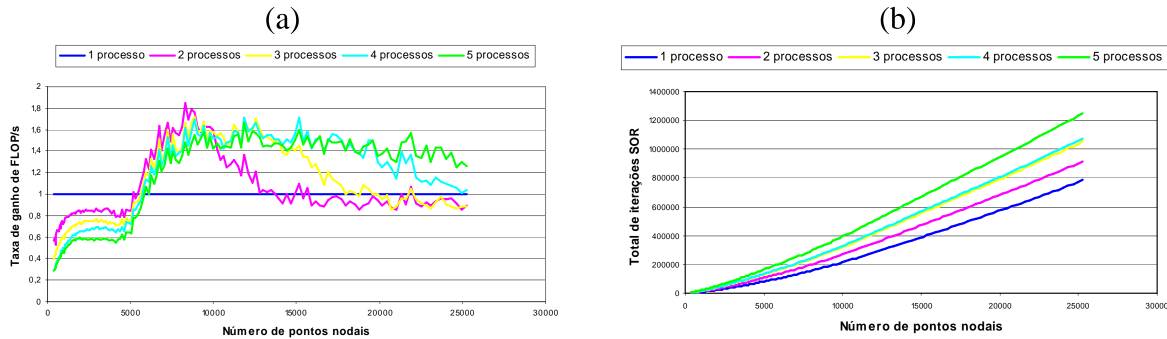

Figura 6: (a) Taxa de ganho na capacidade de cálculos (FLOP/s) em relação ao processo serial e (b) quantidade total de iterações SOR para executar simulações utilizando de 1 a 5 processos em um único processador.

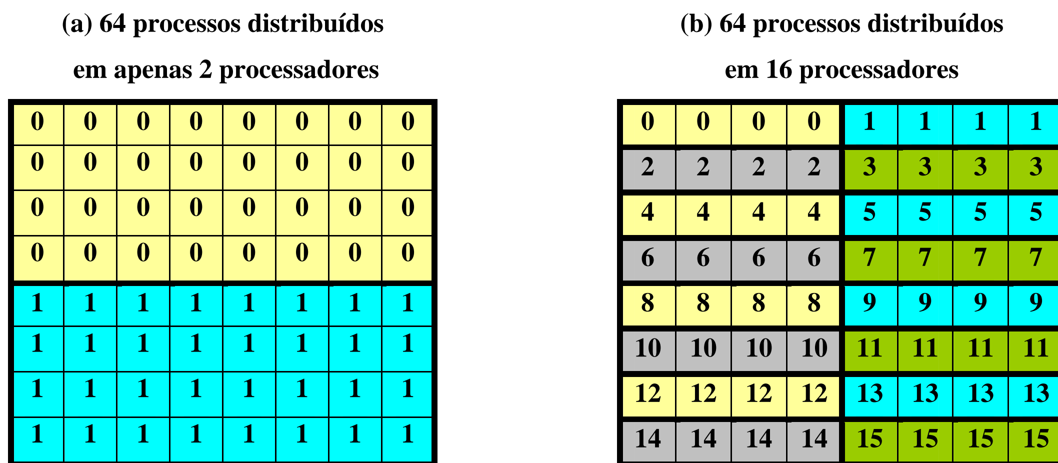

Figura 7: Representação esquemática das distribuições de 64 subdomínios entre os processadores, (a) utilizando 2 processadores, cada um com 32 processos e (b) utilizando 16 processadores com 4 processos cada.

