UNIVERSIDADE FEDERAL DO PARANÁ

EDUARDO DOBASHI FURUZATO

PARALELIZAÇÃO HÍBRIDA DE UMA FORMULAÇÃO CLÁSSICA DO MÉTODO DOS ELEMENTOS DE CONTORNO

CURITIBA

2014

EDUARDO DOBASHI FURUZATO

PARALELIZAÇÃO HÍBRIDA DE UMA FORMULAÇÃO CLÁSSICA DO MÉTODO DOS ELEMENTOS DE CONTORNO

Dissertação apresentada ao Programa de Pós-Graduação em Métodos Numéricos em Engenharia, Setor de Tecnologia, Universidade Federal do Paraná, como requisito parcial à obtenção do título de Doutor em Métodos Numéricos em Engenharia, Área de Concentração: Mecânica Computacional.

Orientador: Prof. Luiz Alkimin de Lacerda

Coorientador: Prof. Manoel Theodoro Fagundes Cunha

CURITIBA

2014

F992p Furuzato, Eduardo
Paralelização híbrida de uma formulação clássica do método dos elementos de contorno / Eduardo Furuzato. – Curitiba, 2014.
87f. : il. [algumas color.] ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Tecnologia, Programa de Pós-graduação em Métodos Numéricos em Engenharia , 2014.

Orientador: Luiz Alkimin de Lacerda -- Co-orientador: Manoel Theodoro Fagundes Cunha.
Bibliografia: p. 82-86.

1. Método dos elementos de contorno. 2. Balanceamento de carga adaptativo I. Universidade Federal do Paraná. II. Lacerda, Luiz Alkimin de. III. Cunha, Manoel Theodoro Fagundes. IV. Título.

CDD: 515.35

TERMO DE APROVAÇÃO

EDUARDO DOBASHI FURUZATO

PARALELIZAÇÃO HÍBRIDA DE UMA FORMULAÇÃO CLÁSSICA DO MÉTODO DOS ELEMENTOS DE CONTORNO COM BALANCEAMENTO DE CARGA ADAPTATIVO

Dissertação aprovada como requisito parcial para obtenção do grau de mestre no Programa de Pós-Graduação em Métodos Numéricos em Engenharia, da Universidade Federal do Paraná, pela seguinte banca examinadora:

Prof. Dr. Luiz Alkimin de Lacerda
Orientador - Membro do PPGMNE/UFPR.

Prof. Dr. Marcelo Rodrigues Bessa
Membro da UFPR

Prof. Dr. Roberto André Hexsel
Membro da UFPR

Participação por vídeo conferência
Prof. Dr. Luis Paulo da Silva Barra
Membro da UFJF

Curitiba, 10 de dezembro 2014

À minha esposa, Juliana,

e às nossas filhas, Julia e Isabel Karina.

AGRADECIMENTOS

Ao meu orientador, professor Alkimin, por sua incomparável dedicação. Sendo eu apenas mais um na longa fila dos seus privilegiados orientandos, nunca me faltou a orientação precisa na ocasião oportuna. Sua clareza de ideias invejável encontra rapidamente respostas para os problemas que vão aparecendo pelo caminho e antevê as dificuldades que estão além da visão do homem comum.

Ao meu co-orientador, professor Manoel, por sua contribuição na fase inicial deste trabalho.

Aos professores do PPGMNE, em especial aos professores Carrer e, novamente, Manoel, pela excelente introdução aos tópicos que fazem parte deste trabalho.

Ao professor Roberto, cujo entusiasmo por sua arte extrapola departamentos e instituições.

Ao professor Marcelo, pelo estímulo constante, pelo ânimo contagiante e pela grande generosidade ao me ceder um espaço no seu local de trabalho.

Ao professor Jeferson, à Mariana e sua equipe, pelo acesso ao cluster do Lactec.

À Amanda, ao Rodrigo, ao Eduardo, à Teilar e à Marcelle pelo companheirismo.

Ao Raphael, por reacender meu entusiasmo pelo Fortran depois de vinte anos.

