UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
MESTRADO EM INFORMÁTICA

JOÃO PAULO DE ANGELI

Implementação de um algoritmo de Mecânica dos Fluidos Computacional projetado para plataformas de processamento paralelo com memória distribuída

VITÓRIA
2005

JOÃO PAULO DE ANGELI

Implementação de um algoritmo de Mecânica dos Fluidos Computacional projetado para plataformas de processamento paralelo com memória distribuída

Dissertação apresentada ao Mestrado de Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Mestre em Informática, na área de mecânica dos fluidos computacional utilizando processamento paralelo e distribuído e arquitetura de computadores.

Orientador: Prof. Dr. Neyval Costa Reis Jr.

Co-orientadores: Profª. Drª. Andréa Maria Pedrosa Valli e Prof. Dr. Alberto Ferreira De Souza.

VITÓRIA
2005

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

De Angeli, João Paulo, 1979-

D284i Implementação de um algoritmo de mecânica dos fluidos computacional projetado para plataformas de processamento paralelo com memória distribuída/ João Paulo De Angeli. – 2005.

87 f. : il.

Orientador: Neyval Costa Reis Júnior.

Co-Orientadores: Andréa Maria Pedrosa Valli, Alberto Ferreira de Souza.

Dissertação (mestrado) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Processamento paralelo (Computadores). 2. Diferenças finitas. 3. Navier-Stokes, Equações de. 4. Memória cache.I. Reis Júnior, Neyval Costa. II. Valli, Andréa Maria Pedrosa. III. Souza, Alberto Ferreira de. IV. Universidade Federal do Espírito Santo. Centro Tecnológico. V. Título.

CDU: 004

JOÃO PAULO DE ANGELI

Implementação de um algoritmo de Mecânica dos Fluidos Computacional projetado para plataformas de processamento paralelo com memória distribuída

Dissertação apresentada ao Mestrado de Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Mestre em Informática, na área de mecânica dos fluidos computacional utilizando processamento paralelo e distribuído e arquitetura de computadores.

Aprovada em 30 de junho de 2005.

COMISSÃO EXAMINADORA

Prof. Dr. Neyval Costa Reis Jr.
Universidade Federal do Espírito Santo
Orientador

Profª. Drª. Andréa M. Pedrosa Valli
Universidade Federal do Espírito Santo
Co-orientadora

Prof. Dr. Aberto Ferreira de Souza
Universidade Federal do Espírito Santo
Co-orientador

Prof. Dr. Cláudio L. Amorim
Universidade Federal do Rio de Janeiro

Prof. Dr. Elias Silva de Oliveira
Universidade Federal do Espírito Santo

Aos meus pais pela vida.

A minha namorada Symone pelo apoio e carinho.

AGRADECIMENTOS

A Deus sobre todas as coisas.

Ao meu orientador Neyval pelos seus ensinamentos.

Aos meus co-orientadores Alberto e Andréa.

Aos meus amigos do LCAD, que batalharam juntamente comigo.

E a todos aqueles que de uma forma ou de outra contribuíram para meu progresso.

“Não há tarefa mais pesada que a ociosidade.”
Goethe, escritor alemão – 1749-1836.

RESUMO

Discute a implementação do algoritmo numérico para simulação de escoamento 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*. O algoritmo de solução para as equações de Navier-Stokes utiliza um esquema explicito para pressão e um esquema implícito para as velocidades. A implementação paralela é baseada na decomposição do domínio, onde o domínio computacional do problema é decomposto em vários blocos, sendo um ou mais destinados a nós de processamento distintos. Todos os nós então processam em paralelo as tarefas de computação sobre os blocos a eles designados. O processamento paralelo inclui inicialização, cálculo de coeficientes, solução linear nos subdomínios, e comunicação entre os nós. A troca de informação entre os processos referentes a cada subdomínio é realizada utilizando a biblioteca *message passing interface* (MPI), o que assegura portabilidade entre diferentes plataformas computacionais, abrangendo desde *máquinas maciçamente paralelas* (MPP) até *clusters de estações de trabalho*. Para melhorar os níveis de desempenho obtidos pelo algoritmo, foram investigadas técnicas para a redução do volume de comunicação entre processadores e utilização mais eficiente da memória *cache* dos microprocessadores. Para avaliar o desempenho do algoritmo desenvolvido e analisar as diferentes estratégias de paralelização foram executadas simulações com cluster de 2 a 56 processadores, nas quais foram avaliados o tempo de execução, *speedup* e eficiência paralela. Os resultados experimentais mostram que as otimizações relacionadas aos fatores de comunicação melhoram o *speedup* em até 165%, e a técnica de utilização mais eficiente da memória *cache* pode melhorar o *speedup* em mais 40% acima da otimização da comunicação.

Palavras-chave: Processamento Paralelo. Diferenças Finitas. Navier-Stokes. MPI. Memória *Cache*.

ABSTRACT

This work discusses the implementation of a numerical algorithm for simulating incompressible fluid flows, based on the finite difference method, and designed for parallel computing platforms with distributed-memory, particularly for clusters of workstations. The solution algorithm for the Navier-Stokes equations utilizes an explicit scheme for pressure and an implicit scheme for velocities. The parallel implementation is based on domain decomposition, where the original calculation domain is decomposed into several blocks, each of which given to a separate processing node. All nodes then execute computations in parallel, each node on its associated sub-domain. The parallel computations include initialization, coefficient generation, linear solution on the sub-domain, and inter-node communication. The exchange of information across the sub-domains, or processors, is achieved using the message passing interface standard, MPI. The use of MPI ensures portability across different computing platforms ranging from massively parallel machines to clusters of workstations. Three different optimization strategies were evaluated in order to improve the computational performance of the algorithm, which include techniques exploring a reduction in the communication volume between processors and a more efficient utilization of the microprocessor's cache memory. In order to evaluate the performance levels obtained, and to analyze the effectiveness of the optimization strategies adopted, simulations using a 64 nodes cluster were executed. The simulations were performed using 2 to 56 processors, where execution time and speed-up were measured. The results indicate that the optimizations related to communication factors can improve the speed-up obtained up to 165%, while the cache memory optimization technique used can improve the speed-up obtained in further 40%.

Keywords: Parallel Processing. Finite Difference Method. Navier-Stokes. MPI. Cache Memory.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

SUMÁRIO

1 Introdução

Nas últimas décadas as técnicas de Mecânica dos Fluidos Computacional (MFC) têm se tornando ferramentas bastante importantes no âmbito de projeto, otimização, pesquisa e desenvolvimento de aplicações industriais e científicas. Todavia, apesar do aumento da velocidade de processamento dos modernos computadores atuais, existem aplicações práticas que transcendem a capacidade computacional atualmente instalada nos maiores centros de processamento do mundo. Problemas como a previsão do clima em escala global, escoamentos em reservatórios de petróleo em escala real e a simulação completa das estruturas turbulentas em escoamentos são exemplos de aplicações que ainda representam desafio para a comunidade científica.

O rápido crescimento do desempenho dos microprocessadores e das redes de interconexão tem permitido a implementação de *clusters de estações de trabalho* com grande poder de processamento por uma pequena fração do preço dos supercomputadores. No entanto, o uso de *clusters de estações de trabalho* requer um diferente paradigma de programação, uma vez que a arquitetura do sistema é baseada no modelo de memória distribuída, que difere consideravelmente do modelo de memória compartilhada, largamente utilizado nas últimas décadas.

A utilização de *clusters* pode ser uma excelente solução, uma vez que a velocidade de processamento de um microprocessador pode ultrapassar os 4 GFLOP/s (10^9 operações de ponto flutuante por segundo), e utilizando *cluster*, a velocidade de processamento teórica é multiplicada pelo número de nós. Entretanto, a maioria das aplicações MFC desenvolvidas para *clusters* obtém níveis de desempenho muito abaixo do desempenho máximo desses sistemas (DOUGLAS *et al*., 2000).

1.1 Motivação

Um dos principais fatores que contribuem para a diferença entre o desempenho real e o desempenho máximo teórico de *clusters* é a baixa velocidade de comunicação entre nós de processamento durante as tarefas de computação. Assim a eficiência da rede de interconexão

é um dos principais "gargalos" para a computação, sendo a comunicação entre os nós de processamento do sistema a chave para a obtenção de bom desempenho.

Outro fator extremamente significativo no desempenho obtido com *clusters* é a grande diferença entre a velocidade de processamento dos microprocessadores e a capacidade de transferência das memórias em máquinas microprocessadas atuais; na verdade, a evolução das memórias não tem acompanhado a dos microprocessadores. A utilização eficiente da hierarquia de memória dos sistemas computacionais pode aumentar a velocidade de processamento em até uma ordem de grandeza (KOWARSCHIK, 2000).

Os dois fatores que afetam o desempenho de *clusters* mencionados nos motivaram a desenvolver novos algoritmos para MFC na direção do uso eficiente dos recursos para comunicação entre processadores e da hierarquia de memória.

1.2 Objetivos

O principal objetivo deste trabalho de pesquisa foi o desenvolvimento de um algoritmo de MFC, desenvolvido especialmente para *clusters de estações de trabalho*, que explora técnicas para minimizar os problemas relacionados aos "gargalos" de comunicação entre processadores e técnicas de uso eficiente da hierarquia de memória. O algoritmo desenvolvido é baseado na discretização das equações de Navier-Stokes através do método das diferenças finitas, conforme a formulação proposta por Griebel, Dornseifer e Neunhoeffer (1998). Esse método foi fortemente influenciado pela técnica "marker-and-cell" (MAC) de Harlow e Welch (1965) e 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. Mesmo com sua simplicidade, esse método é surpreendentemente flexível e relativamente eficiente, podendo ser aplicado em uma variedade de problemas transiente com domínios de fronteiras fixas e livres.

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). O uso de MPI assegura portabilidade através das diferentes plataformas computacionais, abrangendo desde máquinas massivamente paralelas até *clusters de estações de trabalho*.

Para melhorar os níveis de desempenho obtidos pelo algoritmo, foram investigadas técnicas para a redução do volume de comunicação entre processadores e utilização mais eficiente da memória *cache* dos microprocessadores.

1.2.1 Uso Eficiente dos Recursos de Comunicação

Com o objetivo de reduzir o volume de comunicação entre processadores a freqüência a qual a comunicação entre os processadores deveria ocorrer para cada iteração SOR foi investigada.

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, o que, de acordo com experimentos, reduz a velocidade do processamento. É utilizado um procedimento alternativo, que executa algumas iterações SOR entre cada comunicação. Isso reduz a quantidade de comunicação sem diminuir a velocidade de convergência. Além da freqüência de comunicação entre processadores, os passos de comunicação foram realizados de três diferentes formas: comunicação todos-para-todos, mestre-escravo e árvore binária.

1.2.2 Uso Eficiente da Hierarquia de Memória

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 esta é significativamente mais lenta que a memória *cache*.

1.3 Contribuições

Para avaliar o desempenho do algoritmo desenvolvido e analisar as diferentes estratégias de paralelização foram executadas simulações com *cluster* de 2 a 56 processadores e avaliados o tempo de execução, *speedup* e eficiência paralela. Os resultados experimentais mostram que as otimizações relacionadas aos fatores de comunicação melhoram o *speedup* em até 165%, e a técnica de utilização mais eficiente da memória *cache* pode melhorar o *speedup* em mais 40% acima da otimização da comunicação.

1.4 Organização da Dissertação

Essa dissertação é composta por seis capítulos. Após essa introdução, o Capítulo 2 apresenta o método numérico implementado pelo algoritmo paralelo desenvolvido, além das equações governantes dos problemas MFC que o algoritmo soluciona e as estratégias de paralelização empregadas no algoritmo.

O Capítulo 3 descreve a metodologia utilizada nos experimentos empregados na validação e avaliação do algoritmo paralelo desenvolvido, descrevendo os problemas simulados, os modos de comunicação paralela entre os processos, e o detalhamento dos testes que serão aplicados para avaliação do método sugerido neste trabalho. Além de detalhar a configuração dos equipamentos e as instrumentações que foram utilizados nas simulações dos problemas.

O Capítulo 4 apresenta os resultados experimentais e sua discussão. Inicialmente, são apresentadas comparações entre os resultados obtidos pelo algoritmo e dados experimentais, para avaliar a consistência e acurácia do algoritmo desenvolvido. Em seqüência, são analisados o tempo de processamento, a quantidade de FLOP/s (operações de ponto flutuante por segundo), o *speedup*, a eficiência paralela e a quantidade de *cache misses*. Estes testes são efetuados com a variação: do número de máquinas utilizadas; do número de processos espalhados pelos processadores; do tamanho do problema; das formas de comunicação entre os processadores; e de algumas das constantes para solução do problema. O Capítulo 5 apresenta as conclusões obtidas e propõe alguns trabalhos futuros. As referências são encontradas no Capítulo 6.

2 Algoritmo Computacional

Este capítulo apresenta uma breve descrição do método numérico utilizado e do algoritmo de solução utilizado, detalhando a estratégia de paralelização empregada para a divisão da carga de trabalho entre os processadores.

2.1 Equações governantes

O algoritmo é desenvolvido para simular escoamentos bidimensionais transientes de fluidos viscosos incompressíveis. Esta classe de problemas é governada pelas equações de conservação de massa e quantidade de movimento para fluidos Newtonianos, considerando viscosidade e densidade constantes, podem ser escritas como:

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

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

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

onde u e v são os componentes horizontal e vertical da velocidade, respectivamente, p é a pressão, g_x e g_y 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. As Equações (1) e (2) são as conhecidas equações de Navier-Stokes (equações de conservação de quantidade de movimento) e a Equação (3) da continuidade (equação que representa o principio de conservação de massa).

2.2 Método Numérico

O algoritmo desenvolvido é baseado no método das diferenças finitas, conforme a formulação proposta por Griebel, Dornseifer e Neunhoeffer (1998). O domínio é discretizado em i_{max} pontos nodais na direção x e j_{max} pontos nodais na direção y. A região é discretizada usando uma matriz deslocada, na qual a pressão p é localizada no centro de cada elemento, a velocidade horizontal u no ponto médio da face vertical do elemento, e a velocidade vertical v no ponto médio da face horizontal do elemento (Figura 2.1). Esse arranjo das incógnitas previne possíveis oscilações na pressão, que poderiam ocorrer se fossem considerados os valores u, v e p no mesmo ponto.

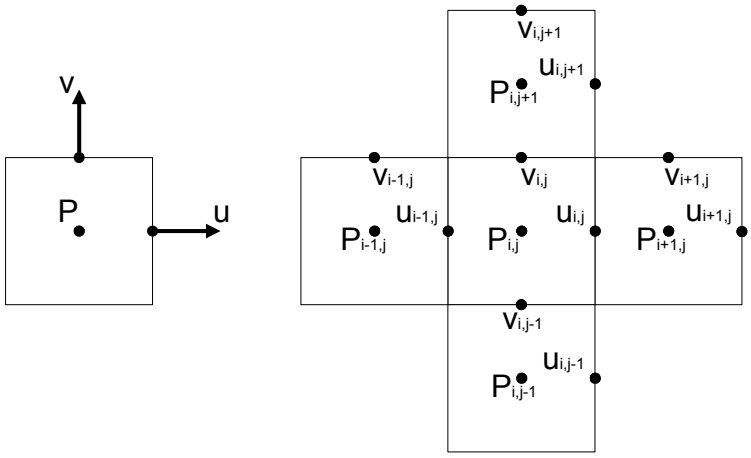


Figura 2.1: Representação esquemática do balanceamento da malha.

Para obter a discretização do tempo das Equações (1) e (2) de momento, foram discretizadas as derivadas do tempo $\partial u/\partial t$ e $\partial v/\partial t$ usando o método de Euler. Introduzindo as funções F e G:

$$F^{(n)} = u^{(n)} + \delta t \left[\frac{1}{\sqrt{\mathrm{Re}}} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial \left(u^2\right)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \right]^{(n)} \tag{4}$$

$$G^{(n)} = v^{(n)} + \delta t \left[\frac{1}{\sqrt{\mathrm{Re}}} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial \left(v^2\right)}{\partial y} + g_y \right]^{(n)} \tag{5}$$

onde o sobrescrito n denota o nível do tempo. A seguir, as equações de momento (Equações (1) e (2)) são reescritas em forma discretizada, utilizando diferenças centradas, como:

$$u_{i,j}^{(n+1)} = F_{i,j}^{(n)} - \frac{\delta t}{\delta x}\left(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}\right)$$

$$i = 1,...,i_{\max} - 1 \qquad j = 1,...,j_{\max} \tag{6}$$

$$v_{i,j}^{(n+1)} = G_{i,j}^{(n)} - \frac{\delta t}{\delta y}\left(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}\right)$$

$$i = 1,...,i_{\max} \qquad j = 1,...,j_{\max} - 1 \tag{7}$$

na qual pode ser caracterizado como sendo explícito nas velocidades e implícito na pressão; ou seja, o campo de velocidade no passo de tempo t_{n+1} pode ser calculado uma vez que a pressão correspondente é conhecida.

Substituindo as Equações (6) e (7) para o campo de velocidade na Equação (3), resulta a Equação de Poisson para a pressão $p^{(n+1)}$ no tempo t_{n+1}:

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = \frac{1}{\delta t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y}\right)$$

$$i = 1,...,i_{\max} \qquad j = 1,...,j_{\max} \tag{8}$$

na qual requer valores de contornos para a pressão. Foi assumido que $p_{0,j} = p_{1,j}$, $p_{imax+1,j} = p_{imax,j}$, $p_{i,0} = p_{i,1}$, $p_{i,jmax+1} = p_{i,jmax}$, com $i = 1,\ldots, i_{max}$ e $j = 1,\ldots, j_{max}$. Além disso, são necessários os valores de F e G nos contornos para calcular o lado direito da Equação (8). Foi definido $F_{0,j} = u_{0,j}$, $F_{imax,j} = u_{imax,j}$, $G_{i,0} = v_{i,0}$ e $G_{i,jmax} = v_{i,jmax}$, com $i = 1,\ldots, i_{max}$ e $j = 1,\ldots, j_{max}$.

Com o propósito de simplificação, a Equação (8) pode ser escrita como:

$$Ap_{j,i}P_{j,i} = Aw_{j,i}P_{j,i-1} + Ae_{j,i}P_{j,i+1} + As_{j,i}P_{j-1,i} + An_{j,i}P_{j+1,i} - B_{j,i}$$

$$i = 1,...,i_{\max} \qquad j = 1,...,j_{\max} \tag{9}$$

onde:

$$Ae_{j,i} = Aw_{j,i} = \frac{1}{(\delta x)^2}, \quad An_{j,i} = As_{j,i} = \frac{1}{(\delta y)^2} \tag{10}$$

$$Ap_{j,i} = Ae_{j,i} + Aw_{j,i} + An_{j,i} + As_{j,i} \tag{11}$$

$$B_{i,j-1} = \frac{1}{\delta t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y}\right) \tag{12}$$

A discretização das derivadas espaciais nas funções *F* e *G* (Equações (4) e (5)) requer uma mistura entre um esquema de diferenças centrais e discretização *donor-cell* para manter estabilidade para problemas fortemente convectivos. Este tratamento é necessário devido à existência dos termos convectivos $\partial(u^2)/\partial x$, $\partial(uv)/\partial y$, $\partial(uv)/\partial x$ e $\partial(v^2)/\partial y$ nas equações de momento, que se tornam dominantes nos números de Reynolds ou velocidades elevadas. Assim, problemas de estabilidade podem ocorrer quando se escolhe um espaçamento muito grande entre os pontos nodais. Para evitar problemas de estabilidade, esses termos convectivos são tratados usando uma média ponderada entre o esquema de diferença central e o esquema *donor-cell* como sugerido por Hirt, Nicolas e Romero (1975).

As derivadas espaciais de primeira ordem $\partial u/\partial x$, $\partial v/\partial y$ e as derivadas de segunda ordem $\partial^2 u/\partial x^2$, $\partial^2 u/\partial y^2$, $\partial^2 v/\partial x^2$ e $\partial^2 v/\partial y^2$, que formam os chamados termos difusivos, podem ser tratadas através do esquema de diferenças centrais.