A Figura 8 apresenta os benefícios de se aplicar a técnica de otimização de *cache*. Nesta avaliação, foi simulado o problema da cavidade formado por 256×256 e 512×512 pontos nodais, com a técnica de otimização de uso da *cache*. As duas configurações de tamanhos da malha foram executadas de forma normal, ou seja, um único processo por processador, e de forma otimizada, com um número de processos por processador escolhido conforme apresentado neste trabalho. O menor número de processos (totais) para uma boa utilização da *cache* é de 16 e 64, respectivamente para as malhas 256×256 e 512×512. Lembrando que esses números podem ser ligeiramente maiores para que o número de processos seja múltiplo do número de processadores. A Tabela 2 apresenta a quantidade de processos por processador utilizados na forma otimizada. A Figura 8a apresenta os tempos de execução das simulações, enquanto a Figura 8b apresenta o *speedup* e a Figura 8c a eficiência paralela obtidos. É importante notar que, o *speedup* (Figura 8b) e a eficiência paralela (Figura 8c), tanto para a forma normal, quanto para a forma otimizada, foram baseadas no tempo de execução da simulação serial (um processo em um processador).

A execução otimizada da malha 256×256 foi realizada com até nove de processadores, pois, a partir de 16 processadores, cada processo já é suportado pela *cache*. Então, não é necessário mais de um processo por processador. Isso pode ser comprovado na Figura 8c (malha 256×256) que, com 16 processadores a eficiência paralela aumentou para 96% comparado com os 60% alcançados com 9 processadores. Além dos fatores que impedem a

melhoria da aplicação da técnica de otimização da *cache* já mencionados anteriormente, como por exemplo: a desproporcionalidade entre os números de pontos nodais na “altura” e “largura” dos blocos de processamento e aumento do tráfego de rede, existe também o excesso de divisões do domínio de forma a garantir um múltiplo do número de processadores, que em certas situações é muito acima do mínimo exigido. Justamente os exemplos da malha 512×512 com 25, 30, 49 e 56 processadores, que possuem os maiores número de processos (75, 90, 98 e 112 respectivamente), são os que apresentaram os menores benefícios, ou até mesmo prejuízo, como no caso do exemplo com 56 processadores. Os melhores benéficos, pela aplicação da técnica de utilização de mais de um processo por processador, apresentados nesse experimento foram: de 48% de aumento na velocidade de processamento da simulação utilizando a malha de 256×256 e 9 processadores, e de 40% para a malha de 512×512 com 36 processadores.

Tabela 2: Quantidade de processos por processador em cada um dos testes para avaliação dos benefícios da forma otimizada de utilização da *cache*.

256×256 pontos nodais			512×512 pontos nodais		
Processadores	Processos por processador	Total de processos	Processadores	Processos por processador	Total de processos
2	8	16	2	32	64
4	4	16	4	16	64
9	2	18	9	8	72
16	1	16	16	4	64
25	1	25	25	3	75
30	1	30	30	3	90
36	1	36	36	2	72
49	1	49	49	2	98
56	1	56	56	2	112

5. CONCLUSÃO

Uma técnica de utilização eficiente da memória *cache* para proporcionar redução no tempo total de processamento foi investigada neste trabalho. Um algoritmo de decomposição de domínio foi empregado. A decomposição foi efetuada em dois níveis, onde o primeiro nível de decomposição divide o problema entre os processadores, e o segundo nível dentro do mesmo processador. Neste trabalho a implementação desta estratégia foi efetuada através da execução de mais de um processo MPI por processador. A técnica baseia-se na divisão das estruturas de dados (matrizes) em blocos suficientemente pequenos para caberem na memória *cache* do processador, favorecendo a reutilização de informações que estão armazenadas em *cache* e garantindo uma disponibilidade mais rápida dos dados ao processador. A idéia principal desta técnica consiste em definir o número de processos que cada processador deve executar com base nos requisitos de memória e tamanho da memória *cache* do processador, não havendo necessidade de nenhuma alteração significativa no código paralelo.

Nos experimentos foi observado que o aumento do número de processos em cada processador pode proporcionar uma redução significativa no tempo de execução. Para garantir um bom rendimento é necessária a escolha do número ideal de processos por processador, tomando como base o tamanho do problema simulado, o algoritmo, o número de processadores e a configuração das máquinas. Da mesma forma que a técnica reduz o tempo total de processamento, em algumas situações ela pode degradar consideravelmente o desempenho obtido. É possível concluir que a divisão em subdomínios tende a degradar o desempenho quando criados muitos processos em cada processador ou quando são produzidos

blocos com uma razão de aspecto ruim, isto é, que implicam em longas interfaces de comunicação entre processadores. Notou-se que os maiores responsáveis pela não obtenção de eficiência quando se divide o problema em vários processos são: o número maior de iterações SOR para atingir a convergência e o aumento do tráfego de rede.