RESUMO

Nos últimos vinte anos, a pesquisa na área do método dos elementos de contorno tem produzido implementações que permitem computação paralela eficiente e alocação de memória em vários ambientes computacionais. No entanto, existe um legado de várias décadas de programas científicos, muitos desses ainda usados na solução de problemas específicos de engenharia, que não se beneficiam dessas ideias. O presente trabalho descreve o processo de paralelização de um programa baseado na formulação clássica do Método dos Elementos de Contorno para problemas de potencial tridimensionais. O resultado desse processo é um programa com paralelismo híbrido e áreas de armazenamento de dados locais. Testes de larga escala são efetuados para validar cada etapa do processo e a eficiência das modificações introduzidas são avaliadas e comparadas com a abordagem original.

Palavras-chave: Método dos Elementos de Contorno, Paralelização Híbrida, Problemas de Potencial, Balanceamento de Carga Adaptativo

ABSTRACT

For the past twenty years, research on the boundary element method has produced implementations that allow the efficient parallel computation and memory allocation for a variety of hardware configurations. However, there remains several decades of legacy scientific programs, currently in use for the solution of specific engineering problems, that do not benefit from these ideas. The present work describes the process of parallelization of a legacy program, originally based on the classical boundary elements formulation for three-dimensional potential problems. The result of the process is a program with hybrid parallelism and data cache. Large scale tests are performed in order to validate each step of the process, the efficiency of the modifications is evaluated and compared with the original approach.

Keywords: Boundary Element Method, Hybrid Parallelization, Potential Problems, Adaptive Load Balance

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAS DE SÍMBOLOS E ABREVIAÇÕES

CARACTERES ROMANOS

MEC	Método dos Elementos de Contorno
n	Número de nós do contorno de um problema
u	Potencial em um determinado ponto
q	Fluxo em relação à normal da superfície de contorno
$\overline{u}$	Condição de contorno essencial
$\overline{q}$	Condição de contorno natural
u^*	Solução Fundamental do problema do potencial
q^*	Derivada em relação à normal da superfície de contorno da Solução Fundamental u^*
$\mathbf{n}$	número de nós do $cluster$ utilizados na execução de um programa MPI
$\mathbf{ppn}$	número de processos por nó utilizados na execução de um programa MPI
$\mathcal{O}$	Ordem de complexidade de um algoritmo

CARACTERES GREGOS

(η_1, η_2)	Coordenada local de um ponto em um elemento de contorno
φ	Função de forma do elemento de contorno
Ω	Domínio de um problema
Γ	Contorno de um problema
Δ	Delta de Dirac
ξ	Ponto fonte
∇	Operador gradiente
$\nabla\cdot$	Operador divergente
∇^2	Operador de Laplace

SUMÁRIO

1 INTRODUÇÃO

A formulação clássica do Método dos Elementos de Contorno (MEC) envolve a montagem de um sistema linear com complexidade da ordem de $\mathcal{O}(n^2)$, onde n é o número de graus de liberdade do problema. Como a matriz do sistema é densa, sua solução através de métodos diretos apresenta complexidade de $\mathcal{O}(n^3)$. O uso de métodos iterativos do tipo Krylov tem produzido bons resultados, reduzindo a complexidade da solução para $\mathcal{O}(i\,n^2) = \mathcal{O}(n^2)$, com $i << n$, onde i indica o número de iterações (BARRA *et al.*, 1992; GRAMA; KUMAR; SAMEH, 1996; VALENTE; PINA, 2006). A matriz densa também requer espaço de armazenamento da ordem de $\mathcal{O}(n^2)$.