Assim, as expressões discretizadas para os termos *F* e *G* podem ser escritas como:

$$\left[\frac{\partial\left(u^2\right)}{\partial x}\right]_{i,j} := \frac{1}{\delta x}\left(\left(\frac{u_{i,j}+u_{i+1,j}}{2}\right)^2-\left(\frac{u_{i-1,j}+u_{i,j}}{2}\right)^2\right)+ + \gamma\frac{1}{\delta x}\left(\frac{\left|u_{i,j}+u_{i+1,j}\right|}{2}\frac{\left(u_{i,j}-u_{i+1,j}\right)}{2}-\frac{\left|u_{i-1,j}+u_{i,j}\right|}{2}\frac{\left(u_{i-1,j}-u_{i,j}\right)}{2}\right) \tag{13}$$

$$\left[\frac{\partial(uv)}{\partial y}\right]_{i,j} := \frac{1}{\delta y}\left(\frac{\left(v_{i,j}+v_{i+1,j}\right)}{2}\frac{\left(u_{i,j}+u_{i,j+1}\right)}{2}-\frac{\left(v_{i,j-1}+v_{i+1,j-1}\right)}{2}\frac{\left(u_{i,j-1}+u_{i,j}\right)}{2}\right)+ + \gamma\frac{1}{\delta y}\left(\frac{\left|v_{i,j}+v_{i+1,j}\right|}{2}\frac{\left(u_{i,j}-u_{i,j+1}\right)}{2}-\frac{\left|v_{i-1,j}+v_{i+1,j-1}\right|}{2}\frac{\left(u_{i,j-1}-u_{i,j}\right)}{2}\right) \tag{14}$$

$$\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} := \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\delta x)^2} \tag{15}$$

$$\left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j} := \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\delta y)^2} \tag{16}$$

$$\left[\frac{\partial p}{\partial x}\right]_{i,j} := \frac{p_{i+1,j}-p_{i,j}}{\delta x} \tag{17}$$

$$\left[\frac{\partial(uv)}{\partial x}\right]_{i,j} := \frac{1}{\delta x}\left(\frac{\left(u_{i,j}+u_{i+1,j}\right)}{2}\frac{\left(v_{i,j}+v_{i+1,j}\right)}{2} - \frac{\left(u_{i-1,j}+u_{i-1,j+1}\right)}{2}\frac{\left(v_{i-1,j}+v_{i,j}\right)}{2}\right) + \\ + \gamma\frac{1}{\delta x}\left(\frac{\left|u_{i,j}+u_{i,j+1}\right|}{2}\frac{\left(v_{i,j}-u_{i+1,j}\right)}{2} - \frac{\left|u_{i-1,j}+u_{i-1,j+1}\right|}{2}\frac{\left(v_{i-1,j}-v_{i,j}\right)}{2}\right) \tag{18}$$

$$\left[\frac{\partial\left(v^2\right)}{\partial y}\right]_{i,j} := \frac{1}{\delta y}\left(\left(\frac{v_{i,j}+v_{i,j+1}}{2}\right)^2 - \left(\frac{v_{i,j-1}+v_{i,j}}{2}\right)^2\right) + \\ + \gamma\frac{1}{\delta y}\left(\frac{\left|v_{i,j}+v_{i,j+1}\right|}{2}\frac{\left(v_{i,j}-v_{i,j+1}\right)}{2} - \frac{\left|v_{i,j-1}+v_{i,j}\right|}{2}\frac{\left(v_{1,j-1}-v_{i,j}\right)}{2}\right) \tag{19}$$

$$\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} := \frac{v_{i+1,j}-2v_{i,j}+v_{i-1,j}}{(\delta x)^2} \tag{20}$$

$$\left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} := \frac{v_{i,j+1}-2v_{i,j}+v_{i,j-1}}{(\delta y)^2} \tag{21}$$

$$\left[\frac{\partial p}{\partial y}\right]_{i,j} := \frac{p_{i,j+1}-p_{i,j}}{\delta x} \tag{22}$$

onde δx e δy são os tamanhos dos espaçamentos entre as linhas da malha, em outras palavras, são as distâncias entres os centros dos pontos nodais, na horizontal e vertical respectivamente. O parâmetro γ está entre 0 e 1. Para $\gamma = 0$ é obtido a discretização em diferença central, e para $\gamma = 1$, é obtido o resultado no esquema *donor-cell*. De acordo com Hirt *et al*. (1975), γ deve ser escolhido tal que:

$$\gamma \geq \max_{i,j}\left(\left|\frac{u_{i,j}\delta t}{\delta x}\right|, \left|\frac{v_{i,j}\delta t}{\delta y}\right|\right) \tag{23}$$

seja satisfeito. Detalhes da discretização espacial podem ser encontrados em (GRIEBEL; DORNSEIFER; NEUNHOEFFER, 1998).

Como resultado da discretização, é necessário resolver um sistema linear de Equações (8) contendo $i_{max} \times j_{max}$ incógnitas $p_{i,j}$, $i = 1,\ldots, i_{max}$ e $j = 1,\ldots, j_{max}$. Nesse trabalho, foi obtida a solução aproximada para pressão usando o método de Gauss-Seidel com algoritmo SOR. Para evitar o surgimento de oscilações, foi usado um controle do tamanho do passo no tempo (δt) adaptável baseado na conhecida condição de Courant-Friedrichs-Lewy (CFL). Isso garante

estabilidade no algoritmo numérico (ANDERSON, 1995). O tamanho do passo no tempo é dado por:

$$\delta t = \tau \min\left[\frac{\mathrm{Re}}{2}\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{u_{\max}}, \frac{\delta y}{v_{\max}}\right] \tag{24}$$

onde o fator $\tau \in (0, 1]$ é o fator de segurança.

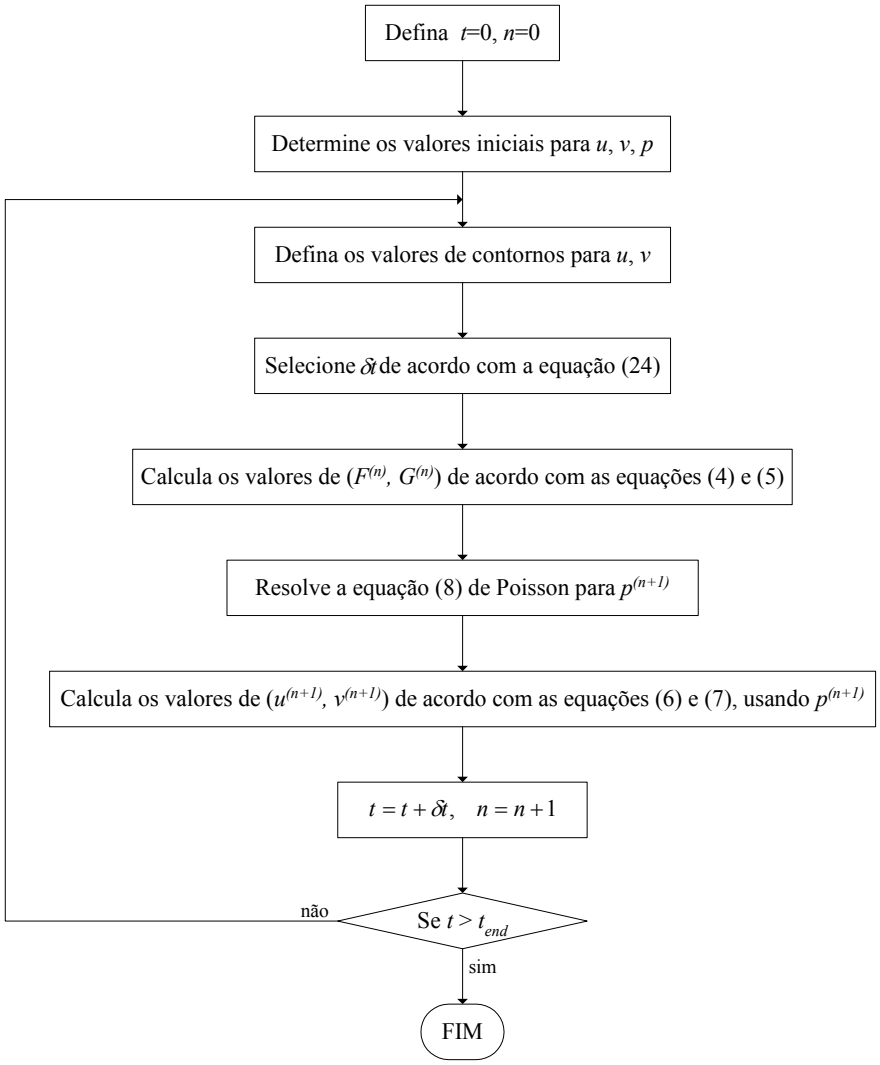


Figura 2.2: Algoritmo para as equações de Navier-Stokes

O procedimento completo consiste nos seguintes passos esquematizados na Figura 2.2. Inicialmente é feita a inicialização dos valores de velocidades e pressões (que dependem do tipo de problema simulado), e o cálculo do tamanho do passo no tempo. O tamanho do passo no tempo escolhido foi para atender o critério CFL (Equação (24)). Depois disso, os valores

de F e G são determinados, e os coeficientes do conjunto linear de equações para a pressão são calculados. Então, é encontrada a solução da equação da pressão para os pontos nodais do domínio, resolvendo o sistema linear de equações usando iterações SOR. Em seguida, os componentes de velocidade são calculados. Esse processo iterativo continua até que o processo alcance o tempo final da simulação.

2.3 Paralelização

Na arquitetura de memória compartilhada, o paralelismo é principalmente direcionado para executar um conjunto idêntico de operações em paralelo na mesma estrutura de dados (paralelização de *do-loops*). O paralelismo em sistemas de memória distribuída é principalmente direcionado em subdividir os dados em estruturas dentro de subdomínios e designar cada subdomínio a um processador. Neste caso, o mesmo código é executado em todos os processadores com seu próprio conjunto de dados. Por exemplo, dividindo o domínio computacional em quatro subdomínios, é possível distribuir o trabalho entre quatro diferentes processadores (Figura 2.3).

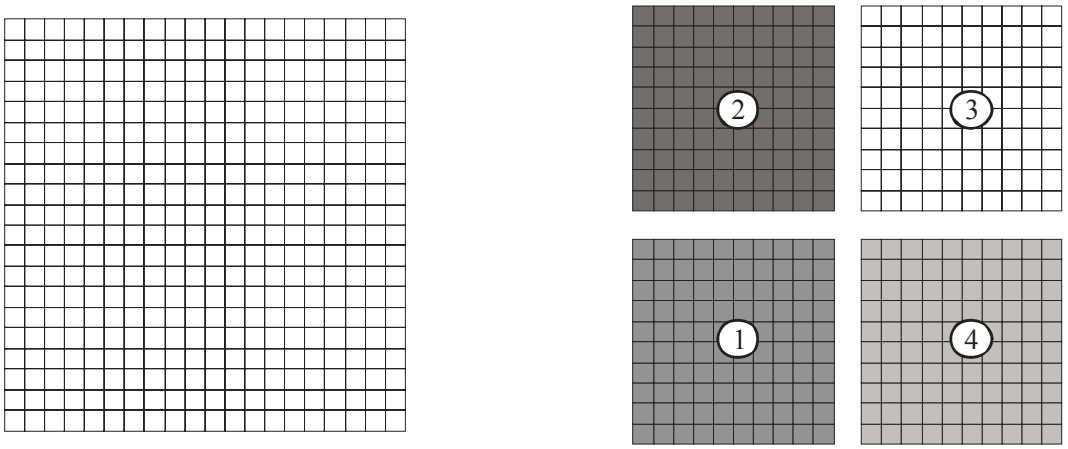


Figura 2.3: (a) Representação esquemática de uma malha computacional genérica e (b) sua decomposição em quatro subdomínios.

No entanto, é importante notar que para calcular as variáveis em cada ponto nodal, as variáveis em sua vizinhança são requisitadas. Dessa forma, para calcular as variáveis nos

pontos próximos à interface entre os subdomínios, um processador requisitará informações armazenadas na memória de um outro processador. Isso requer alguma quantidade de comunicação em intervalos regulares, a qual pode diminuir a velocidade do processamento.

Em geral, o procedimento computacional envolve três passos:

a) Particionamento do domínio de solução;

b) Execução do processamento em cada processador para atualizar seus próprios dados;

c) Comunicação entre os processadores.

Essa técnica é chamada de decomposição do domínio. A chave para um eficiente processamento é manter o mínimo possível de comunicação entre os processadores, além de dividir igualmente a carga de trabalho entre eles.

A decomposição do problema em várias partes é realizada por meio de cortes na matriz dos elementos, tanto na horizontal como na vertical, a escolha da direção é feita de forma a manter o número de pontos nodais na horizontal e vertical em cada bloco, o mais próximo possível. Em outras palavras, cada parte dividida deve ser o mais próximo possível de um quadrado, tomando como base o número de pontos nodais de cada um dos blocos. Essa recomendação representa uma tentativa de obter um número de pontos nodais no contorno entre subdomínios menor possível. Uma vez que os valores das variáveis nos pontos nodais de contorno de cada subdomínio devem ser comunicados com outros processos, um menor número de pontos nodais no contorno representará um menor volume de dados comunicados. Assim, quanto menor for o número de pontos nodais que requerem comunicação em relação ao número total de pontos nodais a processar, melhor será o rendimento do processo. Além disso, a quantidade de pontos nodais em cada um dos blocos devem se igualar, a fim de balancear a carga entre os processadores.

Uma vez que o domínio com múltiplos blocos foi construído, o processamento paralelo em cada um dos blocos pode começar se as condições de contornos desses blocos forem conhecidas. As condições de contorno para cada subdomínio podem ser: (*i*) uma condição de contorno físico ou (*ii*) uma condição de contorno de comunicação, como conseqüência da decomposição do domínio. As condições de contorno físico devem ser especificadas pelo usuário com base nas características físicas do problema. Enquanto os dados do contorno de comunicação devem ser recebidos dos blocos vizinhos, que podem estar em processadores diferentes. Dados do contorno de comunicação são mantidos em *buffers* nos contornos de cada bloco como mostrado na Figura 2.4, que ilustra um subdomínio e os pontos nodais do

buffer usados para armazenar os dados. Logo que os dados do buffer forem recebidos de todos os blocos vizinhos, o processamento desse bloco pode começar, usando o algoritmo seqüencial. No término da solução para o bloco, os dados nos contornos são enviados para os blocos vizinhos. Então, esse bloco aguarda pela atualização de seu buffer proveniente dos blocos vizinhos, em seguida, começa o próximo ciclo de processamento.

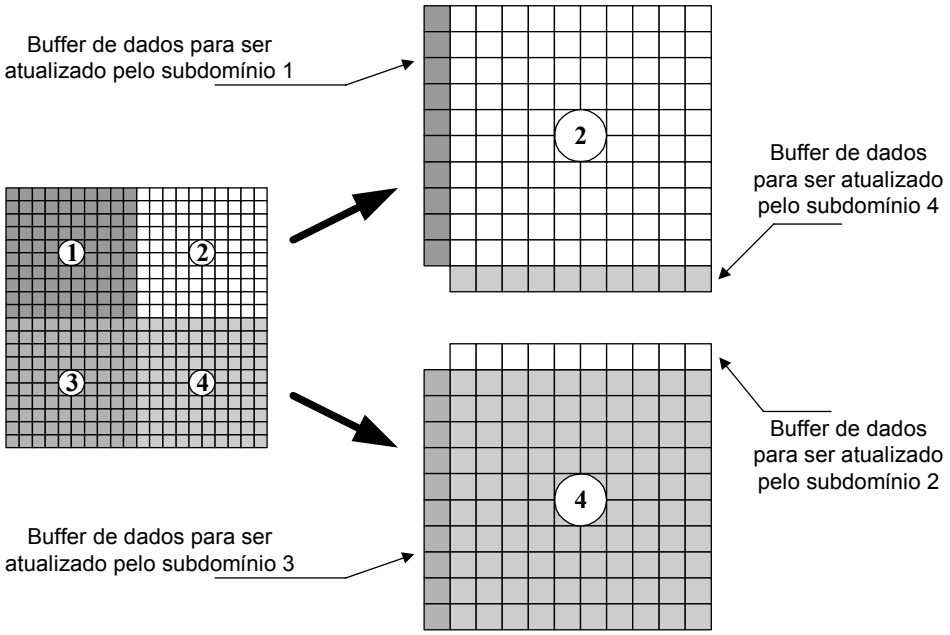


Figura 2.4: Representação esquemática do domínio dividido em quatro subdomínios, indicando os pontos nodais do buffer usado para armazenar os dados do contorno interno.

A Figura 2.5 mostra a seqüência de operações envolvidas no processamento. Cada processador executa o processamento de seu próprio subdomínio, onde um subdomínio é executado por apenas um processador, mas como foco deste trabalho, um processador pode executar o processamento de mais de um subdomínio. O processamento inclui a inicialização dos valores de velocidades e pressões, e cálculo do tamanho do passo no tempo. Assim que for calculado o valor local do tamanho do passo no tempo para cada subdomínio (Equação (24)) baseado nos dados locais do subdomínio, algumas comunicações são necessárias para a escolha do passo no tempo para ser usado em todos os cálculos dos subdomínios. Isso é exigido porque todos os processos necessitam avançar para o próximo passo no tempo usando o mesmo valor de δt no algoritmo de solução atual. O tamanho do passo no tempo deve ser para atender o critério CFL (Equação (24)) para todos os subdomínios, sendo assim, o menor passo no tempo foi escolhido. Depois disso, os valores de F e G são determinados, e os coeficientes do conjunto linear de equações para a pressão são calculados. Então, cada bloco

individualmente encontra a solução da equação da pressão para os pontos nodais em seu subdomínio, resolvendo o sistema linear de equações usando iterações SOR. Uma vez que os valores de pressão são calculados para cada bloco, os valores de pressão em cada contorno do bloco são comunicados para seus vizinhos.

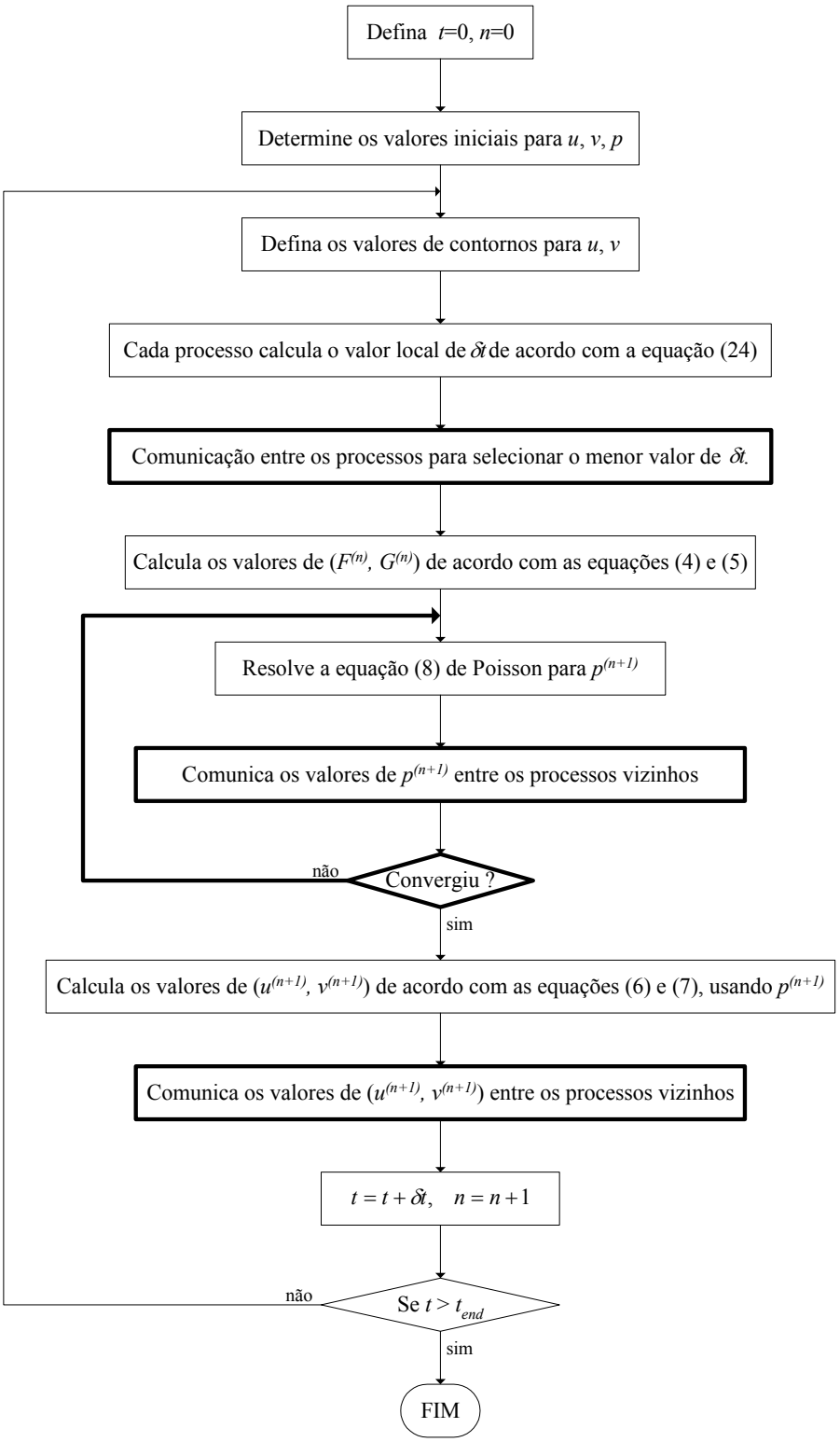


Figura 2.5: Algoritmo paralelo para Navier-Stokes

Esse procedimento é repetido até a convergência ser alcançada, na qual é checada comparando o valor do resíduo após a comunicação. Os componentes de velocidade são calculados em cada subdomínio, e seus valores em cada contorno do bloco são comunicados para seus vizinhos. Esse processo iterativo continua até que o processo inteiro alcance a convergência.

O valor do resíduo foi calculado de acordo com a Eq. (25), que é função dos resíduos ($R_{j,i}$) no cálculo do valor da pressão em cada ponto nodal, utilizando a Eq. (9), após uma ou mais iterações SOR. Independentemente do número de processos e processadores utilizados na execução da simulação, o cálculo do resíduo global (R_{global}) considera um malha inteira de pontos nodais.

$$R_{j,i} = Aw_{j,i}P_{j,i-1} + Ae_{j,i}P_{j,i+1} + As_{j,i}P_{j-1,i} + An_{j,i}P_{j+1,i} - Ap_{j,i}P_{j,i} - B_{j,i}$$

$$R_{global} = \sqrt{\sum_{i=1}^{i_{\max}} \sum_{j=1}^{j_{\max}} R_{j,i}{}^2} \qquad (25)$$

3 Metodologia

Este capítulo apresenta a metodologia utilizada para testar o desempenho e consistência do algoritmo proposto. A Seção 3.1 apresenta uma descrição dos problemas selecionados para avaliação da consistência do algoritmo e da acurácia de suas predições. A Seção 3.2 apresenta o *Cluster Enterprise*, que foi o ambiente paralelo utilizado para realização dos experimentos, e as instrumentações para levantamento de dados. Finalmente, a Seção 3.3 descreve os testes de desempenho, apresentando o procedimento de testes, as métricas utilizadas e as técnicas de instrumentação.