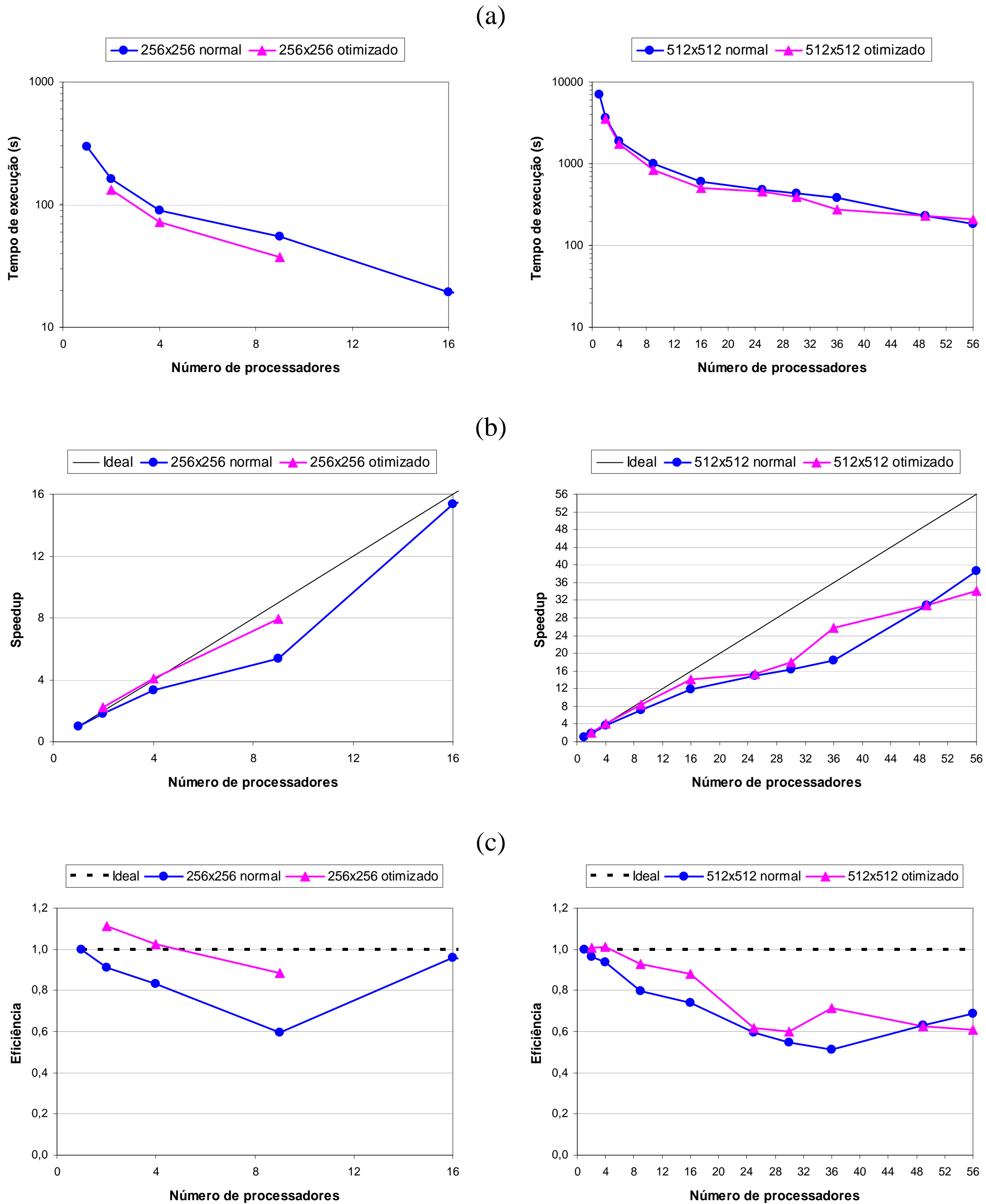

Figura 8: (a) Tempo de execução, (b) *Speedup* e (c) eficiência paralela obtida nos problemas com 256×256 e 512×512 pontos nodais, utilizando o método de árvore-binária de comunicação e freqüência de comunicação igual a cinco. Foram executados de forma convencional (um processo por processador) e otimizado (mais de um processo por processador).

Agradecimentos

Este trabalho foi realizado com o apoio parcial da CAPES, dentro do Programa de Cooperação Internacional CAPES/Universidade do Texas em Austin.

REFERÊNCIAS

Anderson Jr.; J. D., 1995, Computational Fluid Dynamics: The Basics with Applications, *McGraw Hill Inc*.

De Angeli, J. P., Valli, A. M. P., De Souza, A. F., Reis Jr, N. C., 2003, Numerical Simulations of the Navier-Stokes Equations Using Clusters of Workstations. *Computer Architecture and High Performance Computing, São Paulo*.

De Angeli, J. P., 2005, Implementação de um algoritmo de Mecânica dos Fluidos Computacional projetado para plataformas de processamento paralelo com memória distribuída, *Dissertação de Mestrado em Informática, UFES.*

Douglas, C. C. *et al*., 2000, Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transations on Numerical Analysis. Kent State University*, v. 10, p. 21-40.

Griebel, M.; Dornseifer, T.; Neunhoeffer, T., 1998, Numerical Simulation in Fluid Dynamics: A Pratical Introduction. *Philadelphia: SIAM*.

Gullerud, A. S., Dodds Jr., R. H., fev.2001, MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit, 3-D finite element analysis. *Computers & Structures*, v. 79, n. 5, p. 553-575.

Kowarschik, M. *et al*., 2000, Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing 64*, p. 381-399.