Nos últimos vinte anos, a pesquisa na área do MEC tem produzido formulações que utilizam aproximações assintoticamente ótimas das matrizes densas (RJASANOW; STEINBACH, 2007), como a expansão em multipólos (MECMP) (GREENGARD; ROKHLIN, 1997), o uso de *wavelets* (MEC wavelets) (LAGE; SCHWAB, 1999) e a aproximação cruzada adaptativa (ACA) (BEBENDORF; RJASANOW, 2003), que reduzem a complexidade da montagem para $\mathcal{O}(n \log n)$, reduzem os requisitos de armazenamento para $\mathcal{O}(n \log n)$ e permitem computação paralela em uma variedade de ambientes computacionais (BHARADWAJ *et al.*, 1995; SINGER, 1995; GRAMA; KUMAR; SAMEH, 1996; HERNÁNDEZ, 1998; BEBENDORF; KRIEMANN, 2005; MAERTEN, 2010; WOLF; LELE, 2011; QUEVEDO *et al.*, 2012; ZHAO; TONG; JU, 2012). Alguns livros textos recentes que abordam o MEC (BEER; SMITH; DUENSER, 2008) e o MECMP (LIU, 2009) também dedicam capítulos à computação paralela.

A possibilidade de se dividir o esforço computacional em diversos processadores tem sido explorada desde a década de 60, com o surgimento dos primeiros supercomputadores. No entanto, a programação paralela explícita ganha maior destaque a partir de 2004, com o fim da era dos uniprocessadores em computadores pessoais, pois possibilita que um programa científico explore todo o potencial computacional dos multiprocessadores presentes nos computadores pessoais (HENESSY; PATTERSON, 2012). A ideia básica da programação paralela

é dividir o trabalho realizado por um programa sequencial em tarefas concorrentes, que são distribuídas e executadas em múltiplos processadores ou núcleos de processamento. No nível do sistema operacional, há duas possibilidades de se realizar essa distribuição: em **processos** simultâneos ou em **linhas de execução** de um mesmo processo (*threads*).

A divisão de tarefas entre processos concorrentes é baseada no modelo de **processamento paralelo em ambiente computacional com memória distribuída**, pois cada processo possui o seu próprio espaço de endereçamento de memória virtual, que é protegido dos outros processos pelo sistema operacional. Atualmente, o padrão aberto **MPI (*Message Passing Interface*)**, que viabiliza a computação paralela através de trocas de mensagens entre processos, é uma implementação bastante utilizada desse modelo (GRAMA *et al.*, 2003; RAUBER; RÜNGER, 2010). Os processos de um programa que utiliza MPI podem residir em um mesmo computador ou em computadores diferentes. Um ***cluster* de computadores** é um conjunto de computadores ligados em rede. Cada um desses computadores é chamado de um **nó do *cluster***. A execução do programa paralelo deve especificar o **número de nós (n)** e o **número de processos por nó (ppn)** que determinam a quantidade total de processos criados.

A especificação POSIX define as linhas de execução como processos leves (*light weight processes*): o sistema operacional atribui tempo de processador para cada linha de execução como se fosse um processo diferente, mas a criação de novas linhas de execução é menos onerosa do que a criação de um novo processo, pois não há replicação do espaço de memória. Como o espaço de memória de um processo é compartilhado por todas as suas linhas de execução, o processamento paralelo em diversas linhas de execução é um exemplo do modelo de **processamento paralelo em ambiente computacional com memória compartilhada**. O padrão aberto **OpenMP** tem sido uma das implementações mais utilizadas desse modelo, pois retira do programador a responsabilidade de criar, sincronizar e destruir explicitamente as linhas de execução e ao mesmo tempo fornece maior portabilidade ao código (GRAMA *et al.*, 2003; CHANDRA *et al.*, 2001).

A combinação dos dois modelos de processamento paralelo, com memória compartilhada entre linhas de execução e distribuída entre processos, é chamada de **paralelização**

híbrida.