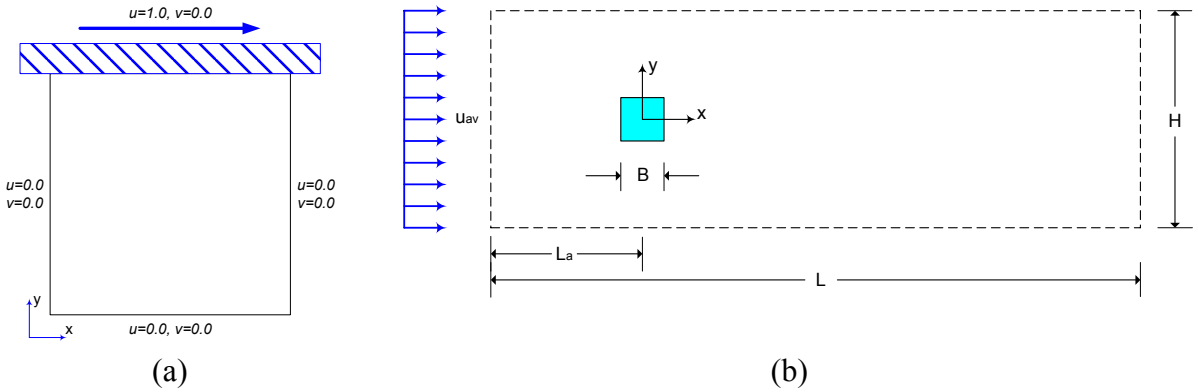
3.1 Problemas simulados

Dois problemas conhecidos na literatura serão simulados neste trabalho, ambos em duas dimensões. O primeiro problema é o da cavidade com cobertura deslizante (Figura 3.1a), que consiste em 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). Os resultados obtidos serão comparados com dados de simulação obtidos por Griebel, Dornseifer e Neunhoeffer (1998). Através da variação do valor de viscosidade do fluido serão estudas configurações com valores do número de Reynolds iguais a 100, 1000 e 10000.

O segundo problema teste representa o escoamento ao redor de um cilindro de secção quadrada (problema da barreira), que simula o comportamento do fluido ao escoar ao redor de um prisma de secção quadrada. A geometria e as dimensões relevantes são mostradas esquematicamente na Figura 3.1b. As dimensões relatadas na figura são: B=1,0; La=6,0; L=24,0 e H=10,0. Os resultados obtidos são comparados com os dados reportados por Saha, Biswas e Muralidhar (2003).

Os dois problemas representam configuração amplamente documentada pela literatura científica, e indicam dois diferentes regimes de escoamento. O primeiro 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. O segundo problema representa uma configuração que não atinge regime permanente, uma vez que o escoamento de fluidos ao redor de corpos angulosos, como o cilindro de secção quadrada, caracteriza-se por um constante movimento oscilatório do escoamento na região a montante do obstáculo.



(a) (b)

Figura 3.1: Representação esquemática dos problemas (a) da cavidade com cobertura deslizante e (b) do escoamento ao redor de um cilindro de secção quadrada (barreira).

3.2 Cluster Enterprise

Totalmente operacional em janeiro de 2003, o Cluster *Enterprise* do Laboratório de Computação de Alto Desempenho do Centro Tecnológico da Universidade Federal do Espírito Santo (LCAD – CT – UFES) é 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. O Enterprise figurava na lista dos clusters mais rápido do mundo (http://clusters.top500.org) em 48º lugar em Janeiro de 2003, com desempenho teórico máximo de 195,8 GFLOP/s e desempenho LINPACK medido de 52,3 GFLOP/s. O Enterprise possui 16GB de memória RAM e 1,2TB de capacidade de armazenamento em disco.

3.2.1 Ferramentas de Programação

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 (VALGRIND, 2004), que tem como uma de suas finalidades medir a taxa de *cache miss* de acesso à memória.

3.2.2 Rede de Interconexão

A rede de interconexão do Enterprise é composta, basicamente, por dois *switches* 3COM modelo 4300. Cada *switch* possui 48 portas *Fast-Ethernet* (100Mb/s) para comunicação com os nós de processamento e são interligados entre si através de um módulo *Gigabit Ethernet* (1000Mb/s). Para uma melhor utilização dos *switches*, a quantidade de nós de processamento é dividida igualmente entre os dois *switches*. Um dos *switches* possui uma porta *Gigabit* a mais para conexão com o servidor (*host*).

3.2.3 Nós de processamento

Tabela 3.1: Sumário dos itens de Hardware dos nós de processamento.

Item de Hardware	Descrição
Processador	ATHLON XP 1800+ (1,53MHz) fabricado pela AMD, versão Palomino de 0,18 mícrons com *cache* L1 de (64+64)Kb e L2 de 256Kb, barramento de 133MHz (ou 266MHz com dois acessos por ciclo de máquina)
Placa Mãe	Soyo, modelo SY-K7VTA PRO, FSB (front side bus) 200/266MHZ, 5 slots PCI de 32 bits e 2 barramentos PCI internos
Memória RAM	256MB, 133MHz com tempo de acesso 7ns
Disco Rígido	20GB, 5.400 RPM, fabricado pela Samsung, compatível com tecnologia SMART, padrão Ultra ATA 100, buffer de 2MB, tempo de acesso 8,9ms, latência 5,56ms, modelo Spintpoint V40
Placa de Rede	3COM, modelo 3C905TX-NM, *Fast Ethernet* 10/100 Mb/s

Cada um dos 64 nós de processamento do Enterprise possui memória SDRAM de 256MB e disco rígido Ultra ATA de 20GB. O processador de cada nó é o AMD ATHLON XP 1800+. A Tabela 3.1 sumariza as configurações de hardware dos nós de processamento. Visto que cada processador possui duas unidades de ponto flutuante e freqüência de operação de 1,53 GHz, tem-se um desempenho teórico de pico igual a 2 × 1,53 = 3,06 GFLOP/s por nó de processamento.

3.2.4 Servidor

A máquina servidora (*host*) possui praticamente a mesma configuração dos nós de processamento, exceto que conta com 512 MB de memória RAM, 80 GB de disco e duas placas de rede, uma 3COM *Gigabit Ethernet* para conexão com um dos *switches* do Enterprise e uma 3COM *Fast-Ethernet* para conexão externa. Esta é a única máquina que possui acesso aos nós de processamento. Esta máquina também é responsável pela distribuição dos processos nos nós de processamento, pelo armazenamento das contas dos usuários (utilizando sistema de arquivos NFS), além das atividades de configuração, atualização e monitoramento dos nós de processamento.

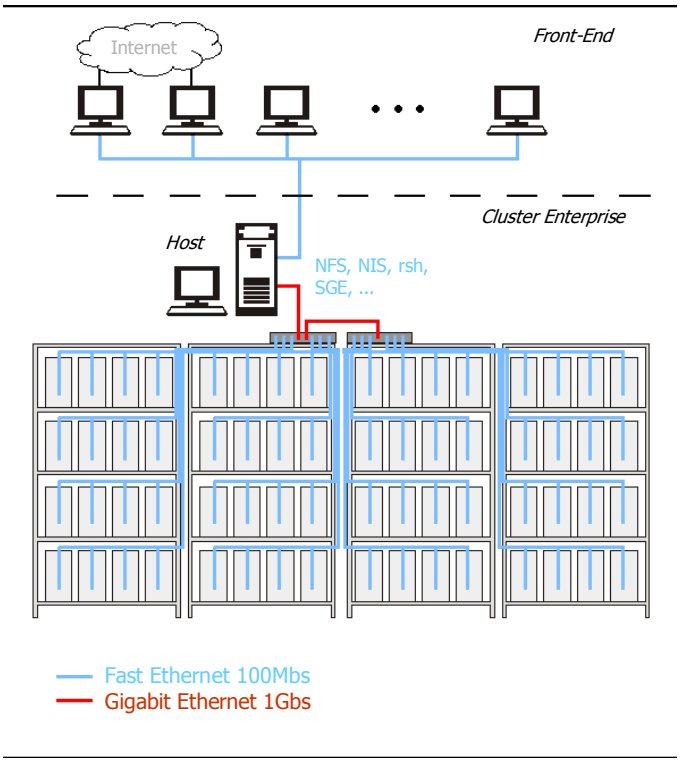


Figura 3.2: Rede de interconexão.

O gerenciamento da distribuição dos processos é feito pelo *Sun Grid Engine* (SGE) (SGE, 2004). A Figura 3.2 ilustra as conexões de rede entre as máquinas externas, o *host* e os nós de processamento.

3.3 Metodologia dos testes de desempenho

Os testes de desempenho apresentados aqui são referentes à execução do problema da cavidade. Uma análise comparativa da dinâmica de execução do algoritmo para os dois problemas em estudo, apresentou figuras de desempenho bastante similares. Assim, o problema da cavidade foi selecionado como caso base para as análises apresentadas.

É importante enfatizar que as simulações executadas com um processador são de fato, um pouco diferentes daquelas executadas com dois ou mais processadores, visto que um código verdadeiramente seqüencial foi utilizado no caso de um processador. Dessa maneira, foi possível avaliar o real ganho de desempenho ao usar processamento paralelo em vez de serial.

Os testes serão realizados em diversas situações, variando: o número de processadores, o número de processos por processador, o tamanho dos problemas, o tipo de problema, o método de comunicação e algumas variáveis. Entretanto, 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 melhor resultado com o 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, será 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.

3.3.1 Instrumentação

A fim de avaliar a eficiência da técnica aplicada neste trabalho, foram feitas comparações dos dados obtidos experimentalmente, entre eles estão: o tempo de execução total, a quantidade de operações em ponto-flutuante por segundo (FLOP/s), a taxa de *cache miss*. Para medir o tempo de execução, foi utilizado a função *gettimeofday()* da biblioteca <*sys/time.h*>. Essa função retorna a hora atual do BIOS da máquina, com uma resolução de 1μs (micro-segundo). Assim, para medir a duração de execução do programa, são feitas duas chamadas dessa função: uma no início e outra no final, e é efetuada a diferença entre os tempos obtidos. A medição da taxa de *cache miss* é realizada pelo software valgrind (VALGRIND, 2004) que será tratado com maior detalhamento na próxima seção. E, para medir a quantidade de FLOP/s foi contada manualmente a quantidade de operações em ponto flutuante por função. Dessa forma, a totalização das operações em ponto flutuante é feita por instruções inseridas em cada uma das funções do algoritmo, com o propósito de acumular a quantidade de operações em ponto flutuante cada vez que a função é chamada. Por fim, esse total é divido pelo tempo de processamento, obtendo assim, a quantidade de operações em ponto flutuante por segundo (FLOP/s). No fim de cada execução, é apresentado o tempo total e o número de FLOP/s.

3.3.1.1 Instrumentação da taxa de *cache miss*

Foi utilizado o software Valgrind (VALGRIND, 2004) para medição de *cache misses*. O *cache miss* é o termo utilizado quando a memória *cache* não possui o bloco de dados que contenha a informação solicitada pelo processador ou por um nível de memória *cache* acima na hierarquia. Também pode ocorrer *cache miss* quando alguma informação é escrita e o bloco destino não está presente na memória *cache*.

O Valgrind é uma ferramenta de depuração, que lança mão do recurso extremo de emular uma CPU x86. O código x86 é traduzido para um formato intermediário, semelhante ao SPARC RISC, e este então é executado por um interpretador. Como o objetivo do Valgrind é depuração, seu emulador x86 tem várias limitações, bem como características “anormais” para um emulador de CPU: só funciona em Linux/x86; só serve para depurar executáveis Linux/x86; o executável não pode ter certos tipos de otimização, nem usar algumas técnicas de baixo nível. Todas as chamadas de sistema relacionadas a memória, como *malloc()*, são monitoradas de modo a pegar vazamentos, *overflows* etc., o que em tese escapa à competência de um emulador. O emulador do Valgrind também simula as memórias *caches* L1 e L2, o que, para este trabalho, permitiu identificar o mau uso da memória *cache* por parte da aplicação. O

interpretador de código intermediário é quase um processador RISC, faz inclusive "*register renaming*" para atingir melhor desempenho. Ainda assim, o programa instrumentado para simular as memórias *caches* executa de 20 a 100 vezes mais lento do que na velocidade normal, e pode ocupar muitas vezes mais memória.

Como foi dito, uma de suas ferramentas é o analisador de memória *cache*. Seu funcionamento consiste em analisar a execução de um determinado programa, avaliando a quantidade de leituras e escritas de dados e instruções, assim como, a taxa de *cache miss* por essas operações. Apesar de o valgrind detalhar a taxa de *cache miss* de leitura/escrita de dados e leitura de instruções em cada um dos níveis de memória *cache* que compõem a máquina, este trabalho foca somente os *cache misses* de dados na memória *cache* L2.

Devido ao Valgrind fazer emulação do processador para analisar a memória *cache*, consequentemente não é levado em consideração nenhum fator externo, por exemplo, troca de contexto entre o processo da aplicação e algum outro processo que esteja em execução, ou até mesmo rotinas do próprio sistema operacional. Sempre que há esse chaveamento entre os processos, os dados na memória *cache* são totalmente ou parcialmente removidos para dar lugar aos dados do novo processo. Ao voltar à execução para a aplicação em questão, haverá *cache miss* extras para recuperar informações que já poderiam estar na memória *cache* antes da troca. Portanto, os resultados das emulações do Valgrind são ligeiramente divergentes do real. Existem diversos trabalhos na literatura que utilizaram o Valgrind, alguns avaliaram o funcionamento do Valgrind (NETHERCOTE; SEWARD, 2003), outros que o utilizaram para apresentar um algoritmo de *prefetch* (pré-busca de dados) de hardware (WEIDENDORFER; TRINITIS, 2004), e ainda para testar a vulnerabilidade de softwares contra ataques (NEWSOME; SONG, 2005).

3.3.2 Métricas

Para avaliar os métodos de algoritmos paralelos, foram adotados alguns parâmetros de medida de desempenho dos algoritmos para processamento paralelo. Em busca de um menor tempo de execução das simulações numéricas, é comum o acréscimo de processadores ao computador paralelo, o que não significa necessariamente um ganho considerável. Pode ocorrer da adição de um novo processador causar uma redução do desempenho de execução da simulação.

Os métodos de avaliação adotados foram: *Speedup* e Eficiência Paralela (MINTY *et al*., 1999). O s*peedup* é o número de vezes que um processo paralelo fica mais rápido em relação à velocidade de processamento utilizando apenas um processador. Quando o valor do *speedup* é superior ao número de processadores, esse é conhecido como superlinear. Abaixo, é conhecido como sublinear, e igual como linear. 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. Embora pareça impossível, o *speedup* superlinear (eficiência > 1), pode ser alcançado se houver uma melhor utilização da hierarquia de memória pelo programa paralelo em relação ao programa serial (SIMOES, 2004).

4 Resultados

Neste capítulo são apresentados os resultados obtidos pela avaliação de desempenho computacional do algoritmo proposto. O capítulo está dividido em 3 seções. A Seção 4.1 apresenta uma avaliação da consistência do algoritmo e da acurácia de suas predições, através da comparação de seus resultados com outros trabalhos da literatura. A Seção 4.2 contém a análise de desempenho do algoritmo, conforme originalmente proposto por Griebel, Dornseifer e Neunhoeffer (1998). A Seção 4.3 apresenta estratégias para a otimização dos níveis de desempenho obtidos, avaliando a eficácia de cada uma destas estratégias.

4.1 Validação do Algoritmo Computacional

Conforme citado anteriormente, dois problemas conhecidos na literatura são simulados neste trabalho. O primeiro problema é o da cavidade com cobertura deslizante (Figura 3.1a), que consiste em um recipiente quadrado com lados de tamanhos iguais a 1 metro, e preenchido com um fluido. O segundo problema teste representa o escoamento ao redor de um cilindro de secção quadrada (problema da barreira), que simula o comportamento do fluido ao escoar ao redor de um prisma de secção quadrada. A geometria e as dimensões relevantes são mostradas esquematicamente na Figura 3.1b.

O problema da cavidade representa um escoamento que parte de uma condição inicial, 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. Simulações com número de *Reynols* (*Re*) iguais a 1, 10, 100, 1000, 5000 e 10000 foram executadas para o problema da cavidade. A Figura 4.1 apresenta os níveis de pressão e campos de velocidades obtidos através da simulação numérica dos escoamentos na cavidade para valores do número de Reynolds iguais a 100, 1000 e 10000. Pode-se notar que com aumento dos valores do número de Reynolds, as recirculações no interior da cavidade tornam-se mais intensas, a região central de cada recirculação exibe os menores valores de pressão enquanto as regiões de escoamento incidente na parede exibem os maiores níveis de pressão. Esta é uma tendência semelhante à obtida nos resultados reportados por Griebel, Dornseifer e Neunhoeffer (1998).

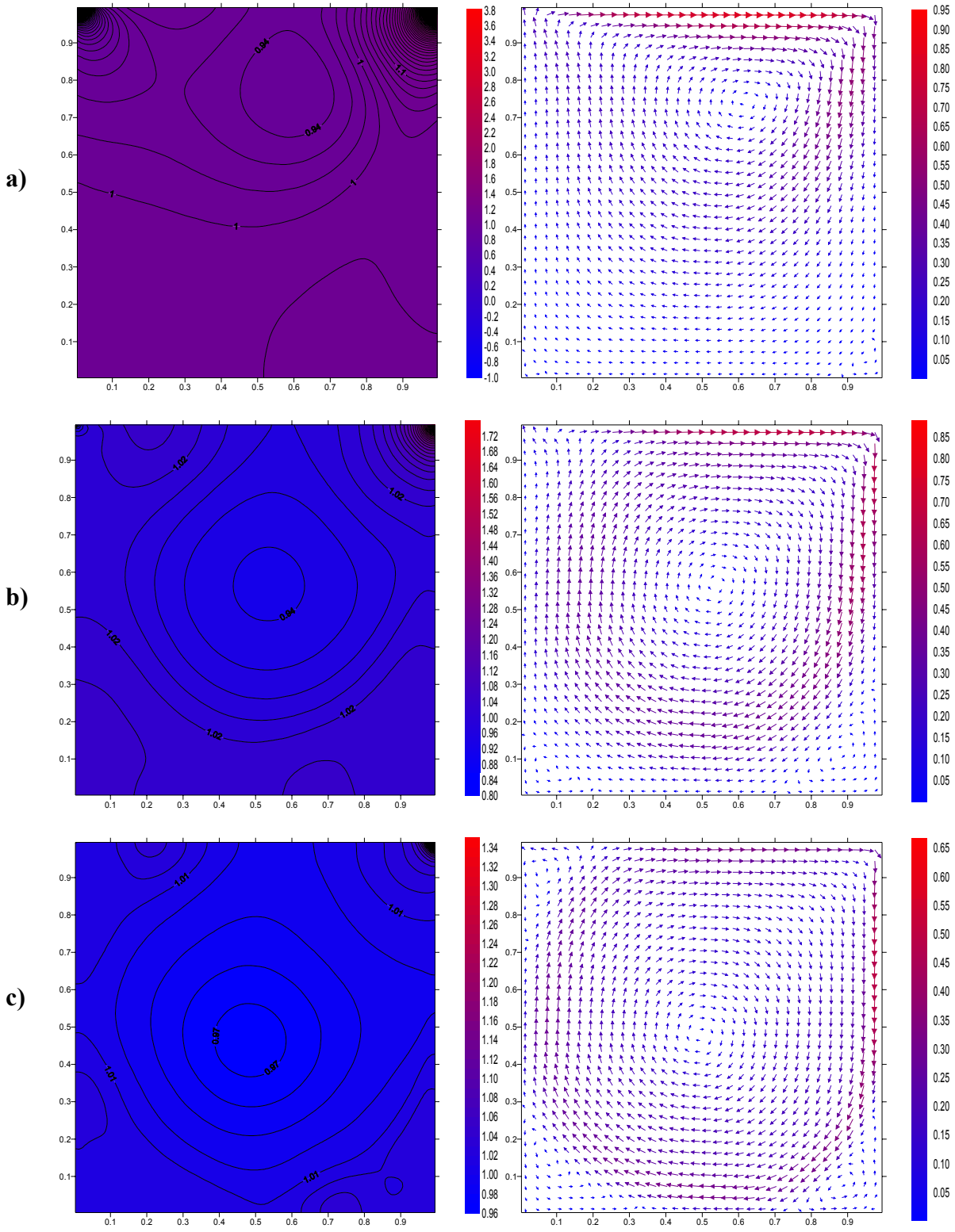


Figura 4.1: Níveis de pressão e campos de velocidades, respectivamente, para o problema da cavidade, utilizando três valores de Reynolds: (a) 100, (b) 1000 e (c) 10000. A matriz é formada por 448×448 pontos nodais, e foi simulado até o tempo de 40s.

A Figura 4.2 apresenta uma comparação entre os campos de velocidade obtidos no presente trabalho e os obtidos por Griebel, Dornseifer e Neunhoeffer (1998), para um escoamento com

número de Reynolds igual a 10000. É possível notar que existe uma aproximação muito grande entre os dois campos de velocidade, o centro da principal recirculação é localizado aproximadamente na mesma posição dentro da cavidade, e a segunda recirculação no canto apresenta dimensões similares.

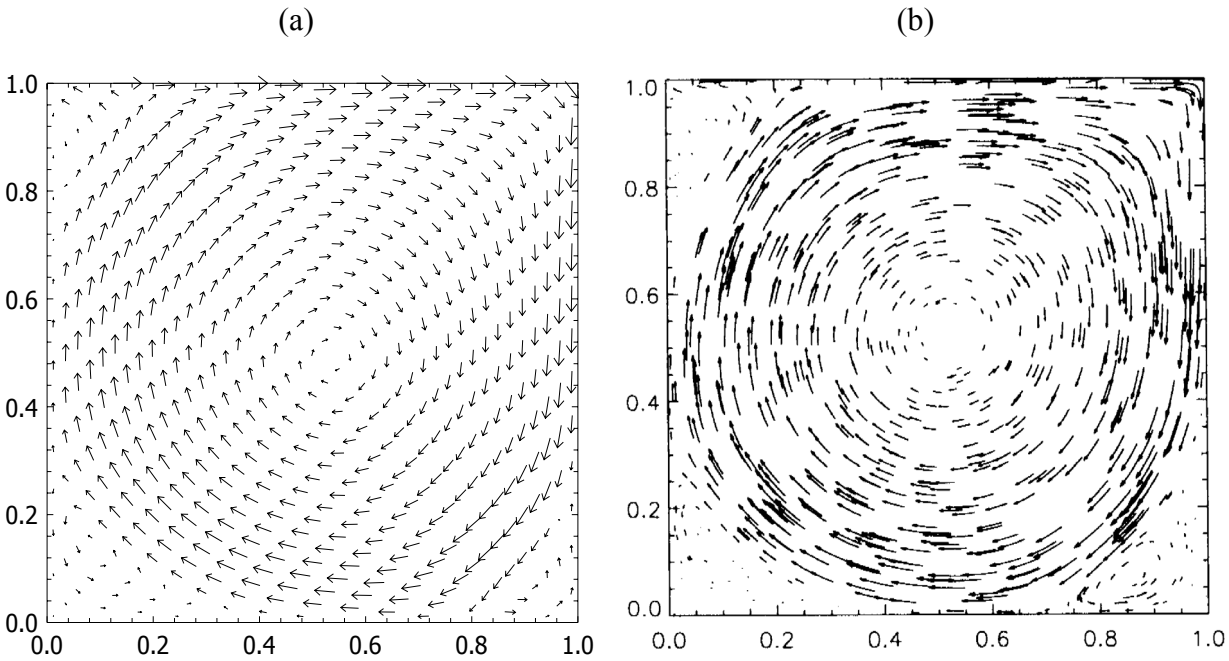


Figura 4.2: Campos de velocidades obtidos pelo problema da cavidade por *Re*=10000, (a) usando o algoritmo proposto neste trabalho e (b) os resultados apresentados por Griebel, Dornseifer e Neunhoeffer (1998).