LAM-MPI, 2003, Uma implementação do padrão Message Passing Interface (MPI). Versão 6.5.9., *LAM*, http://www.lam-mpi.org. Plataforma Linux.

Minty, E., DAVEY, R., SIMPSON, A., HENTY, D., 1999, Decomposing the Potentially Parallel: A one day course. Course Notes. Versão 2.0. *Edinburgh Parallel Computing Centre. The University of Edinburgh*.

OLIVEIRA, R. S., CARISSINI, A. S., TOSCANI, S. S., 2001, Sistemas Operacionais. 2ª Edição. *Porto Alegre, Sagra-Luzzato*, ISBN: 85-241-0643-3.

Silva, M., 2003, Cache-Aware Data Laying for the Gauss-Seidel Smoother. *Electronic Transactions on Numerical Analysis. Kent State University*, v. 15, p. 66-77.

Simoes, S. N., 2004, Uma Comparação Entre um Algoritmo Síncrono e um Parcialmente Assíncrono Para Solução de Problemas de Mecânica dos Fluidos Computacional. *Dissertação de Mestrado. Universidade Federal do Espírito Santo*.

Streng, M., 1996, Load Balancing for Computacional Fluid Dynamics Calculations. *High Performance Computing Fluid Dynamics, ed. P. Wesseling. Kluwer Academic Publishers*.

Takahashi, D., mai.2003, Efficient implementation of parallel three-dimensional FFT on clusters of PCs. *Computer Physics Communications*, v. 152, n. 2, p. 144-150.

Tomko, K. A., Abraham, S. G., 1994, Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors. *International Conference on Supercomputing*.

Tseng, C. W., 2000, Sofware Support for Improving Locality in Advanced Scientific Codes. *Departament of Computer Science, University of Maryland*.

Valgrind, 2004, GPL'd system for debugging and profiling x86-Linux programs. Versão 2.2.0., *GNU*, http://valgrind.kde.org. Plataforma Linux.