Apesar da popularização da computação paralela nas formulações mais recentes do MEC, dos benefícios da programação paralela na utilização dos recursos computacionais atuais e do amadurecimento dos padrões abertos disponíveis ao programador, existe um legado de décadas de programas científicos que foram concebidos para o processamento sequencial e não se beneficiam dessas ideias, sendo limitados tanto pelo poder de processamento de um único núcleo quanto pela memória oferecida pela máquina em que são executados. Neste trabalho é descrito o processo de paralelização de um programa científico legado baseado na formulação clássica do BEM para problemas do potencial tridimensionais. A primeira versão do programa foi escrita em 2001 e foi modificada por vários programadores ao longo de sua história. As últimas versões foram usadas no cálculo da resistividade do solo para a proteção catódica de torres elétricas e usam algoritmo genético para resolver o problema inverso correspondente (LACERDA; SILVA, 2006; CARRAZEDO; LACERDA, 2009). Para uma análise mais concisa, este trabalho se concentra no núcleo do programa legado, que corresponde à montagem e à solução do sistema de equações lineares. Porém, esse isolamento procura preservar ao máximo a estrutura do código original, com o intuito de facilitar a eventual inserção das antigas funcionalidades.

1.1 FORMULAÇÃO CLÁSSICA PARALELA DO MEC

As primeiras implementações paralelas do MEC foram realizadas nos anos 80 por Symm (1984) e Davies (1988) para o *Distributed Array Processor* (DAP), um dispositivo capaz de armazenar matrizes quadradas e realizar instruções vetoriais, instruções de máquina que operam simultaneamente todos os elementos de uma coluna da matriz. Ambos trabalharam com o problema do potencial. Symm paralelizou a solução direta do sistema linear e Davies adicionou processamento paralelo às integrações numéricas e ao rearranjo das matrizes.

Kreienmeyer e Stein (1997) caracterizam duas formas de distribuição de dados e tarefas entre os processadores de um ambiente computacional de memória distribuída: decom-

posição de dados e decomposição de domínio. Na decomposição de dados, os nós funcionais dos elementos de contorno são igualmente distribuídos entre os processadores. A vantagem desse método é um balanceamento de carga razoável obtido de forma simples, mas em contrapartida há maior comunicação entre os processadores durante a solução do sistema. A decomposição de domínio foi introduzida por Kamiya, Iwase e Kita (1996) e consiste em subdividir o domínio em um número de subdomínios correspondente ao número de processadores através de contornos virtuais. Assume-se valores para os contornos virtuais e cada processador resolve um pequeno problema independente. A união dos resultados é comparada com o resultado esperado, ajustes são realizados nos valores de contorno virtuais e nova iteração da computação paralela é realizada. Uma das dificuldades do segundo método é a geração manual dos contornos virtuais, especialmente quando se trabalha em ambientes com centenas de processadores.

Cunha, Telles e Coutinho (2004) fornecem duas alternativas de paralelização de uma programa legado que resolve o problema elastoestático tridimensional. A primeira alternativa implementa a paralelização em ambiente computacional com memória compartilhada: a solução do sistema utiliza a biblioteca numérica LAPACK e a montagem é paralelizada com OpenMP. Na segunda alternativa, em ambiente com memória distribuída, a solução do sistema utiliza a biblioteca numérica ScaLAPACK e decomposição de domínio. O trabalho de Ataseven (2005) segue a linha proposta pela segunda alternativa de Cunha para geração de imagens do cérebro humano através de fontes eletromagnéticas.

Beer, Smith e Duenser (2008) propoem a solução paralela do MEC elastoestático pela combinação de uma solução iterativa do sistema (BiCGStab) com a montagem da matriz elemento-por-elemento e paralelizado com MPI.

Araujo, Azevedo e Gray (2010) apresentam uma versão paralelizada, em ambiente com memória distribuída, do seu MEC Subregião-por-Subregião (SPS) para a análise microestrutural de nanotubos de carbono. O MEC SPS é um método de decomposição de domínio sem sobreposições que, ao contrário do método de Kamiya, não é iterativo, pois as condições de acoplamento nas interfaces de contorno são aplicadas diretamente. Como no caso elemento-por-elemento, a matriz global não é diretamente montada, reduzindo os requisitos

de armazenamento para cada subregião. Foi utilizado o método iterativo BiCG com pré-condicionamento para a solução sistema.