O segundo problema representa uma configuração que não atinge regime permanente, uma vez que o escoamento de fluidos ao redor de corpos angulosos, como o cilindro de secção quadrada, caracteriza-se por um constante movimento oscilatório do escoamento na região a montante do obstáculo. Foi simulado um escoamento com número de Reynolds igual a 150. A seguir estão apresentados a evolução temporal do campo de velocidade (Figura 4.3), níveis de pressão (Figura 4.4) e variações do componente horizontal de velocidade (Figura 4.5). É possível notar claramente o comportamento oscilatório do escoamento.

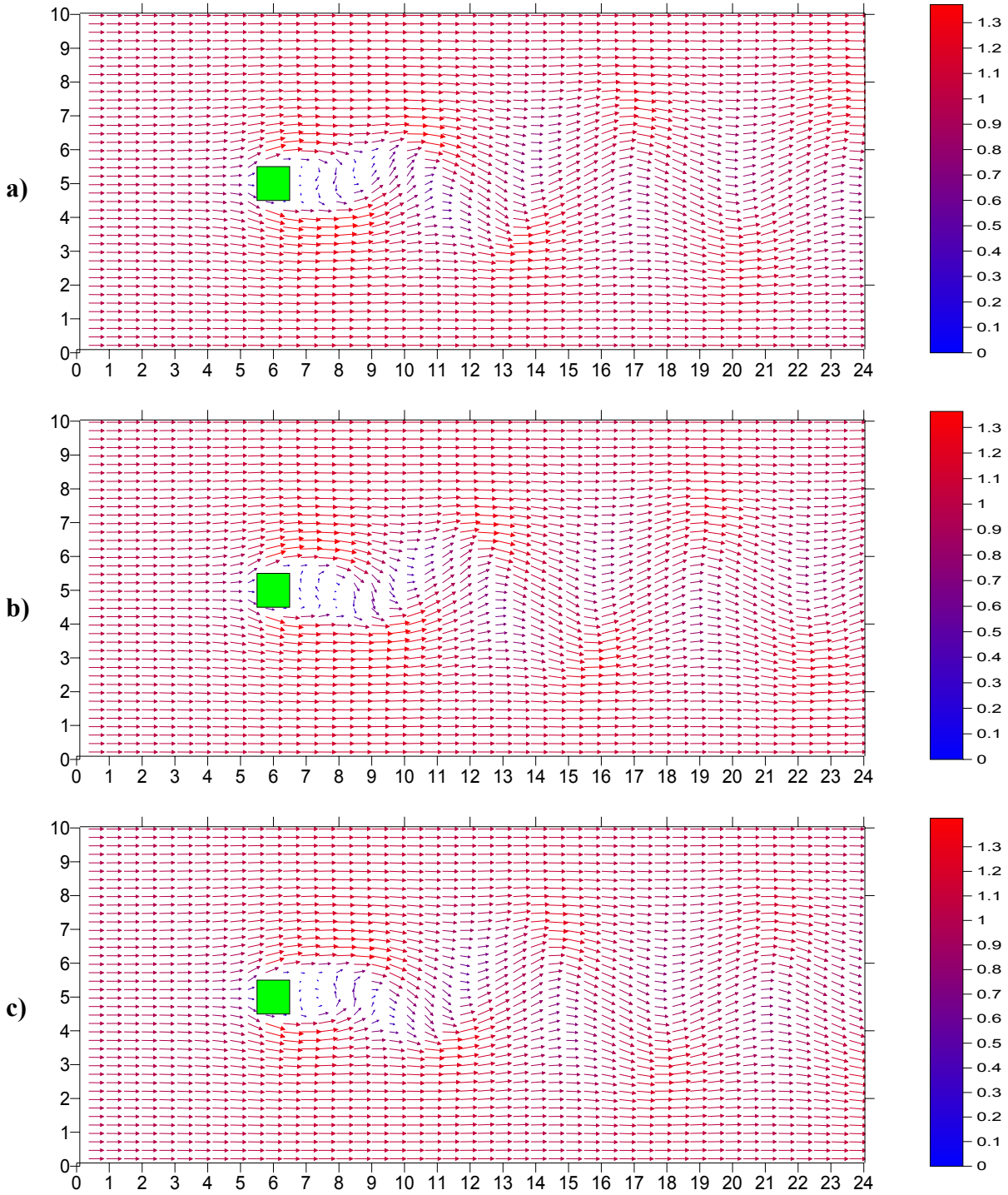


Figura 4.3: Campos de velocidades obtidas na simulação do problema da barreira utilizando 356×160 pontos nodais, Reynolds 150 e instantes de tempo iguais a a) 245.6s, b) 247.8s e c) 250.0s.

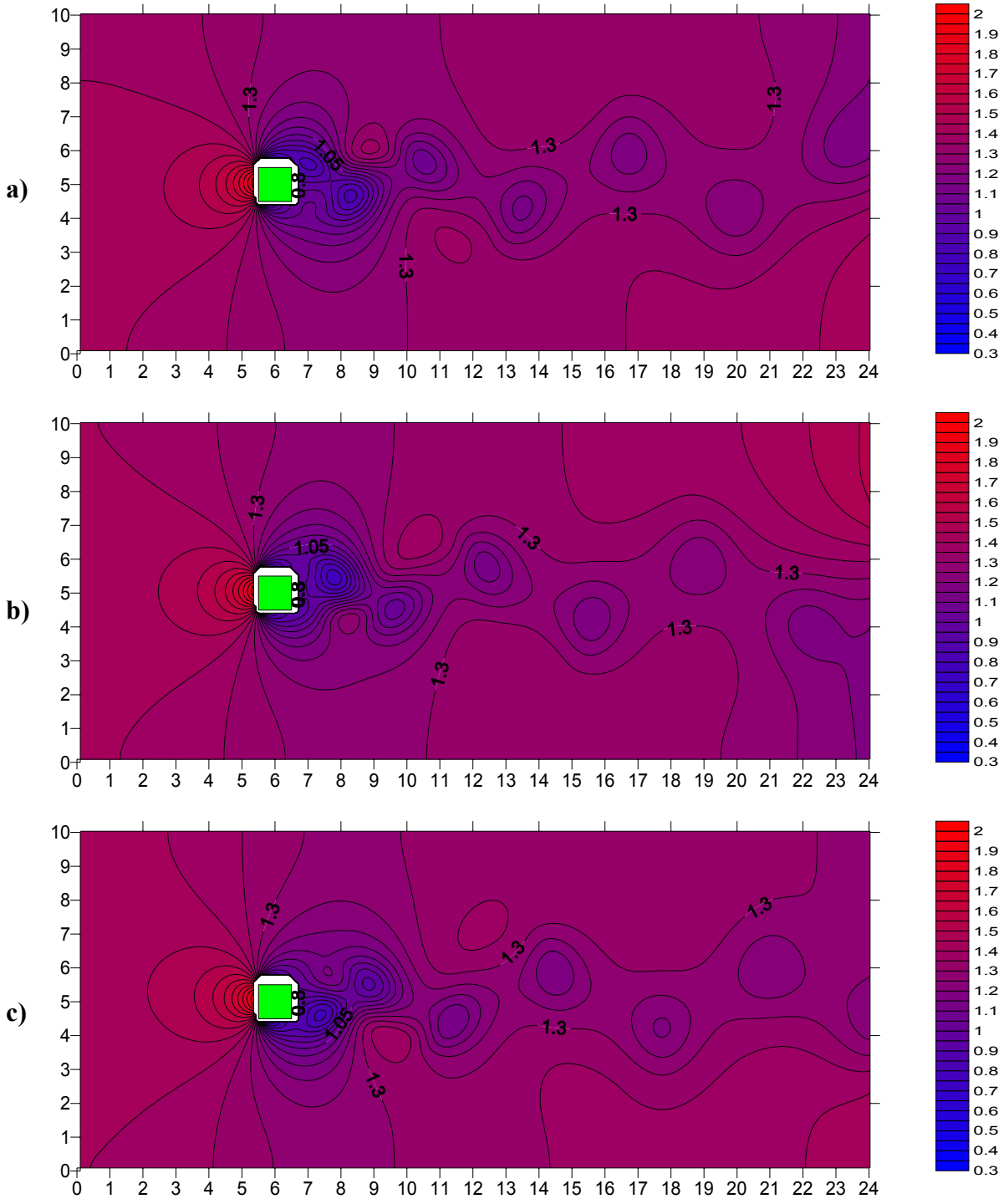


Figura 4.4: Níveis de pressão obtidos na simulação do problema da barreira utilizando 356×160 pontos nodais, Reynolds 150 e instantes de tempo iguais a a) 245.6s, b) 247.8s e c) 250.0s.

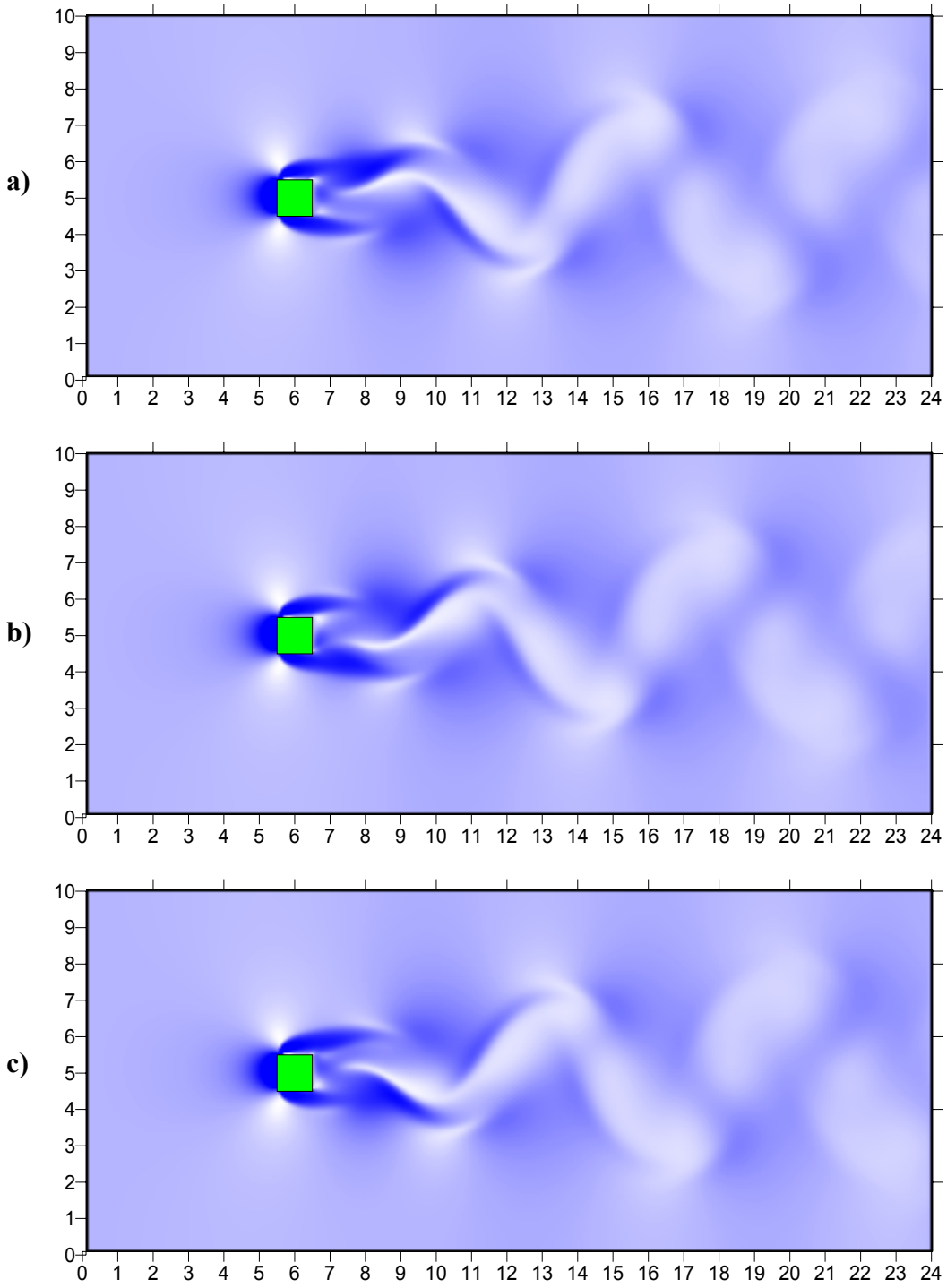


Figura 4.5: Variações do componente u (horizontal) da velocidade obtido na simulação do problema da barreira utilizando 356×160 pontos nodais, Reynolds 150 em instantes de tempo iguais a a) 245.6s, b) 247.8s e c) 250.0s.

Saha, Biswas e Muralidhar (2003) simularam numericamente esse escoamento e determinaram a freqüência das oscilações verticais e horizontais de pressão ao redor do obstáculo. Esses autores obtiveram freqüências de oscilação na vertical e na horizontal nos valores de 0,163Hz e 0,333Hz, respectivamente. Neste trabalho, foram obtidos valores de

0,169Hz para a freqüência de oscilação na vertical e 0,333Hz para a freqüência de oscilação na horizontal, indicando uma boa concordância com as simulações efetuadas por Saha, Biswas e Muralidhar (2003).

4.2 Análise de Desempenho

A avaliação de desempenho foi baseada no tempo de execução, no *speedup* e na eficiência paralela do código computacional, determinados pelo acréscimo no número de processadores usados na computação. Foram empregadas 3 malhas de tamanhos diferentes: 256×256, 512×512 e 1024×1024, as quais representam aproximadamente 65×10^3, 260×10^3 e 1×10^6 pontos nodais, respectivamente. A primeira malha representa um problema de proporções medianas em MFC, a segunda representa um problema grande e a terceira um problema bastante grande. O Gráfico 4.1 apresenta o tempo de execução para as simulações com três malhas diferentes. É possível notar uma considerável redução no tempo de execução para todas as três malhas à medida que o número de processadores aumenta.

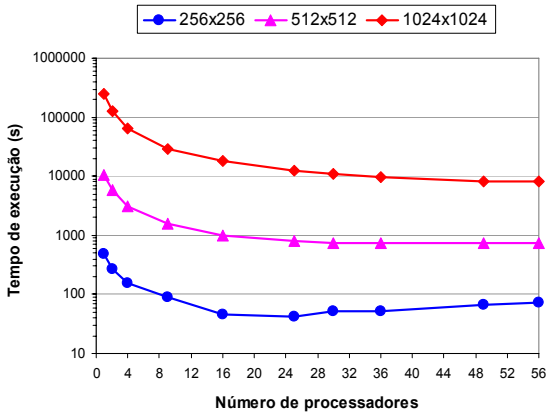
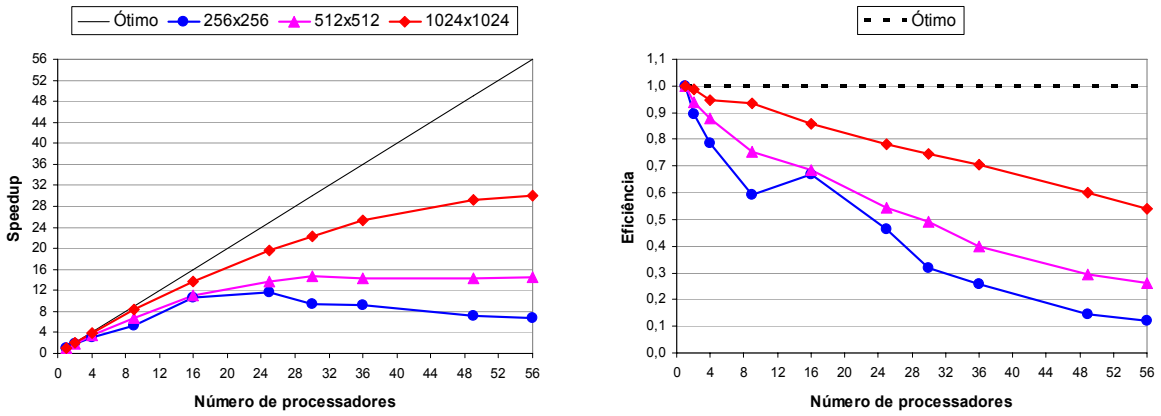


Gráfico 4.1: Tempo de execução obtidos para malhas de 256×256, 512×512 e 1024×1024, com simulações executadas em 1, 2, 4, 9, 16, 25, 30, 36, 49 e 56 processadores.

O Gráfico 4.2a compara os resultados obtidos com o *speedup* ideal (redução linear do tempo de processamento à medida que aumenta o número de processadores). É possível notar que o *speedup* obtido para a malha 256×256 está longe do *speedup* ideal. Quando dois ou mais processadores são usados, existe um significativo aumento de desempenho, mas não um ganho linear. Conforme o número de processadores aumenta, o comportamento não linear do

speedup obtido se torna mais evidente. Embora os pontos nodais sejam divididos igualmente entre os processadores, os pontos nodais que requerem comunicação em cada bloco de processamento representam uma grande proporção do total do número de pontos nodais em cada bloco. Por causa da velocidade limitada das redes de comunicação, cada processador gasta uma quantidade significativa de tempo aguardando por informação de outros processadores.

Quanto menor for o número de pontos nodais no subdomínio, e quanto mais veloz for o processador para executar as operações designadas a ele, mais significativo será o tempo gasto devido à comunicação. Pois, quanto menor for o subdomínio, maior será a razão entre a quantidade de pontos nodais para comunicação e quantidade de pontos nodais para processamento. Dessa forma, o processador aguardará por mais tempo a comunicação do que em relação ao tempo de processamento dos pontos nodais. Esse fato é até mais notável quando o número de processadores é incrementado e a proporção de pontos nodais que requerem comunicação em cada bloco de processamento se torna maior. Isso pode ser visto nos processamentos da malha com 256×256 pontos nodais utilizando 30 ou mais processadores, pois obtiveram tempo de execução ligeiramente maior do que com 25 processadores (Gráfico 4.1).



(a) (b)

Gráfico 4.2: (a) *Speedup* e (b) eficiência paralela obtidos para malhas de 256×256, 512×512 e 1024×1024, com simulações executadas em 1, 2, 4, 9, 16, 25, 30, 36, 49 e 56 processadores.

Essa tendência é mais evidente em malhas menores. Dessa forma, malhas maiores tendem a oferecer *speedup* maiores e um uso mais eficiente do *cluster* ao se adicionar processadores ao sistema. Os *speedups* obtidos com a utilização de 56 processadores foram de 6.7, 14.6 e 30.2, para as malhas de 256×256, 512×512 e 1024×1024, respectivamente.

O Gráfico 4.2b apresenta a eficiência paralela obtida para o processamento das 3 simulações com o número de processadores variando de 1 a 56. Pode-se notar que os níveis de eficiência são inferiores a um, indicando que a velocidade de processamento do sistema não é totalmente aproveitada; isso se agrava com o aumento do número de processadores.

Ainda sobre o problema com malha 256×256, no Gráfico 4.2b, pode-se notar que há uma melhoria da eficiência ao passar de 8 para 16 processadores. Esse efeito está relacionado ao uso mais eficiente da memória *cache* do processador. Embora neste caso exista uma tendência de diminuir a velocidade de processamento devido à comunicação, o tamanho necessário da memória para processar cada bloco diminui à metade quando é passado de 8 para 16 processadores. Dessa forma, dividir a malha de 256×256 em 16 blocos permite um uso mais eficiente da memória *cache* do processador, reduzindo o número de *cache misses*. Visto que o tempo de acesso à memória *cache* é de 5 a 10 vezes menor do que à memória convencional, cada processador executa a computação em seu próprio subdomínio ligeiramente mais rápido, reduzindo o tempo total de processamento. Todavia, à medida que o domínio é ainda mais dividido, o aumento do desempenho fornecido pelo uso mais eficiente da *cache* não é suficiente para evitar a degradação do desempenho devido ao aumento relativo da comunicação frente à computação.

4.3 Estratégias de Otimização

As figuras de desempenho obtidas, apresentadas na seção anterior, revelam que apenas uma fração da capacidade total de processamento do sistema é utilizada em *clusters* com muitos processadores. Nesta seção são apresentadas algumas estratégias para melhorar o desempenho, reduzindo o tempo total de processamento. São analisados parâmetros referentes à otimização da comunicação entre processadores e uso mais eficiente da hierarquia de memória. A Seção 4.3.1 explora a redução da freqüência de comunicação entre os processadores, reduzindo o volume das operações de comunicação e sincronização durante o processamento. A Seção 4.3.2 analisa diferentes estratégias para a execução da operação de sincronização entre os processadores, avaliando a eficiência da utilização de sincronizações do tipo mestre-escravo, broadcast (ou todos-para-todos) e árvore binária. Finalmente, a Seção 4.3.3 discute o uso mais eficiente da memória *cache*.

4.3.1 Freqüência de Comunicação

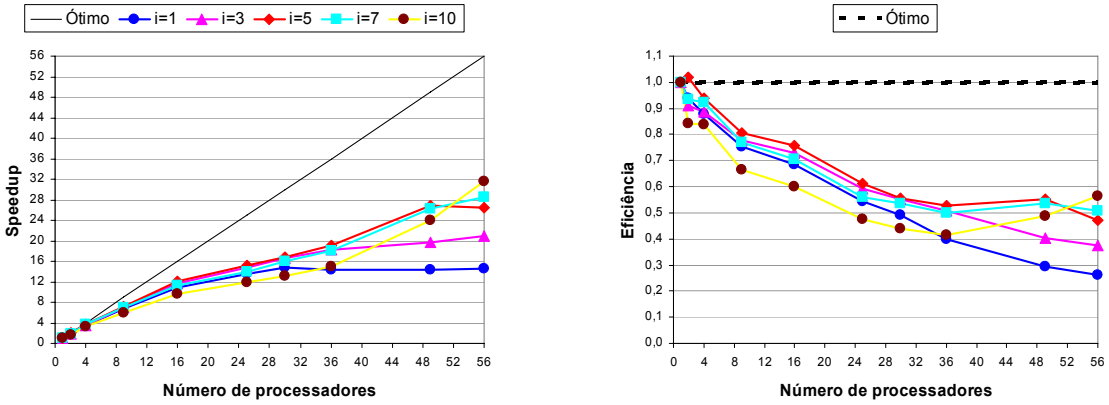
O algoritmo computacional originalmente proposto por Griebel, Dornseifer e Neunhoeffer (1998) assume que após cada iteração SOR deveria ser realizado um passo na comunicação, de forma que os valores nos contornos fossem atualizados em cada iteração SOR. Embora esse procedimento assegure uma taxa rápida de convergência, devido ao forte acoplamento entre os subdomínios, isso pode também introduzir uma quantidade significativa de comunicações, o qual pode aumentar significativamente o tempo de execução.

Alternativamente, poderiam ser realizadas algumas iterações SOR antes de cada comunicação, reduzindo assim, a quantidade total de comunicações durante o processamento. No entanto, a redução da freqüência de comunicação reduz o acoplamento entre os subdomínios, que pode reduzir a taxa de convergência, levando a um maior número de iterações globais. Dessa forma, a escolha da freqüência de comunicação é um compromisso entre o acoplamento da solução nos subdomínios e o tráfego da rede devido ao volume de mensagens. Na realidade, deseja-se comunicar com maior freqüência possível para assegurar uma rápida convergência, mas sem causar uma sobrecarga na rede.

Para avaliar a freqüência a qual as comunicações entre os processos devem ocorrer, o número de iterações SOR antes de cada comunicação foi variado. O Gráfico 4.3 apresenta os *speedups* e as eficiências paralelas obtidas por diversas freqüências de comunicação, com simulações executadas em 1, 2, 4, 9, 16, 25, 30, 36, 49 e 56 processadores, para uma malha com 512×512 pontos nodais. O número de iterações SOR entre as comunicações foi de 1, 3, 5, 7 e 10. É importante notar que, nestes testes, a comunicação é realizada exatamente na freqüência da comunicação do início ao fim da simulação, por exemplo, se foi definido utilizar 7 iterações SOR para cada passo na comunicação, esse valor será usado do inicio ao fim da simulação.

Os resultados indicam uma redução significativa no tempo de execução pela diminuição da freqüência de comunicação, especialmente para grandes números de processadores. Por exemplo, usando 56 processadores um *speedup* de aproximadamente 32 foi obtido na utilização de 10 iterações entre as comunicações, enquanto que para as simulações onde um passo da comunicação foi feito para cada iteração SOR, o *speedup* obtido foi próximo de 15. Para simulações com grandes números de iterações SOR entre as comunicações, é possível notar que o *speedup* obtido está abaixo do *speedup* para simulações onde o passo de comunicação foi realizado a cada 5 iterações SOR, exceto no caso de 56 processadores. Neste caso, devido ao excesso de comunicações, convém manter o processador ocupado realizando mais iterações antes de comunicar. Isso indica claramente, que a escolha da freqüência de

comunicação é um compromisso entre o acoplamento da solução nos subdomínios e do tráfego de rede devido ao alto volume de mensagens, como declarado anteriormente.



(a) (b)

Gráfico 4.3: (a) *Speedup* e (b) eficiência paralela obtidos para várias freqüências de comunicação, definindo o número de iterações SOR iguais a 1, 3, 5, 7 e 10, em uma malha de 512×512 pontos nodais.

4.3.2 Métodos de Comunicação

Como descrito anteriormente, existem dois estágios no procedimento computacional que requerem comunicação entre todos os processadores: (*i*) o cálculo do tamanho do passo no tempo para a próxima iteração no tempo e (*ii*) o teste de convergência da solução. Por exemplo, no cálculo do tamanho do passo no tempo, cada processador calcula seu próprio tamanho máximo permitido do passo no tempo (Equação (24)). No entanto, o valor do passo no tempo usado para a próxima iteração precisa ser o valor mínimo entre os valores calculados em todos os processadores. Esta tarefa exige a comunicação entre todos os processadores.

Esta operação pode ser realizada em três diferentes estratégias de comunicação: troca de mensagens entre todos os processadores (*broadcast*), estrutura mestre-escravo ou em uma árvore binária. Na primeira estratégia (Figura 4.6a), todo processador deve comunicar seu tamanho do passo no tempo para os outros processadores no sistema, assim, cada processador calcula o valor mínimo e o seleciona com o tamanho global do passo no tempo para a próxima iteração no tempo. Esse método requer uma comunicação *todos-para-todos* ou *broadcast*, a

qual pode causar excesso de comunicação na rede, uma vez que *n(n-1)* mensagens precisariam ser enviadas (onde *n* é o número de processadores envolvidos na computação).

Alternativamente, todo processador poderia enviar seu tamanho do passo no tempo para um único processador (Figura 4.6b), o qual selecionaria o valor mínimo e o enviaria de volta para todos os processadores. Essa estratégia é conhecida como *mestre-escravo*. Esse procedimento proporciona uma considerável redução no número de mensagens enviadas (*2n-2*). Todavia, esse procedimento obriga a realização de dois estágios de comunicação, um estágio para envio de informações dos nós escravos para o nó mestre e outro para envio de informações do nó mestre para os escravos. A comunicação em dois estágios pode causar alguma diminuição no desempenho devido à alta latência da rede. Além disso, o mais relevante é que uma única máquina receberá e enviará mensagens para todas as outras, causando assim, um "gargalo" na comunicação.

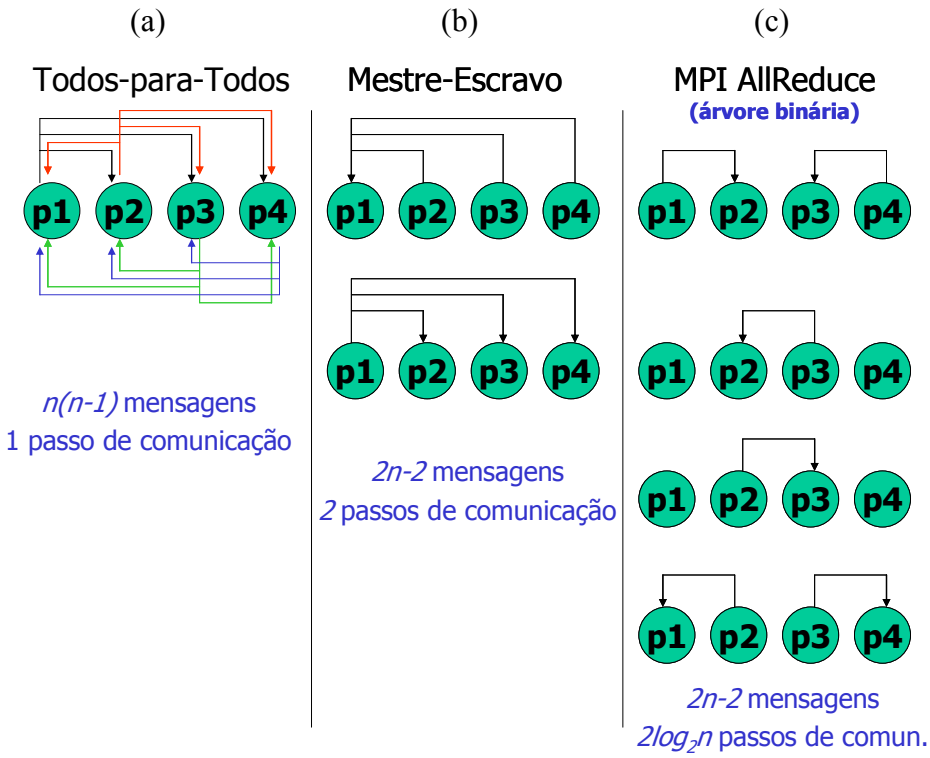


Figura 4.6: Representação das trocas de mensagens entre os processos, para os métodos de comunicação: (a) todos-para-todos, (b) mestre-escravo e (c) árvore-binária.

O terceiro método emprega uma árvore binária para efetuar as operações de cálculo de mínimo e comunicação. Sua implementação é obtida através da primitiva MPI "*All-reduce*" (Figura 4.6c). A função "*All-reduce*" executa todas as comunicações exigidas usando uma árvore binária para comunicação, ou seja, comunicações são combinadas em grupos de dois para produzir um único resultado, que por sua vez, é transmitido formando um novo grupo.

Esse ciclo é repetido até que o resultado intermediário da operação em cada passo chegue ao processo raiz, o qual é então enviado de volta para todos os processadores usando a mesma árvore binária. O uso da função "*All-reduce*", assim como o *mestre-escravo*, utiliza *2(n-1)* mensagens, mas ao contrário do *mestre-escravo*, distribui o processamento do cálculo do valor mínimo entre todos os processadores. Porém, ao utilizar o algoritmo de comunicação de árvore binária, o número de estágios na comunicação seria de entorno de $2log_2n$, ou seja, uma operação "*All-reduce*" com 32 processadores seria executado com 5 passos de comunicação até encontrar o valor mínimo no processo raiz, mais 5 passos para transmiti-lo para todos os processos. Para evitar esse excesso de comunicação, as implementações do MPI utilizam internamente na operação "*All-reduce*" estratégias para reduzir o elevado número de passos de comunicação, por exemplo, nos últimos passos de comunicação até chegar ao processo raiz, poderia ser feito uma comunicação *todos-para-todos*, uma vez que o número de processos envolvidos é pequeno. Dessa forma, evitam-se alguns passos de comunicação sem sobrecarregar a rede. Em outras palavras, não haverá um único processo raiz, mas sim, um conjunto, que seleciona entre si o menor valor e o transmite para todos os outros processadores. Além disso, como não é feito nenhum cálculo sobre os dados nas mensagens de volta, é possível fazê-las utilizando *broadcast* em vez de percorrer novamente a árvore binária em sentido oposto.

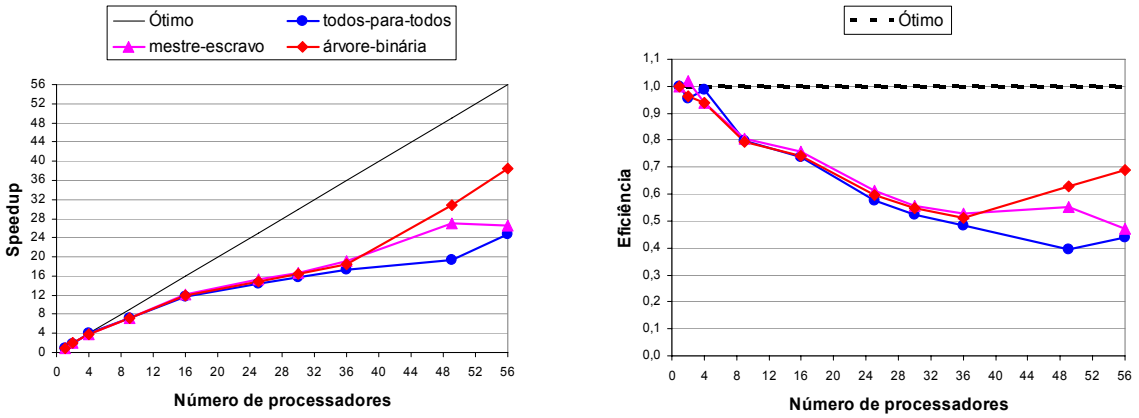


Gráfico 4.4: (a) *Speedup* e (b) eficiência paralela obtidos para comunicações: todos-para-todos, metre-escravo e árvore binária. Definindo o número de iterações SOR igual 5, em uma malha de 512×512 pontos nodais.

O Gráfico 4.4 mostra o tempo de execução, *speedups* e eficiência paralela obtidos para comunicação: *todos-para-todos*, *mestre-escravo* e *árvore-binária* com simulações executando com 1, 2, 4, 9, 16, 25, 30, 36, 49 e 56 processadores para a malha com 512×512 pontos nodais e utilizando 5 iterações SOR entre as comunicações. Os resultados indicam que não existe

diferença significativa entre as implementações para simulações executadas com menos de 36 processadores. No entanto, à medida que o número de processadores ultrapassa 36, a vantagem de uma implementação *árvore-binária* é bastante notável. Conforme o número de subdomínios aumenta, o número de mensagens se torna um importante fator limitante e tende a baixar a velocidade de processamento. Note 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.

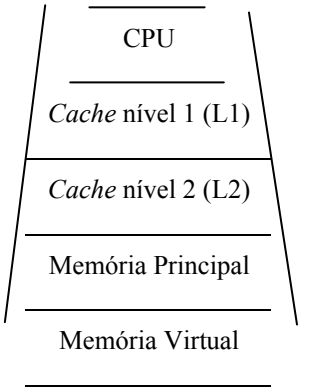
4.3.3 Estratégias de utilização eficiente da *cache*

Conforme discutido anteriormente, além da velocidade de comunicação entre nós, outro fator extremamente significativo no desempenho obtido é a grande diferença entre a velocidade de processamento dos microprocessadores e a capacidade de transferência das memórias dos sistemas microprocessados, uma vez que a evolução das memórias não tem acompanhado a dos microprocessadores no mesmo passo.

A Figura 4.7 apresenta um exemplo de uma arquitetura hierárquica de memória, onde se pode observar o tamanho, a velocidade de transferência de dados para a CPU e a latência de cada nível hierárquico. Estes níveis hierárquicos usam em geral 3 ou 4 tipos de memória. Os níveis hierárquicos de menor capacidade possuem maior velocidade de transferência de dados para a CPU, enquanto os níveis de maior capacidade possuem menor taxa de transferência. Os níveis de memória *cache* (L1, L2 e L3) são consideravelmente mais rápidos que a memória principal do sistema, e esses 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 deveria ser capaz de acessar variáveis (ou blocos de memória) que estariam armazenados em um dos níveis da memória *cache* (preferencialmente L1). Infelizmente, aplicações de 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. Isto 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).



	Capacidade	Velocidade de transferência de dados	Latência
←	128 Kbytes	9403 MB/s	2 ns
←	256 Kbytes	3121 MB/s	16 ns
←	256 MBytes	386 MB/s	195 ns

Figura 4.7: Hierarquia de Memória Athlon XP 1533MHz.

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. O trabalho enfocou explicitamente o método *multigrid*, mas as técnicas apresentadas de codificação podem ser utilizadas em outros problemas iterativos, uma vez que os núcleos computacionais são bastante similares em muitos métodos iterativos. Foram apresentadas três técnicas de otimização da utilização da memória *cache* nas repetidas relaxações Gauss-Seidel com ordenação *red-black*, são elas: *Fusion Technique*, que atualiza cada elemento *black* assim que todos os pontos nodais *red* vizinhos a ele forem atualizados, isso garante que ao atualizar um *black*, os dados *reds* vizinhos ainda permanecerão em *cache*; *Blocking Technique*, que começa a próxima iteração Gauss-Seidel antes do fim da anterior, em outras palavras, é um avanço da técnica anterior, fazendo a segunda atualização do elemento *red* (2ª iteração) após os seus pontos nodais *black* vizinhos terem sido atualizados; *Windshield Wiper Technique*, que, diferentemente das outras duas técnicas que visam uma melhor otimização dos níveis mais baixos da hierarquia de memória *cache* (por exemplo: *cache* L3), esta otimiza a melhor utilização da *cache* L1 e até dos registradores. O funcionamento da terceira técnica consiste em usar um pequeno bloco que é movido ao longo da malha, atualizando todos os nós contidos nele. A alta utilização dos registradores e da *cache* L1 é alcançada pelo maior número possível de atualizações dentro do

bloco, mas ainda respeitando todas as dependências de dados. Foi concluído que a aplicação dessas técnicas pode manter até 250 MFLOP/s (25% do pico de desempenho) independentemente do tamanho da malha, que seriam perdidos se não fosse levado em consideração a 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 dentro do mesmo processador.

Takahashi (2003) implementou o algoritmo FFT (*Fast Fourier Transform*) paralelo tridimensional para clusters de PCs. Ele reduziu 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. Ele obteve um ganho de 30% no tempo de processamento em relação a 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). O autor aplicou estas técnicas 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 e outras técnicas de distribuição de dados, permitindo aos programadores a avaliação das estratégias antes do desenvolvimento do código. Adicionalmente, o autor prevê como trabalho futuro o estudo de malhas não-estruturadas, implementações 3D do algoritmo de Gauss-Seidel e implementações 2D do método de Gauss-Seidel com coeficientes variáveis (problema semelhante ao apresentado neste trabalho).

As próximas seções descrevem a estratégia de utilização mais eficiente da *cache* empregada neste trabalho. Um algoritmo de decomposição de domínio é empregado. Essa decomposição em dois níveis é similar ao proposto Gullerud e Dodds (2001), 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 será efetuada através da

execução de mais de um processo MPI por CPU. A Seção 4.3.3.1 apresenta o cálculo teórico do tamanho dos blocos utilizados para melhor desempenho. A Seção 4.3.3.2 discute taxa de *cache miss* em problemas maiores que o tamanho da memória *cache*, enquanto a Seção 4.3.3.3 discute a estratégia de utilização de "mais de um processo em um único processador". A Seção 4.3.3.4 analisa a eficiência da técnica. Finalmente, a Seção 4.3.3.5 apresenta um procedimento para uma estimativa do número ideal de processos por processador.

4.3.3.1 Cálculo do tamanho máximo do problema

A subrotina SOR do algoritmo é a que demanda maior esforço computacional, além de lidar com a maior parte da memória alocada para o processamento. Por essa razão ela será estudada mais detalhadamente.

A equação discretizada para cada ponto foi descrita na Equação (9) (página 22), onde os valores *P* são as incógnitas (ou vetor solução) e os coeficientes *Aw*, *Ae*, *As*, *An* e *B* são os coeficientes do sistema linear de equações, conforme descrito no Capítulo 2. O Código 4.1 (abaixo) apresenta a implementação da rotina de SOR do método de Gauss-Seidel em linguagem C (linguagem de programação utilizada neste trabalho).

```
void SOR()
{
        <outros commandos>

        j_min = 1;
        j_max = M-1
        i_min = 1;
        i_max = N-1;

        for (j=j_min; j<=j_max; j++)
        {
                for (i=i_min; i<=i_max; i++)
                {
                        temp =
                        ( A(j,i).Aw * P(j,i-1)
                        + A(j,i).Ae * P(j,i+1)
                        + A(j,i).As * P(j-1,i)
                        + A(j,i).An * P(j+1,i)
                        - B(j,i)
                        ) / A(j,i).Ap;

                        P(j,i) = (1-W) * P(j,i) + W * temp;
                }
        }

        <outros comandos>
}
```

Código 4.1: Trecho principal do procedimento SOR escrito na linguagem C.

A subrotina SOR é executada diversas vezes até que se tenha alcançado a convergência. A verificação dessa convergência é testada pela função ERROSOR, que acessa praticamente a

mesma região de memória que o SOR, e é a segunda na lista das rotinas que demandam mais computação.

Um dos objetivos deste trabalho é fazer com que haja o mínimo de *cache misses* no nível L2. Então, é necessário saber o quanto de espaço de memória é utilizado na subrotina SOR. Para armazenar a matriz de elementos do problema simulado é utilizado um vetor de memória linear, onde a segunda linha da matriz é armazenada no vetor após a primeira linha, a terceira linha após a segunda e assim sucessivamente (Figura 4.8).

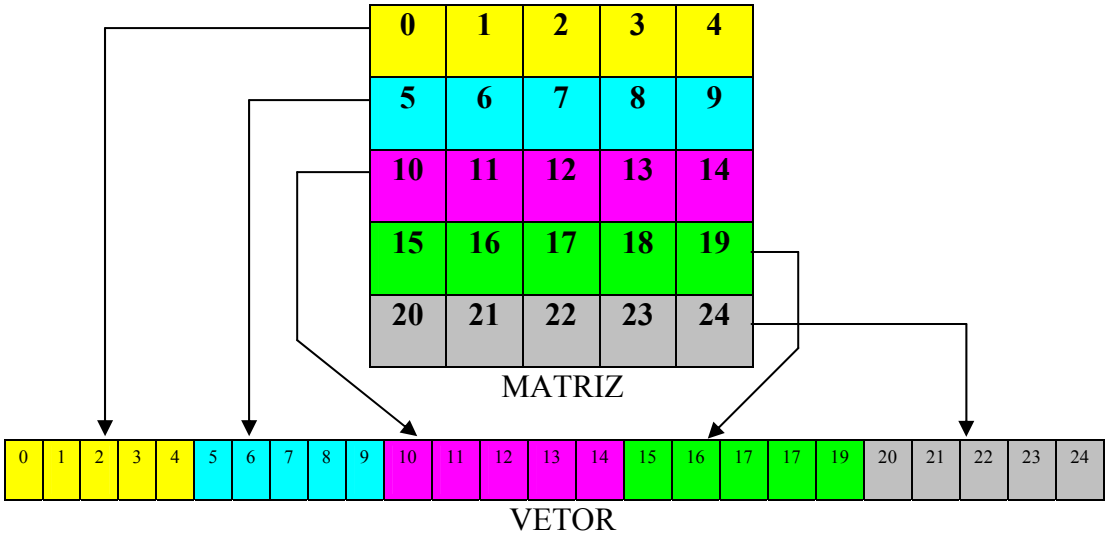


Figura 4.8: Armazenamento de uma matriz em um vetor. Em uma matriz A com M linhas e N colunas, o elemento A(i, j) é representado pelo no vetor V por V(i.M + j), onde i é a linha e j é a coluna da matriz.