Labaki, Ferreira e Mesquita (2011) apresentam implementações paralelas das formulações clássicas do MEC para problemas do potencial e elastoestáticos bidimensionais em *Graphical Processing Units* (GPUs) com 240 unidades de processamento e obtendo melhora de performance de 56 vezes em relação à CPU da mesma máquina. Foi utilizada a linguagem CUDA (*Compute Unified Device Architecture*), desenvolvida pelo próprio fabricante da GPU, e o paradigma de programação paralela *Single Instruction Multiple Data* (SIMD), em que a mesma instrução é efetuada em centenas de dados diferentes.

Neste trabalho, aplicamos e estendemos as ideias de Beer, Smith e Duenser (2008) a uma implementação do problema do potencial, transformando a paralelização em ambiente com memória distribuída em paralelização híbrida e utilizando armazenamento em memória disponível.

Apesar de Beer, Smith e Duenser (2008) apontarem a redução do requisito de memória para $\mathcal{O}(n)$ como motivação da montagem elemento-por-elemento, a implementação fornecida pelos autores armazena em memória as submatrizes dos elementos e requer pelo menos duas vezes mais memória que a montagem clássica [p.206]. Portanto, o requisito de memória continua sendo $\mathcal{O}(2n^2) = \mathcal{O}(n^2)$. Talvez a intenção dos autores tenha sido diminuir dessa forma a penalidade imposta ao processamento, porém, mostra-se adiante que isso pode não acontecer para malhas com milhares de elementos.

A ideia de guardar dados frequentemente utilizados em memória é uma ideia simples, porém, quando se trabalha com um grande volume de dados, um certo conhecimento do ambiente computacional é necessário para que seja implementada de forma a produzir resultados positivos.

Este trabalho mostra como a utilização da memória física disponível pode minimizar o esforço computacional do programa, com ganhos de até 10.47 nos tempos de execução. A paralelização híbrida multiplica esse ganho por até 2.15.

Outro assunto abordado aqui é uma técnica simples de balanceamento de carga que considera a carga imposta aos nós de processamento por processos alheios ao programa para-

lelizado. Usualmente, o processamento paralelo com MPI ocupa igualmente todos os núcleos de processamento disponíveis, sem levar em conta que parte do processamento é sempre utilizado por processos do sistema operacional e outros eventuais processos concorrentes. Este trabalho contém ensaios em que a carga adicional dos processadores, decorrente da utilização compartilhada de recursos, é prevista pelo algoritmo de balanceamento de carga.

A forma como o armazenamento de dados em memória disponível e o balanceamento de carga são aplicados ao MEC neste trabalho não foi encontrada na literatura, talvez justamente por sua especificidade e singeleza, como será descrito adiante.

1.2 OBJETIVOS

Este trabalho tem como objetivo efetuar alterações no programa legado sequencial que permitam utilizar todos os recursos de processamento, memória e comunicação disponibilizados por um cluster de computadores, mantendo o conceito clássico da formulação do MEC, porém maximizando o tamanho dos problemas resolvidos.

1.3 ORGANIZAÇÃO DO TEXTO

O texto está organizado em cinco capítulos. O capítulo 2 aborda a formulação clássica do problema do potencial e descreve as principais subrotinas do programa original. O capítulo 3 descreve a metodologia do trabalho, incluindo ambiente computacional, ferramentas, medidas e experimentos utilizados e as alterações efetuadas no código. O capítulo 4 apresenta os resultados experimentais obtidos. Finalmente, o capítulo 5 contém as conclusões do trabalho.

2 FORMULAÇÃO CLÁSSICA DO MÉTODO DOS ELEMENTOS DE CONTORNO PARA PROBLEMAS DO POTENCIAL

A formulação clássica do Método dos Elementos de Contorno (MEC) para o problema do potencial descrita aqui é bastante difundida na literatura e pode ser encontrada, praticamente sem variações, em Brebbia, Telles e Wrobel (1984), Becker (1992), Beer, Smith e Duenser (2008), Liu (2009).

No entanto, optou-se por apresentá-la com o intuito de auxiliar a descrição das rotinas principais do núcleo do programa original.