Como pode ser visto nas linhas de código acima, são acessados três vetores, P, A e B, sendo que o vetor A é dividido em cinco partes (A_p, A_n, A_s, A_w, A_e), totalizando, assim, sete variáveis. Portanto, é necessário que existam sete variáveis para cada ponto nodal. A malha 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 outros subdomínios e seus valores não são atualizados pelo SOR.

Os vetores P, A e B, utilizados pelo SOR, são dimensionados para comportar os $N \times M$ pontos nodais, mas somente o vetor P é acessado na totalidade, inclusive nas fronteiras, devido ao fato de o cálculo do novo P ser influenciado pelos valores de P dos pontos nodais vizinhos, dessa forma, acessando também as fronteiras. Os vetores A e B são acessados somente na região útil, não considerando as fronteiras.

Ao acessar algum dado que não esteja na memória *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, que nas máquinas Athlon do *cluster* possuem 64 bytes. 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 64bits (8 bytes) cada uma. Então, do vetor *P* são acessados $8\times M\times N$ bytes, do vetor *B*, por não acessar a primeira e última linha, são acessados $8\times(M-2)\times N$ bytes. Do vetor *A*, que é composto por cinco partes e cuja primeira e última linha não são acessadas, são acessados $8\times5\times(M-2)\times N$ bytes. Portanto, o total de memória (vetores *P*, *A* e *B*) acessada pelo procedimento SOR é dado pela expressão abaixo:

$$Acessado = 8\times\left[M\times N + 6\times\left(M-2\right)\times N\right] \qquad (26)$$

Nas máquinas do cluster, o nível 2 da memória *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.

A Tabela 4.1 mostra a quantidade de espaço utilizado pelos vetores do SOR. Os valores *T* e *H* são, respectivamente, os valores da largura e altura da matriz de elementos, considerando apenas os elementos úteis (sem os contornos). Portanto $T=N-2$ e $H=M-2$.

De acordo com a tabela, para processar o problema da cavidade garantindo que todos os dados utilizados para o processamento do SOR estejam em L2, é necessário que $T\times H$ seja menor 4096 pontos nodais (blocos menores que 64×64).

Tabela 4.1: Utilização de memória em função do tamanho da matriz de elementos. *T* e *H* são as quantidade de pontos nodais úteis (sem considerar as bordas) na largura e altura da matriz, respectivamente. *T*=*N*-2 e *H*=*M*-2.

T×*H*	Utilização total (Kbytes)	% de utilização da *cache* (256KB)
1.600	92,53	36,15%
2.304	132,03	51,57%
3.136	178,53	69,74%
4.096	232,03	90,64%
16.384	912,03	356,26%
65.536	3.616,03	1.412,51%
262.144	14.400,03	5.625,01%
1.048.576	57.472,03	22.450,01%

4.3.3.2 Taxa de *cache miss* em problemas maiores

O Gráfico 4.5 mostra a taxa de *cache miss* em função do tamanho do problema. Pode-se notar que a taxa de *cache miss* começa a crescer após o tamanho 64×64 (4096 pontos nodais).

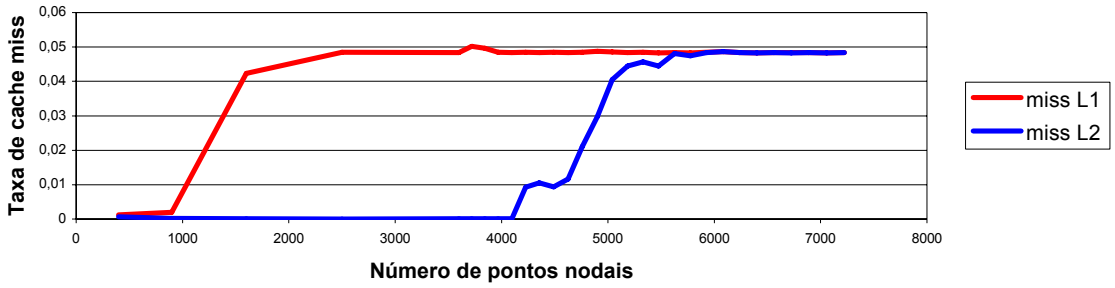


Gráfico 4.5: Variação da taxa de *cache miss* de leitura de dados em função do número de pontos nodais úteis da matriz. A *cache* L1, devido ao seu tamanho reduzido em relação ao L2, tem um número elevado de *cache misses* mesmo em problemas pequenos.

A Tabela 4.2 mostra, com detalhe, o rápido crescimento da taxa de *cache miss* com um pequeno acréscimo no tamanho do problema. Essa tabela foi construída a partir de dados gerados pelo software Valgrind (descrito na Seção 3.3.1.1).

Conforme explicado anteriormente, os procedimentos SOR e ERROSOR são os que demandam maior esforço computacional. Por isso, influenciam fortemente a taxa de *cache miss* global e serão estudados mais detalhadamente.

Tabela 4.2: Quantidade de leituras, quantidade de *cache misses* de leitura em L2 e a taxa de *cache miss* de leitura em L2 para o problema da cavidade.

T	*H*	*T*×*H*	Leitura de dados	*Cache miss*es de leitura L2	Taxa de *cache miss* de leitura L2
60	60	3.600	196.984.588	12.533	0,00636%
61	61	3.721	209.352.460	13.575	0,00648%
62	62	3.844	222.209.126	14.502	0,00653%
63	63	3.969	236.446.220	14.929	0,00631%
64	64	4.096	250.791.698	15.338	0,00612%
65	65	4.225	266.621.558	2.475.725	0,92855%
66	66	4.356	283.064.606	2.975.397	1,05114%
67	67	4.489	300.630.987	2.801.917	0,93201%
68	68	4.624	319.889.871	3.730.706	1,16625%
69	69	4.761	339.886.988	7.169.146	2,10927%
70	70	4.900	360.635.200	10.751.663	2,98131%
71	71	5.041	383.267.133	15.520.876	4,04962%
72	72	5.184	406.164.528	18.067.953	4,44843%
73	73	5.329	431.659.379	19.697.640	4,56324%
74	74	5.476	456.874.332	20.308.724	4,44514%
75	75	5.625	484.844.890	23.328.439	4,81153%

A taxa de *cache miss* de dados pode ser calculada tanto para leitura quanto para escrita de dados. Elas são dadas pelas expressões:

$$Taxa\ de\ miss\ de\ escrita = \frac{número\ de\ misses\ ao\ escrever}{número\ de\ escritas\ de\ dados} \tag{27}$$

$$Taxa\ de\ miss\ de\ leitura = \frac{número\ de\ misses\ ao\ ler}{número\ de\ leituras\ de\ dados} \tag{28}$$

A taxa de *cache miss* de escrita é zero devido ao simples fato da escrita de *P*(i, j) ser precedida da leitura de seu próprio valor antigo. Assim, uma cópia da região de memória destino da escrita já está na memória *cache*. A escrita de *P*(i, j) 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. Portanto, os procedimentos SOR e cálculo de resíduo não apresentam taxa de *cache miss* de escrita.

A taxa de *cache miss* de leitura pode ser quantificada teoricamente. 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 memória *cache*. Mas 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 na memória *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*.

De acordo com o Gráfico 4.5, independentemente do tamanho do problema (nos casos maiores), 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. Este valor limite de 5% pode ser estimado através da análise do Código 4.1. Ao calcular o novo valor da pressão em um determinado elemento da matriz, é necessário buscar na memória os valores de B, A_p, A_n, A_s, A_w, A_e e do próprio P referente ao mesmo elemento, além dos valores de P nos pontos nodais vizinhos.

A leitura de um valor de B pode ocasionar um *cache miss*, fazendo com que a *cache* busque o dado da memória principal da máquina. Como foi mencionada, a leitura é feita em blocos de 64 bytes e, supondo que o elemento B lido ocupe os primeiros 8 bytes (*double*) do bloco, significa que no cálculo dos próximos sete valores de pressão não ocorrerá *cache miss* na leitura de B, visto que a leitura é feita percorrendo primeiramente em uma linha para, em seguida, avançar para a próxima. Em outras palavras, para cada passo de execução do SOR, o vetor B é acessado por um todo, exceto a primeira linha e última linha que são as fronteiras superiores e inferiores. Normalmente, ocorrerá *cache miss* ao acessar um elemento de B quando este ocupar o primeiro elemento de um bloco de *cache*, a exceção ocorre ao acessar o primeiro elemento da linha da matriz, visto que os pontos nodais das fronteiras laterais não são acessados em se tratando do vetor B. Então, para o vetor B, ocorre aproximadamente 1/8 de *cache misses* ao percorrê-lo, totalizando $(M\text{-}2)\times N/8$ *cache misses*.

A consideração a ser feita para o vetor A é semelhante a do vetor B, sendo a única diferença o fato do vetor A ser subdividido em cinco partes (A_p, A_n, A_s, A_w, A_e). Devido às cinco subdivisões do vetor A, o seu tamanho é 5 vezes maior que o vetor B, consequentemente 5 vezes mais *cache misses*, totalizando $5\times(M\text{-}2)\times N/8$ *cache misses*.

O vetor P tem uma particularidade: cada elemento da matriz pode ser acessado até cinco vezes, quando é recalculado o seu próprio valor e quando são recalculadas as pressões dos seus quatro vizinhos (quando esse não for fronteira). Ou seja, supondo um elemento P(i, j) na

matriz, sendo i a coluna e j a linha, esse elemento é primeiramente acessado quando é calculado o elemento *P*(i, j-1), pois este requer os valores de seus quatro vizinhos: *P*(i, j-2), *P*(i-1, j-1), *P*(i+1, j-1) e *P*(i, j). O primeiro acesso a *P*(i, j) possivelmente ocasionará *cache miss* nos problemas maiores. O segundo acesso a esse elemento é quando *P*(i-1, j) é calculado, o que da mesma forma necessita de seus vizinhos, inclusive *P*(i, j). Nesse segundo acesso, o dado de *P*(i, j) já estará em *cache*, exceto se o problema for tão grande que as variáveis de duas linhas da matriz não caibam na memória *cache* L2. O terceiro acesso ocorre na atualização do próprio *P*(i, j). O quarto acesso ocorre no cálculo de *P*(i+1, j), e, por fim, no quinto acesso é feito cálculo de *P*(i, j+1). Então, cada elemento do vetor *P* é acessado cinco vezes, mas são carregados apenas uma vez da memória principal para a *cache* L2 em cada execução do SOR, a menos que o problema seja tão grande que as variáveis dos pontos nodais de duas linhas não caibam na *cache* L2. O total de *cache miss* para ler *P* será de $M \times N/8$ *cache misses*, que também leva em consideração as fronteiras superior e inferior. Na integra, o procedimento SOR terá:

$$L2d_{mrd} = \frac{M \times N + 6 \times (M-2) \times N}{8} \qquad (29)$$

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 ainda, o fato de as instruções e as outras variáveis do procedimento ocuparem algum espaço no L2, entre outros fatores.

Para calcular o número total de acessos de leitura realizado pelo procedimento SOR, foi examinado o código em C do algoritmo traduzido para assembly (Código 4.2). Essas instruções foram geradas pelo compilador gcc do Cluster Enterprise (Seção 3.2.1) com otimização nível O3. As instruções destacadas com [R] fazem leitura de dados e as destacadas com [W] fazem escrita de dados e as com [RW] fazem tanto leitura como escrita. As instruções no *loop* mais interno são repetidas $(M\text{-}2)\times(N\text{-}2)$ vezes, pois não processam os pontos nodais das fronteiras. Das instruções que compõe o *loop* mais interno, 18 fazem leitura de dado, totalizando $18\times(M\text{-}2)\times(N\text{-}2)$ acessos. Por sua vez, o *loop* mais externo é repetido $(M\text{-}2)$ vezes, possuindo 12 instruções de leitura (não considerando as instruções dentro do *loop* interno), totalizando $12\times(M\text{-}2)$. Além desses acessos, o procedimento SOR faz 32 acessos de leitura com outros propósitos como, por exemplo, inicialização de variáveis de controle do procedimento (essas instruções não foram listadas no código).

```
.L803:
            movl    -24(%ebp), %esi         [R]
            cmpl    -28(%ebp), %esi         [R]
            ja      .L813
            movl    -16(%ebp), %edx         [R]
            movl    -56(%ebp), %eax         [R]
            decl    %edx
            imull   %eax, %edx
            movl    %eax, -40(%ebp)         [W]
            movl    -16(%ebp), %eax         [R]
            incl    %eax
            movl    %edx, -44(%ebp)         [W]
            imull   -40(%ebp), %eax         [R]
            movl    -68(%ebp), %edx         [R]
            movl    -60(%ebp), %edi         [R]
            movl    -64(%ebp), %ebx         [R]
            movl    %eax, -48(%ebp)         [W]
            movl    %edx, -52(%ebp)         [W]
            fld     %st(0)
.p2align 2
.L807:
            movl    -16(%ebp), %ecx         [R]
            imull   -40(%ebp), %ecx         [R]
            addl    %esi, %ecx
            leal    (%ecx,%ecx,4), %edx
            sall    $3, %edx
            movl    -44(%ebp), %eax         [R]
            fldl    -8(%ebx,%ecx,8)         [R]
            fldl    8(%ebx,%ecx,8)          [R]
            addl    %esi, %eax
            fmull   32(%edi,%edx)           [R]
            fxch    %st(1)
            fmull   24(%edi,%edx)           [R]
            faddp   %st, %st(1)
            fldl    (%ebx,%eax,8)           [R]
            movl    -48(%ebp), %eax         [R]
            addl    %esi, %eax
            fmull   16(%edi,%edx)           [R]
            faddp   %st, %st(1)
            fldl    (%ebx,%eax,8)           [R]
            fmull   8(%edi,%edx)            [R]
            faddp   %st, %st(1)
            movl    -52(%ebp), %eax         [R]
            fldl    W                       [R]
            fxch    %st(1)
            fsubl   (%eax,%ecx,8)           [R]
            fld     %st(2)
            fxch    %st(1)
            fdivl   (%edi,%edx)             [R]
            fxch    %st(1)
            fsub    %st(2), %st
            fmull   (%ebx,%ecx,8)           [R]
            fxch    %st(2)
            fmulp   %st, %st(1)
            faddp   %st, %st(1)
            fstpl   (%ebx,%ecx,8)           [W]
            incl    %esi
            cmpl    -28(%ebp), %esi         [R]
            jbe     .L807
            fstp    %st(0)
.L813:
            incl    -16(%ebp)               [RW]
            movl    -32(%ebp), %edx         [R]
            cmpl    %edx, -16(%ebp)         [R]
            jbe     .L803
```

Código 4.2: Trecho principal do procedimento SOR convertido em assembly, destacando as instruções de leitura [R], gravação [W] e leitura e gravação [RW] de dados.

O número de leitura de dados em uma iteração do SOR é dado por:

$$D_{rd} = 18 \times (M-2) \times (N-2) + 12 \times (M-2) + 32 \quad (30)$$

onde D_{rd} é o número de leituras de dados[1].

A taxa de *cache miss* de leitura em L2 no procedimento SOR é dada pela razão entre as Equações (29) e (30):

$$taxa\ L2d_{mrd} = \frac{M \times N + 6 \times (M-2) \times N}{8 \times [18 \times (M-2) \times (N-2) + 12 \times (M-2) + 32]} \quad (31)$$

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 e a experimental (utilizando o Valgrind) é apresentada na Tabela 4.3. 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.

Tabela 4.3: Comparativo entre as taxas de *cache misses* teóricas e as coletadas experimentalmente utilizando o software valgrind.

			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%

[1] 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. E mesmo o código em Assembly examinado não é necessariamente igual ao código binário gerado pelo compilador já que o compilador faz otimizações na tradução do código em Assembly para linguagem de máquina.

4.3.3.3 Mais de um processo em um único processador

Conforme citado 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 será efetuada através da execução de mais de um processo MPI por CPU. Esta seção apresenta uma breve avaliação do efeito desta estratégia sobre a taxa de *cache miss*.

Conforme apresentado na seção anterior, para as simulações com até 4096 pontos nodais a taxa de *cache miss* é aproximadamente zero, e para um problema um pouco maior, a taxa atinge seu máximo. A idéia deste método é transformar problemas maiores em problemas suficientemente pequenos para que a taxa de *cache miss* 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 a 4096 pontos nodais. É importante garantir que o número de partes seja múltiplo do número de processadores para balanceamento de carga entre os processadores.

O Gráfico 4.6 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, como já foi mencionado, mede a taxa de *cache miss* sem levar em consideração se existe um outro processo em execução. Tendo em vista 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. Portanto, no Gráfico 4.6, 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.

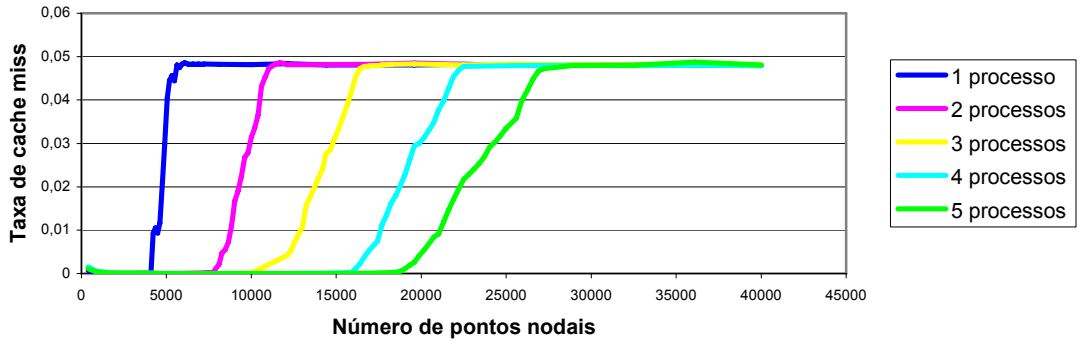


Gráfico 4.6: Taxa de *cache miss* em função do número de pontos nodais variando o número de processos em um único processador.

Será mostrado na próxima seção que entre uma troca e outra de contexto, ou seja, quando um outro processo adquire a CPU e o estado do processo antigo é salvo e o estado do novo processo é recuperado, um processo executa pelo menos 33 iterações do procedimento SOR. Por essa razão, a taxa de *cache miss* não é significativamente maior do que 5%, como nos demais casos.

4.3.3.4 Análise da Eficácia da Técnica

Na Seção 4.3.3.2 foi discutido o aumento das falhas na *cache* em função do incremento do número de pontos nodais. Em conseqüência desse aumento, a velocidade de processamento da máquina diminui consideravelmente devido a ociosidade do processador aguardando pelas informações que não estão em *cache*. Esta seção apresenta 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*. Adicionalmente, 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.

4.3.3.4.1 Execução em apenas um processador

O Gráfico 4.7 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% (Gráfico 4.6).

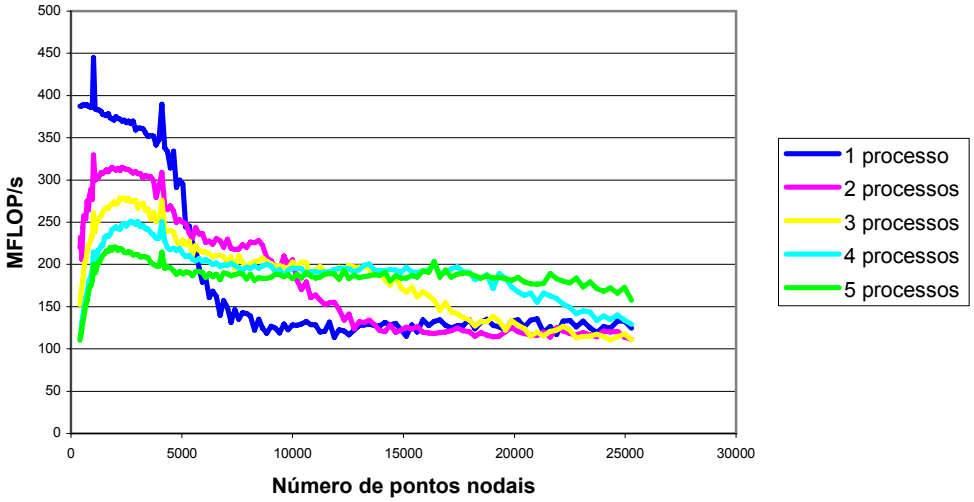


Gráfico 4.7: Número total de operações de ponto flutuante por segundo em função do número de pontos nodais.

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.

Entre 5184 e 9216 pontos nodais, os exemplos com 2, 3, 4 e 5 processos executados no mesmo processador possuem velocidade de processamento maior que no caso serial (1 processo). 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 comunicações.

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.

Uma questão relevante ainda por ser abordada é o efeito da troca de contexto entre os processos no mesmo processador. De acordo com o Gráfico 4.6, quando os exemplos são comportados pela *cache* L2, foi medido um desempenho de pelo menos 175MFLOP/s, sendo que a troca de contexto entre os processos do cluster Enterprise ocorre, no mínimo, a cada 10ms (OLIVEIRA; CARISSINI; TOSCANI, 2001), então cada processo executa pelo menos 1,75 milhões de cálculos de ponto flutuante antes da troca de contexto. Conforme explicado anteriormente, um processo não pode ultrapassar os 4096 pontos nodais para caber na *cache* L2, e o procedimento SOR faz 13 operações de ponto flutuante para cada elemento. Então, em um passo do SOR são realizadas no máximo 53248 operações. Sendo assim, antes de cada troca de contexto, o processador pode executar mais de 33 iterações SOR.

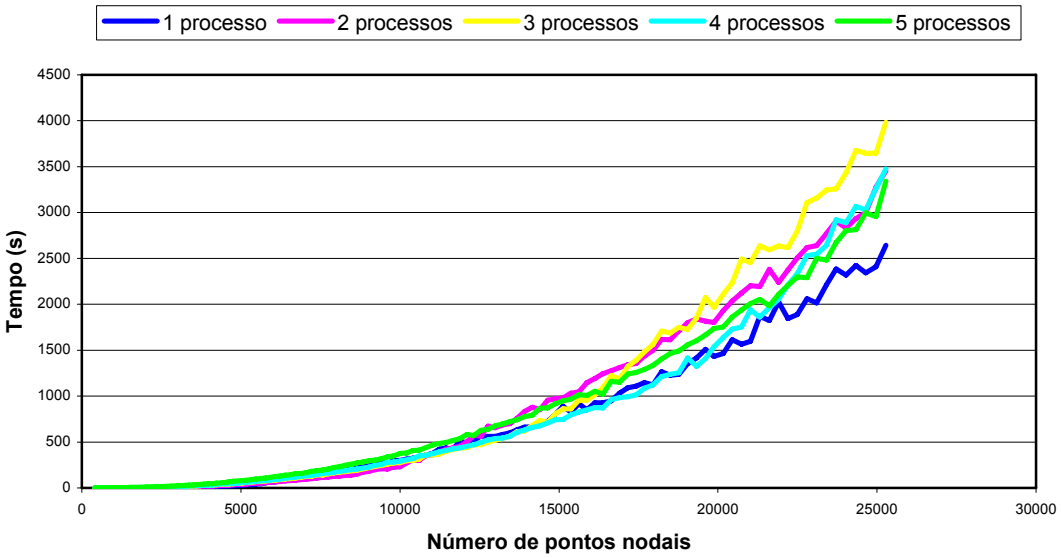


Gráfico 4.8: Tempo de processamento em função do número de pontos nodais. Apresenta o tempo para um processo serial, e para processos paralelos executados em uma única máquina.

O Gráfico 4.8 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 o Gráfico 4.7), 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.

O Gráfico 4.9 mostra o aumento da velocidade de execução dos exemplos em relação ao exemplo serial (*Speedup* = $T_{serial}/T_{n\ processos}$). Pode-se observar, que quanto mais processos em um mesmo processador, menor será o ganho, mesmo quando a memória *cache* L2 é utilizada eficientemente. A degradação é causada principalmente pela grande quantidade de comunicação entre processos, mesmo sendo no mesmo processador, e pelo retardo na convergência. É importante notar que o exemplo com cinco processos, em momento nenhum, é mais rápido que o serial. Por outro lado, o exemplo com dois processos pode reduzir em até 33% o tempo do serial (tempo serial igual a 1,5 vezes o tempo com dois processos).

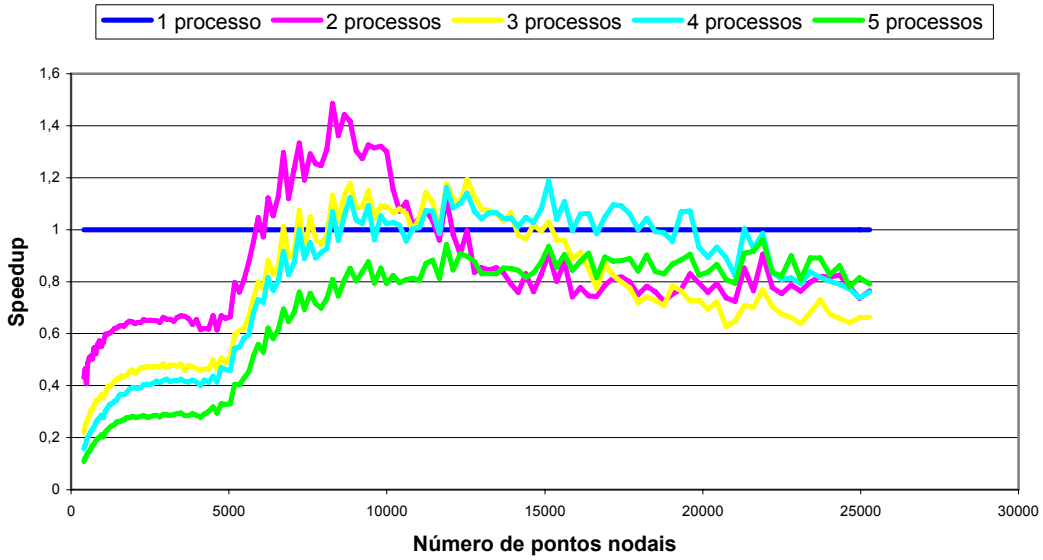


Gráfico 4.9: Taxa de ganho no tempo de processamento em relação ao processo serial, para os diversos tamanhos de problemas.

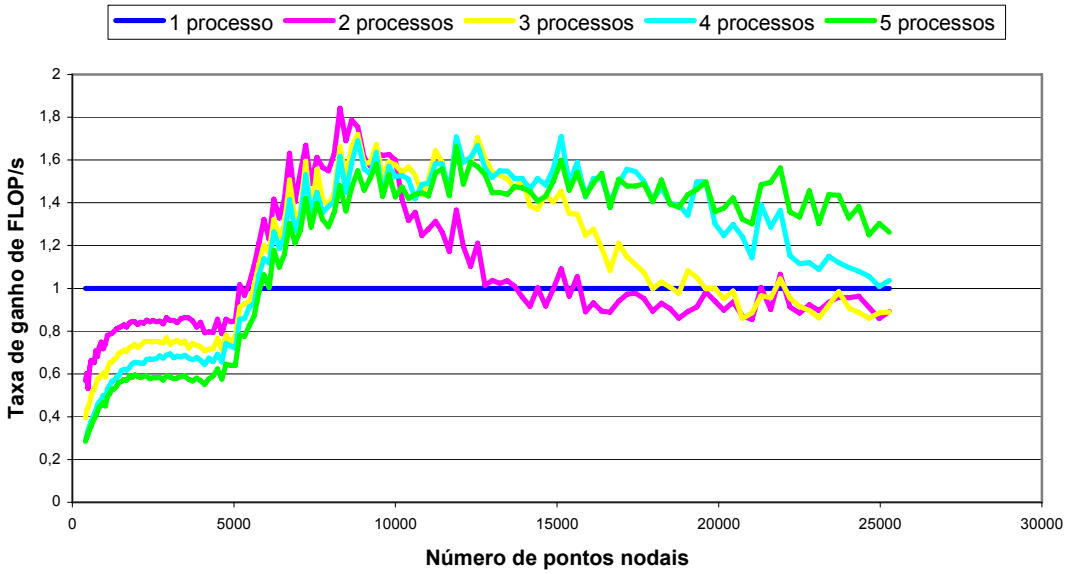


Gráfico 4.10: Taxa de ganho na capacidade de cálculos (FLOP/s) em relação ao processo serial, para os diversos tamanhos de problemas.

O Gráfico 4.10 mostra o ganho na velocidade de cálculos (FLOP/s) devido à divisão em processos num mesmo processador ($ganhoFLOP/s = FLOP/s_{n\ processos}/FLOP/s_{serial}$). Com esse gráfico e o Gráfico 4.9 é possível concluir que um grande responsável pela não obtenção de eficiência quando se divide o problema em vários processos em um mesmo processador é o número maior de iterações SOR para atingir a convergência. Com o aumento do número de processos ocorre o aumento do volume de comunicação, mas este não degrada consideravelmente o número de FLOP/s, devido a maior eficiência do uso da *cache*. Contudo este aumento do número de FLOP/s não é suficiente para compensar o aumento do número de iterações devidas à decomposição do domínio.

O efeito do aumento do número de processos na quantidade total de iterações pode ser observado no Gráfico 4.11.

4.3.3.4.2 Execução em diversos processadores

Conforme descrito anteriormente, quando o número de processos em um único processador aumenta, o rendimento é degradado pelo excesso de comunicação entre os processos e pela demora em atingir a convergência devido ao número maior de iterações SOR. Para discutir esse efeito, a seguir estão os resultados de experimentos onde foi variado o número de processos em cada processador.

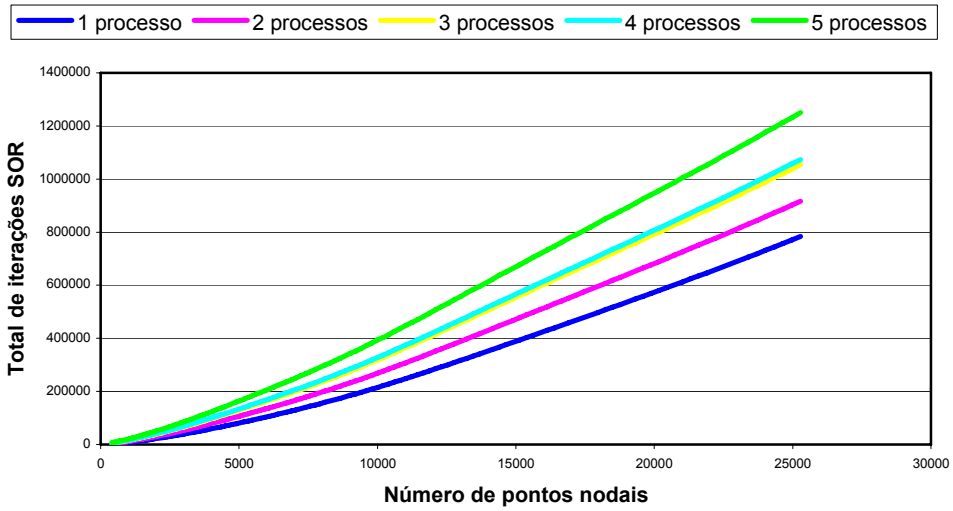


Gráfico 4.11: Quantidade total de iterações SOR para executar simulações de diversos tamanhos utilizando de 1 a 5 processos em um único processador.

Um processo com aproximadamente 4096 pontos nodais ocupa em torno de 88% da memória *cache* L2. Os resultados desses experimentos foram comparados com os resultados da mesma configuração (mesmo número de processadores executando uma simulação com a mesma malha computacional), utilizando um único processo por processador. Esse teste foi efetuado utilizando 2, 3, 4, 6 e 9 processos por processador, para simulações empregando 1, 2, 3, 4, 6 e 9 processadores.

O Gráfico 4.12 mostra uma comparação de desempenho entre as simulações utilizando um processo por processador e mais de um processo por processador para simulações com 2, 3, 4, 6 e 9 processos por processador (Gráfico 4.12a, b, c, d, e, respectivamente). As curvas apresentadas no Gráfico 4.12 representam: (*i*) a razão do tempo de processamento obtido ao utilizar um processo por processador e o tempo de processamento obtido ao utilizar mais de um processo por processador, (ii) a razão do número de FLOP/s obtido ao utilizar um processo por processador e o número de FLOP/s obtido ao utilizar mais de um processo por processador e (iii) a razão do número de iterações SOR obtido ao utilizar um processo por processador e o número de iterações SOR obtido ao utilizar mais de um processo por processador.

O bom rendimento mostrado no Gráfico 4.12a se deve ao fato de que, com dois processos por processador, cada processo executa o procedimento SOR com os dados totalmente na *cache* L2, enquanto que para a simulação com apenas um processo por processador isto não acontece. O benefício adquirido pela melhor utilização da *cache*, que pode ser observado nas simulações utilizando de 1 a 9 processadores, ainda é maior que a degradação causada pelo

excesso de comunicação e de iterações SOR. Por outro lado, quando são utilizados mais processos em um mesmo processador, esse benefício tende a se tornar cada vez menor.

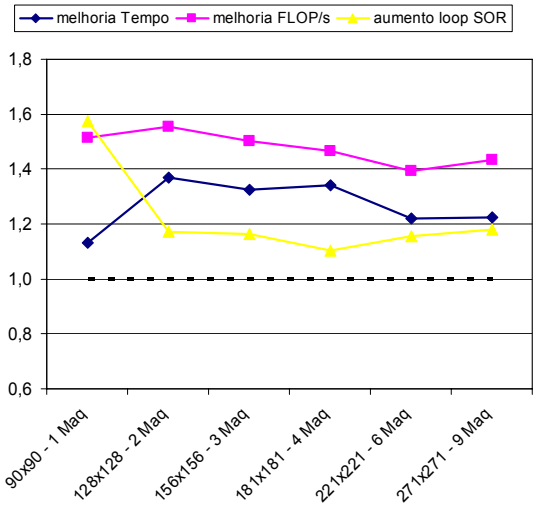
O Gráfico 4.12b mostra o ganho que se pode obter ao executar três processos em um mesmo processador. É importante ressaltar que cada um dos três processos em cada processador não pode ultrapassar os 4096 pontos nodais. Mesmo com um volume maior de comunicação, o experimento com três processos por processador executou com uma quantidade de FLOP/s maior do que com apenas um processo por processador, mas não tão grande quanto no caso anterior (com dois processos por processador). Outro aspecto relevante é o aumento do número de iterações SOR, que se tornou ainda maior comparado com o experimento anterior com dois processos por processador.

Os dois fatores, capacidade menor de FLOP/s e número maior de iterações SOR, são os responsáveis pelo aumento no tempo de processamento do problema. A mesma tendência pode ser observada nos outros experimentos: com quatro (Gráfico 4.12c), cinco (Gráfico 4.12d), seis (Gráfico 4.12e) e nove (Gráfico 4.12f) processos por processador. Nesses experimentos, o número de iterações SOR cresceu ligeiramente em relação aos experimentos com um número menor de processos por processador.

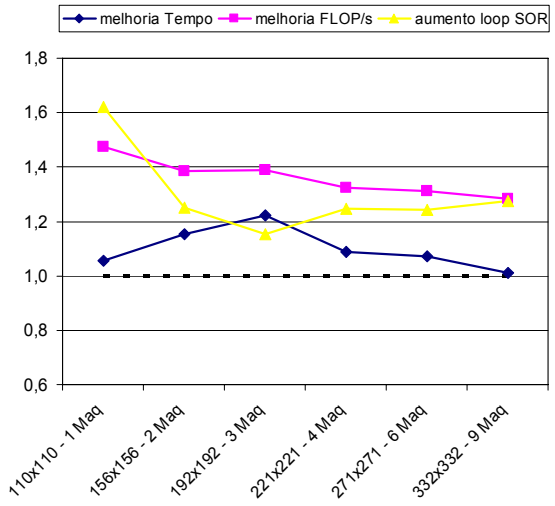
Todavia, ao contrário do esperado, no experimento com quatro processos por processador, em alguns casos, a quantidade de FLOP/s subiu com o aumento do número de processos por processador, mesmo com um número maior de comunicações comparado com o experimento com três processos por processador. Isso pode ocorrer devido ao fato de que as dimensões da matriz dos elementos em cada processo estejam desproporcionais (longe da forma quadrada sugerida na Seção 2.3).

No Gráfico 4.12b, pode-se notar que o pior rendimento foi no exemplo com a matriz de 332×332 utilizando 9 máquinas (processadores). A possível explicação desse fato é que, no modo convencional de paralelização (com um processo por processador), serão 9 processos, que poderão ser divididos em uma matriz de 3×3 blocos. Como a matriz de elementos é quadrada, nesse caso, cada subdomínio também será quadrado (aproximadamente 110×110 pontos nodais). No caso da utilização de três processos por processador, pois serão 9 máquinas com 3 processos cada, totalizando 27 processos, que podem ser divididos no melhor caso em 9×3 blocos, que mesmo assim é uma configuração ruim. Cada bloco possuirá aproximadamente 37×110 pontos nodais, o que é bastante diferente de uma forma quadrada, que minimiza as fronteiras de comunicação.

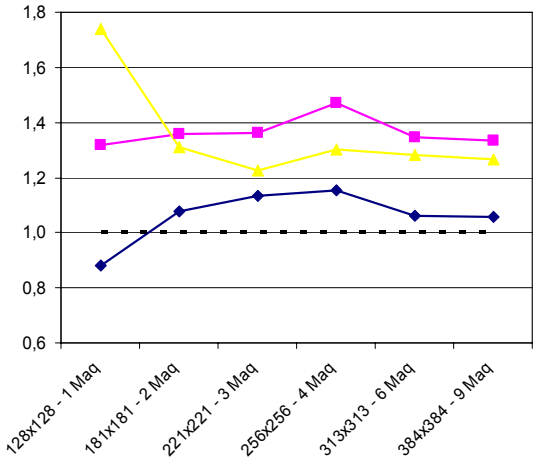
(a) 2 processos por processador



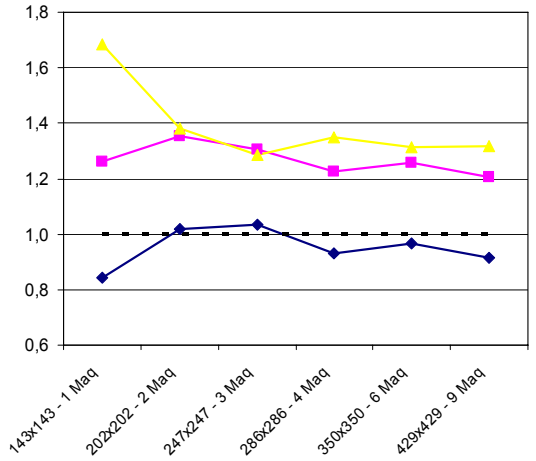
(b) 3 processos por processador



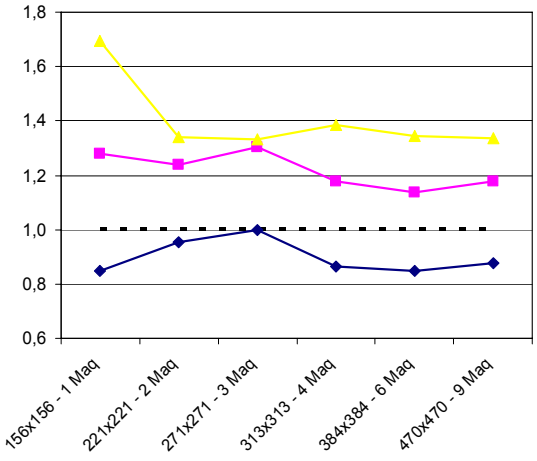
(c) 4 processos por processador



(d) 5 processos por processador



(e) 6 processos por processador



(f) 9 processos por processador

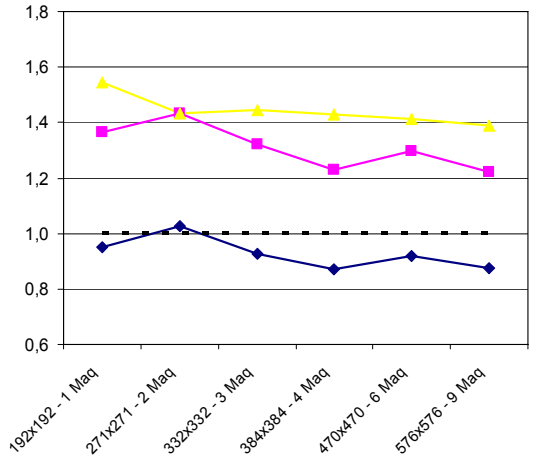


Gráfico 4.12: Comparação do aumento da velocidade de processamento (*Speedup)*, do aumento da quantidade de FLOP/s e do aumento do número de iterações SOR entre executar o experimento da cavidade com (a) dois, (b) três, (c) quatro, (d) cinco, (e) seis e (f) nove processos por processador e o mesmo experimento na forma tradicional (um processo por processador).

Ainda no Gráfico 4.12b, o melhor *speedup* foi com a malha 192×192 utilizando 3 processadores, pois ao utilizar a técnica, são 9 processos (3×3 blocos), sendo todos formados por 64×64 pontos nodais. Enquanto que sem a técnica são apenas 3 processos com formação de 64×192 pontos nodais. Por isso a alta eficiência foi alcançada.

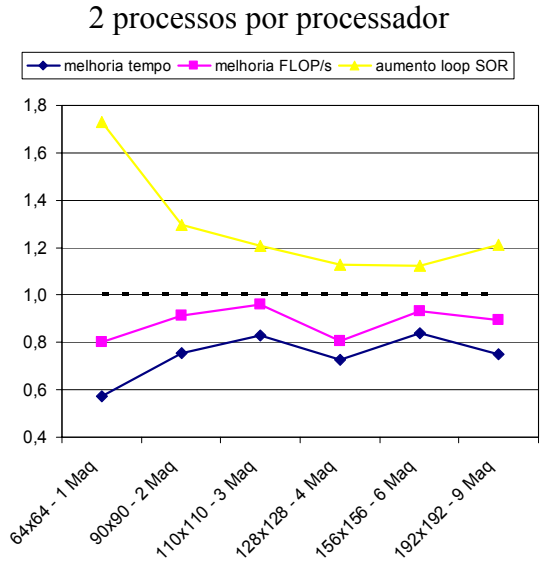


Gráfico 4.13: Comparação do aumento da velocidade de processamento (*Speedup)*, do aumento da quantidade de FLOP/s e do aumento do número de iterações SOR entre executar o experimento da cavidade com dois processos por processador sendo que, a utilização de um único processo por processador já é suportado pela *cache*.

A técnica de divisão em um número maior de processos por processador mostrada neste trabalho, só faz sentido se os blocos destinados aos processadores não couberem em *cache*, e a sua divisão em sub-blocos couberem. Dessa forma, cada um dos sub-blocos no processador, individualmente, será comportado pela *cache*. Por outro lado, se fosse mantido apenas um bloco no processador, este não caberia na *cache*. O Gráfico 4.13 mostra qual a perda ao dividir um problema onde cada subdomínio (bloco) já é suportado pela *cache*. Ao dividir cada processo em dois, pode-se notar que a quantidade de FLOP/s diminui devido ao acréscimo de comunicações. Ao contrário dos experimentos anteriores, não há benefício na melhor utilização da memória *cache*, uma vez que, o experimento com um processo por processador já é suportado na *cache*.

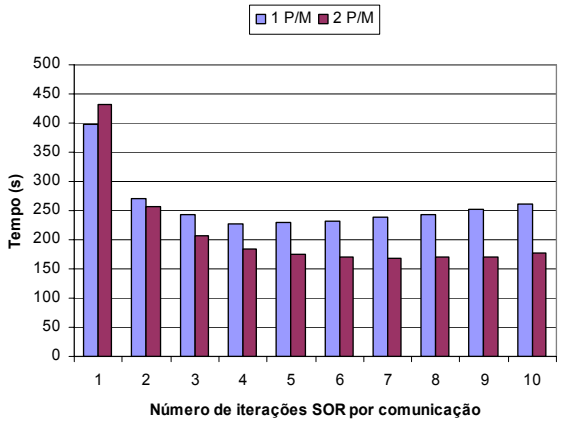
Aumentar o número de processos em um mesmo processador pode proporcionar uma redução da taxa de *cache miss*, em conseqüência pode diminuir o tempo de execução da simulação. Além disso, outro aspecto importante, mas que não é o foco deste trabalho, é que ao se utilizar mais de um processo em um único processador pode fazer com que o processador seja mais utilizado. Em outras palavras, o processador ficará ocioso se possuir apenas um processo e este estiver aguardando por informações de um outro processador. Por outro lado, se houvesse

mais de um processo, este outro poderia realizar computação enquanto o primeiro está aguardando a comunicação.

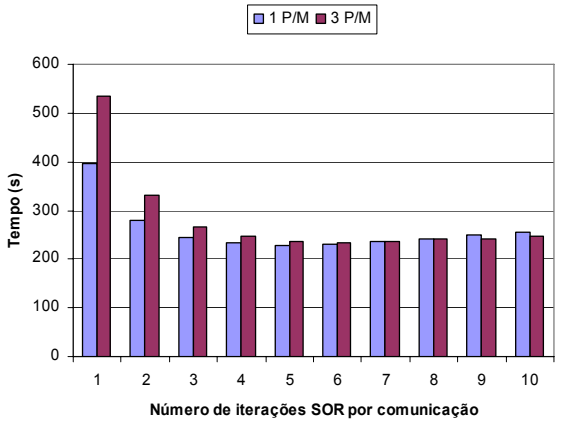
4.3.3.4.3 Quantidade de iterações entre comunicações

Foi apresentado na Seção 4.3.1 a quantidade de iterações SOR entre comunicações para um problema sem a utilização da técnica de exploração da *cache*. A conclusão foi que utilizar um número muito baixo ou muito alto de iterações por comunicação é desaconselhável. Primeiro, devido ao grande intervalo de tempo necessário para as tarefas de comunicação, reduzindo a parcela de tempo relacionada ao "processamento útil" e deixando o processador ocioso. Segundo, devido ao excesso de iterações utilizando dados de fronteiras desatualizados e provocando processamentos desnecessários.

Quando é utilizada a técnica de mais de um processo por processador e o número de iterações entre comunicações for muito baixo (por exemplo, apenas uma iteração entre as comunicações), não haverá nenhum benefício, pois não serão reutilizados os dados na *cache*. Neste caso, o fluxo de execução do programa deixa a rotina SOR, e passa para o envio das informações das fronteiras, aguardando o recebimento das informações provenientes de outro processo. Havendo mais de um processo por processador e o processo em questão aguardando a chegada de dados, ocorre a troca de contexto para um outro processo do processador. Os dados, que foram carregados na *cache* pelo primeiro processo, serão sobrepostos pelos dados do segundo processo, acarretando assim a não reutilização das informações em *cache*.



(a)



(b)

Gráfico 4.14: Tempo de processamento ao executar o problema da cavidade em função do número de iterações por comunicação: o primeiro (a) com tamanho de 362×362 pontos nodais e 16 processadores com 1 e 2 processos por processador; o segundo (b) com tamanho de 332×332 pontos nodais e 9 processadores com 1 e 3 processos por processador.

O Gráfico 4.14 mostra como o aumento do número de iterações entre comunicação aumenta a eficácia da técnica de reutilização dos dados em *cache* em relação à execução sem a técnica. O Gráfico 4.14a mostra uma comparação entre uma execução com 1 processo por máquina e outra com 2 processos por máquina, enquanto o Gráfico 4.14b mostra uma comparação entre uma execução com 1 processo por máquina e outra com 3 processos por máquina.

4.3.3.5 Escolha do número de processos por processador

Esta seção contém um roteiro de como deve ser utilizada a técnica apresentada neste trabalho. Independentemente do algoritmo MFC, existem alguns requisitos para a aplicação da técnica proposta de utilização eficiente da memória *cache*:

- o algoritmo deve ser iterativo como, por exemplo, o método SOR, uma vez que o foco da eficiência é a reutilização dos dados;
- 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 apresentará benefício 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;
- conhecer o tamanho máximo do problema para que se tenha uma taxa de *cache miss* muito baixa. Isso pode ser feito de duas formas:

  → **Determinação teórica:** conhecendo a quantidade de memória *cache* (por exemplo, a L2) e a quantidade de memória acessada e atualizada pela rotina iterativa do algoritmo, em função do tamanho do problema. É escolhido o maior tamanho do problema de forma que a utilização de memória não seja maior que o nível de *cache* escolhido (é necessário manter uma folga para o código e outras variáveis do programa);

  → **Determinação experimental:** utilizando algum software de medição da taxa de *cache miss* (por exemplo, o Valgrind). Avalia-se a taxa de *cache miss* variando o tamanho do problema até alcançar o maior tamanho onde a taxa ainda é baixa.

Com os requisitos verificados, é necessário conhecer o número de processos (t) no qual fique garantido que a taxa de *cache miss* em cada um deles seja muito baixa. Para isso, t deve satisfazer as duas condições:

- *t* deve ser um número inteiro igual ou maior que a razão entre a *Quantidade de Pontos Nodais* do problema a ser simulado (*QPN*) pela *Quantidade de Pontos Nodais Suportados pela Cache* (*QPNSC*), garantindo assim, que os dados caibam na *cache*;
- *t* 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.

Portanto:

$$t \geq \frac{QPN}{QPNSC} \quad e \quad t = k \cdot m \tag{32}$$

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. Foi apresentado 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.

Tomando como exemplo a cavidade com uma malha computacional de 500×500 pontos nodais, utilizando 32 processadores disponíveis no cluster *Enterprise* e considerando que:

- o algoritmo desse trabalho utiliza o método iterativo SOR;
- o algoritmo foi desenvolvido para processamento paralelo, utilizando 5 iterações por comunicação;
- o tamanho máximo do problema para usar eficientemente a memória *cache* é de 4096 pontos nodais, pois:

  → a *cache* L2 é de 256KB, mantendo um folga em torno de 10%, e de acordo com a Tabela 4.1, o problema com 4096 pontos nodais (64×64) utiliza 232KB (cerca de 90% do tamanho da *cache*).

  → o Valgrind mostra também (Tabela 4.2) que o problema com 4096 pontos nodais é o maior que ainda possui um taxa de *cache miss* baixa.

Se a matriz de elementos possui dimensões 500×500, então a cavidade é formada por 25000 pontos nodais. Como cada processo pode possuir até 4096 pontos nodais, o número total de processos deve ser pelo menos igual a 62. Ainda para satisfazer a Equação (32), é necessário que o número de processos seja múltiplo do número de processadores, e de acordo com o exemplo, são 32 processadores. Portanto, o menor número múltiplo de 32 que seja maior ou igual a 62, é 64. Em outras palavras, utilizar dois processos por processador neste exemplo, é aplicar de forma mais otimizada a técnica de utilização eficiente da memória *cache* apresentada neste trabalho.

Para este problema de exemplo, o número ideal de processos por processador é dois. É importante lembrar que quanto maior for esse número, pior poderá ser o benefício da *cache* em relação ao custo da comunicação e do acréscimo do número de iterações para atingir a convergência. Salvo quando a divisão do domínio garantir uma boa proporcionalidade das dimensões de seus subdomínios ou quando a divisão dos subdomínios entre os processadores for bem estruturada de forma a proporcionar um baixo tráfego na rede. Neste caso, mesmo com um número elevado de processos por processador, ainda é possível obter desempenho, desde que os diversos subdomínios em um mesmo processador façam o máximo de fronteiras possíveis entre si, de forma que haja o mínimo de comunicação com subdomínios alocados em outro processador.

(a) 64 processos distribuídos em apenas 2 processadores

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

(b) 64 processos distribuídos em 16 processadores

0	0	0	0	1	1	1	1
2	2	2	2	3	3	3	3
4	4	4	4	5	5	5	5
6	6	6	6	7	7	7	7
8	8	8	8	9	9	9	9
10	10	10	10	11	11	11	11
12	12	12	12	13	13	13	13
14	14	14	14	15	15	15	15

Figura 4.9: Representação esquemática das distribuições dos subdomínios entre os processadores, (a) utilizando 2 processadores, cada um com 32 processos e (b) utilizando 16 processadores com 4 processos cada.

A Figura 4.9 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 4.9, 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.

O Gráfico 4.15 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 (já apresentados nas seções anteriores), com a técnica de otimização de uso da *cache*. Foi usado o método de árvore binária para as comunicações entre os processos, realizando cinco iterações SOR entre as comunicações. 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 o roteiro desta seção. 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 4.4 apresenta a quantidade de processos por processador utilizados na forma otimizada.

Tabela 4.4: 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

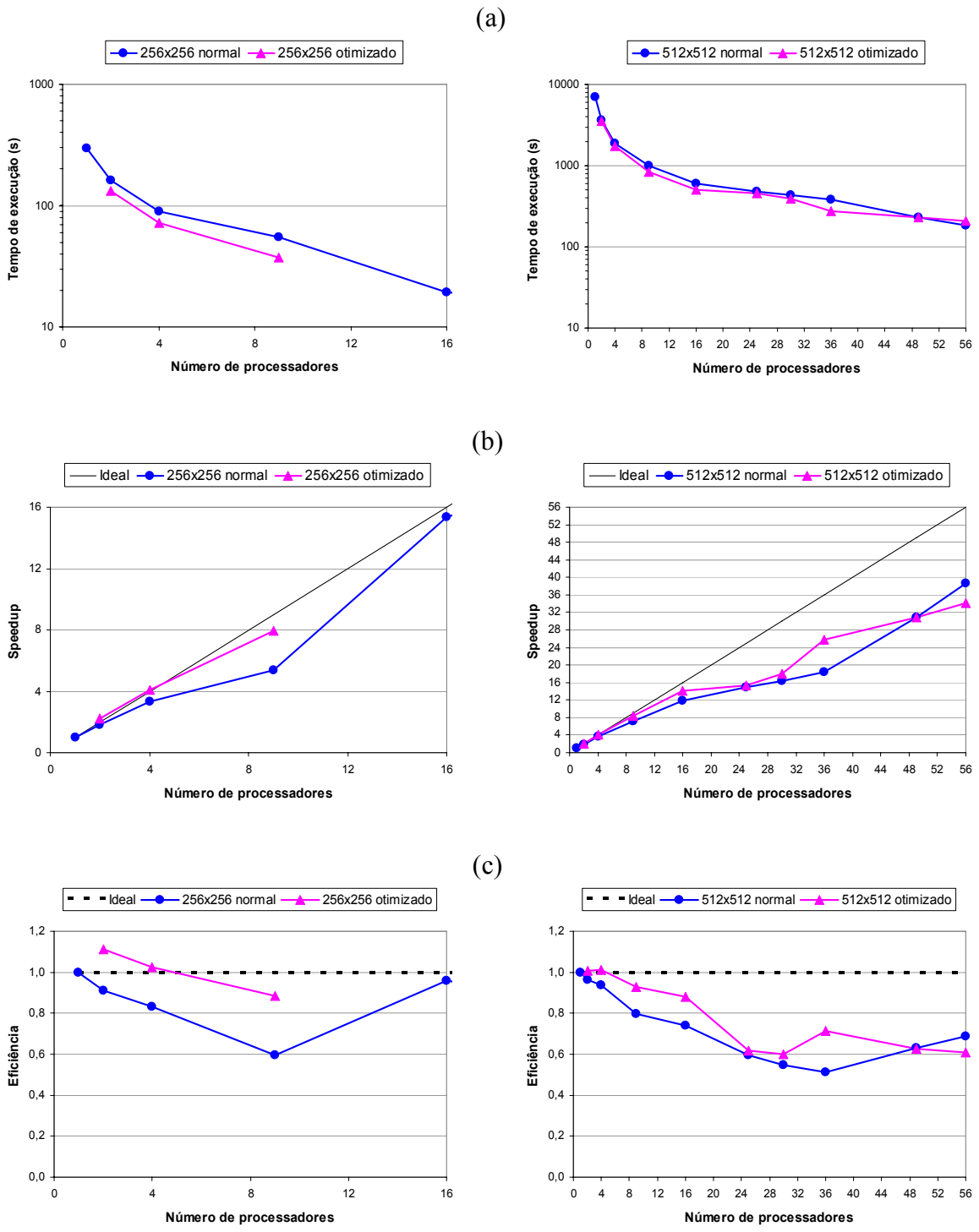


Gráfico 4.15: (a) Tempo de execução, (b) *Speedup* e (c) eficiência paralela obtida nos problemas com 256×256 e 512×512 pontos nodais, utilizando o método de árvore-binária de comunicação e freqüência de comunicação igual a cinco. Foram executados de forma convencional (um processo por processador) e otimizado (mais de um processo por processador).

O Gráfico 4.15a apresenta os tempos de execução das simulações, enquanto o Gráfico 4.15b apresenta o *speedup* e o Gráfico 4.15c a eficiência paralela obtidos. É importante notar que, o *speedup* (Gráfico 4.15b) e a eficiência paralela (Gráfico 4.15c), 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 até com até 9 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 no Gráfico 4.15c (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[2]. 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.

[2] A Tabela 4.4 mostra que na malha 512x512 utilizando 56 processadores foi necessário 112 processos, sendo que o mínimo exigido é de 64.

5 Conclusão e Recomendações para trabalhos futuros

Um algoritmo numérico para simulação de escoamentos bidimensionais transientes de fluidos viscosos incompressíveis foi implementado baseado no método das diferenças finitas e projetado para plataformas de processamento paralelo com memória distribuída, particularmente para *clusters de estações de trabalho*. Com o intuito de reduzir o tempo total de processamento e melhorar o desempenho do algoritmo paralelo, foi investigado neste trabalho o uso de três estratégias de comunicação, uma técnica para reduzir a freqüência a qual as comunicações entre os processadores devem ocorrer e uma estratégia de utilização eficiente da memória *cache*.

Para avaliar o desempenho do algoritmo paralelo e analisar as diferentes estratégias de paralelização, simulações com 1, 2, 4, 9, 16, 25, 30, 36, 49 e 56 processadores foram executadas. As três estratégias de comunicação avaliadas para distribuir os estágios de comunicação foram: comunicação *todos-para-todos*, arquitetura *mestre-escravo* e *árvore-binária*. Os resultados indicam que não existe diferença significativa entre as três implementações quando são utilizados menos de 36 processadores. No entanto, o uso da árvore-binária implementada via função "*All-reduce*" do MPI apresentou desempenho muito superior às outras duas estratégias quando o número de processadores aumenta.

Nas simulações realizadas também foi possível observar que a freqüência de comunicações entre os processadores altera significativamente o tempo de processamento. A escolha da freqüência de comunicação é um compromisso entre o acoplamento da solução dos subdomínios e o tráfego da rede devido ao volume de mensagens. Na realidade, deseja-se comunicar com maior freqüência possível para assegurar uma rápida convergência, mas sem causar uma sobrecarga na rede. Conclui-se que melhor resultado acontece quando cinco iterações SOR são realizadas antes da comunicação.

Uma técnica de utilização eficiente da memória *cache* para proporcionar redução no tempo total de processamento também 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 disponibilização 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.

A realização deste trabalho apresenta respostas técnicas para muitas perguntas práticas referentes à implementação de algoritmos de MFC em *clusters de estações de trabalho*. Entretanto, este trabalho de pesquisa mostra que existem muitos pontos onde estudos mais aprofundados podem ser realizados. Algumas sugestões de trabalhos futuros que podem dar continuidade as investigações feitas neste trabalho, estão descritos a seguir:

- **Mudança dinâmica do número de iterações**. Conforme descrito na Seção 4.3.3.4.3 (página 74)**,** para otimizar o desempenho de soluções paralelas é conveniente que ocorra mais de uma iteração SOR entre cada comunicação entre os processos. O uso desse número constante pode ser ruim, uma vez que os números totais de iterações em cada passo de tempo normalmente são diferentes. Definir um número alto de iterações por comunicação, pode fazer com que o final de uma simulação que atingiu um estado estacionário faça inúmeras iterações sem necessidade. Da mesma forma, utilizar um número baixo faz com o início da simulação (que exige mais para atingir a

convergência), faça comunicações em excesso. Por isso, manter o mesmo número de iterações para cada comunicação pode ser prejudicial.

- **Problemas utilizando malhas não estruturadas**. A aplicação da técnica de divisão por blocos para otimização do uso da *cache* apresentada neste trabalho em malhas não estruturadas pode ser de grande sucesso, pois os pontos nodais muitas vezes não estão organizados de forma a garantir a uma boa eficiência da memória *cache*. Com a técnica, os dados estariam em *cache* diminuindo a taxa de *cache miss*.

- **Problemas com outro método de solução.** Uma sugestão é aplicar a técnica de divisão por blocos para otimização do uso da *cache* deste trabalho em problemas que utilizam outro método para solução dos sistemas lineares, já que foi aplicada ao caso particular do método SOR. Outros métodos como o Gradientes Conjugados (TREFETHEN; BAU, 1997) ou GMRES (TREFETHEN; BAU, 1997) devem ser analisados.

- **Melhor divisão dos blocos de processamento**. Na Seção 4.3.3.4.2 (página 69), foi observado que uma divisão ruim dos blocos entre os processadores pode fazer com que a técnica de otimização da *cache* não apresente bons resultados. A desproporcionalidade entre a altura e a largura dos blocos pode influenciar os resultados, sendo que esse desequilíbrio causa um aumento no número de iterações necessárias para atingir a convergência global.

- **Dividir os problemas em um mesmo processador sem as comunicações do MPI, utilizando apenas um processo**. Neste trabalho a implementação da estratégia otimização da *cache* foi efetuada através da execução de mais de um processo MPI por processador. Se a divisão dos problemas em um mesmo processador ocorresse sem as comunicações do MPI, utilizando apenas um processo, não haveria custo de comunicação MPI em um mesmo processador. Esta forma de implementação pode acelerar de maneira significativa a computação. Todavia, o uso desta estratégia implica em grandes alterações no algoritmo para se beneficiar da capacidade da memória *cache*.

- ***Prefetch* de dados**. Na família dos processadores Intel, a partir do Pentium MMX, surgiu instruções de pré-busca (*prefetch*) de dados. Essas instruções iniciam uma leitura ou escrita de dados, mas não aguardam pela sua conclusão. O dado solicitado é trazido até um nível especificado da *cache*, mas sem alterar qualquer registrador. Por

outro lado, se o dado já estiver em *cache*, nada será feito. A finalidade do *prefetch* é de disponibilizar em *cache* os dados que serão futuramente utilizados. Em simulações numéricas, onde são percorridos vetores de grandes dimensões ordenadamente (malhas estruturadas, por exemplo), podem-se obter grandes benefícios de desempenho ao iniciar uma busca de informações que serão certamente necessárias posteriormente.

- ***Overlap* entre comunicação e processamento**. Avaliar o acréscimo de desempenho ao utilizar processamento paralelo, havendo mais de um processo em cada processador. Mas em vez de explorar o efeito da memória cache, explorar o aproveitamento do processador. Isto é, enquanto um processo está aguardando a comunicação, um outro processo está realizando computação. Isso contribui para a diminuição da ociosidade do processador.
- **Outras configurações de computadores**. Aplicar as técnicas apresentadas neste trabalho utilizando diferentes arquiteturas de computadores, principalmente variando o tamanho da memória cache. Outra sugestão seria utilizar duas configurações similares, entretanto, uma com o processador mais veloz e memória principal mais lenta, e a outra com o processador mais lento e memória principal mais rápida. É esperado que a primeira configuração apresente um melhor resultado com a aplicação da técnica de utilização eficiente da memória cache, uma vez que, sem a técnica o processador ficaria muito tempo ocioso aguardando por informações provenientes da memória principal.

6 Referências

ANDERSON Jr.; J. D. **Computational Fluid Dynamics**: The Basics with Applications. [S.I.]: McGraw Hill Inc., 1995.

DE ANGELI, J. P., VALLI, A. M. P., DE SOUZA, A. F., REIS Jr, N. C. **Numerical Simulations of the Navier-Stokes Equations Using Clusters of Workstations**. Computer Architecture and High Performance Computing, São Paulo, 2003.

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

GRIEBEL, M.; DORNSEIFER, T.; NEUNHOEFFER, T. **Numerical Simulation in Fluid Dynamics**: A Pratical Introduction. Philadelphia: SIAM, 1998.

GULLERUD, A. S., DODDS Jr., R. H. **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, fev.2001.

HARLOW, F.; WELCH, J. **Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface**: Phys. Fluids. [S.I.], 1965. v. 8, p. 2182-2189.

HIRT, C.; NICOLAS, B.; ROMERO, N. **SOLA – A Numerical Solution Algorithm for Transient Fluid Flows**: Technical report LA-5852. Los Alamos, NM: Los Alamos National Lab, 1975.

KOWARSCHIK, M. *et al*. **Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions**. [S.I.]: Computing 64, 2000. p. 381-399.

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

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

NETHERCOTE, N., SEWARD, J. **Valgrind: A Program Supervision Framework**. Electronic Notes in Theoretical Computer Science, v. 89, n. 2, p. 1-23, out.2003.

NEWSOME, J., SONG, D. **Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software**. Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, fev.2005.

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

SAHA, A. K., BISWAS, G., MURALIDHAR, K. **Three-dimensional study of flow past a square cylinder at low Reynolds numbers**. International Journal of Heat and Fluid Flow, 2003. p. 64-66.

SGE: Sun™ Grid Engine: Enterprise Edition. Versão 5.3p4. [S.I.]: Sun Microsystems Inc., 2004.

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

SIMOES, S. N. **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, 2004.

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

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

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

TREFETHEN, L. N., BAU, D. **Numerical Linear Algebra**. Soc for Industrial & Applied Math, mai.1997. ISBN: 0898714540.

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

VALGRIND: GPL'd system for debugging and profiling x86-Linux programs. Versão 2.2.0. [S.I.]: GNU, 2004. http://valgrind.kde.org. Plataforma Linux.

WEIDENDORFER, J., TRINITIS, C. **Cache Optimizations for Iterative Numerical Codes Aware of Hardware Prefetching**. Technische Universität München, Germany, 2004.