Além das equações mais importantes do MEC, foram incluídas também, especialmente na seção dedicada à discretização do contorno, algumas fórmulas triviais que serão utilizadas na descrição das subrotinas. Uma visão geral do estado inicial do programa facilita a descrição das alterações introduzidas no código nos capítulos posteriores.

Esse capítulo é dividido em três seções. A primeira seção apresenta o problema do potencial e a derivação das equações integrais utilizadas no MEC. A segunda seção aborda as discretizações do contorno e das equações integrais apresentadas na seção anterior. A última seção descreve a implementação das equações discretizadas em subrotinas.

2.1 DERIVAÇÃO DAS EQUAÇÕES INTEGRAIS

Partindo do problema do potencial, derivam-se duas equações integrais: a **equação integral para pontos interiores** e a **equação integral para pontos de contorno**. A notação adotada nesta seção foi retirada de Brebbia, Telles e Wrobel (1984). A derivação da primeira equação pelas identidades de Green foi adaptada de Binegar (1996) ao contexto do MEC. A derivação da segunda equação segue o desenvolvimento de Becker (1992).

2.1.1 Problema do potencial

Seja Ω uma região fechada e conexa do espaço $\mathbb{R}^3$, delimitada por uma superfície suave $\Gamma = \Gamma_1 \cup \Gamma_2$. O problema com valores de contorno conhecido na literatura como **problema do potencial** é definido:

- pela equação de Laplace:

$$\nabla^2 u(\mathbf{x}) = 0 \qquad \text{para} \quad \mathbf{x} \in \mathbf{\Omega} \tag{1}$$

- pela condição de contorno essencial ou de Dirichlet:

$$u(\mathbf{x}) = \overline{u}(\mathbf{x}) \qquad \text{para} \quad \mathbf{x} \in \mathbf{\Gamma_1} \tag{2}$$

- e pela condição de contorno natural ou de Neumann:

$$q = \overline{q}(\mathbf{x}) \qquad \text{para} \quad \mathbf{x} \in \mathbf{\Gamma_2} \tag{3}$$

sendo que $\mathbf{n}$ denota a normal à superfície Γ, $q = \frac{\mathrm{d}u(\mathbf{x})}{\mathrm{d}\mathbf{n}}$ e os valores de $\overline{u}(\mathbf{x})$ e $\overline{q}(\mathbf{x})$ são conhecidos em Γ_1 e Γ_2, respectivamente.

As Eqs.2 e 3 implicam que, para cada $\mathbf{x}$ no contorno Γ, um dos valores $\overline{u}(\mathbf{x})$ ou $\overline{q}(\mathbf{x})$ seja prescrito, o que é suficiente para garantir a unicidade de solução para o problema. Por simplicidade, considere-se Γ_1 e Γ_2 disjuntos, isto é, apenas um dos valores é prescrito para cada ponto do contorno.

A solução da equação:

$$\nabla^2 u^* = -\Delta(\xi, \mathbf{x}) \tag{4}$$

onde $\Delta(\xi, x)$ representa o delta de Dirac, é denominada **solução fundamental** do problema do potencial e é dada por:

$$u^*(\xi, \mathbf{x}) = \frac{1}{4\pi r} \tag{5}$$

onde r é a distância euclidiana entre $\mathbf{x}$ e ξ.

A derivada de u^* com relação à normal do contorno é dada por:

$$q^*(\xi, \mathbf{x}) = \frac{-1}{4\pi r^2} \frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}} \tag{6}$$

2.1.2 Equação integral para pontos interiores ao domínio

Brebbia, Telles e Wrobel (1984) apontam três caminhos para derivar a **equação integral para pontos interiores**: o método dos resíduos ponderados, as identidades de Green e analogia com as leis físicas do potencial elétrico. Por brevidade, serão utilizadas as identidades de Green, adaptando o desenvolvimento de Binegar (1996) ao contexto do MEC.

Dada uma função vetorial $\mathbf{v}$ de classe $\mathbb{C}^1$ definida em Ω, segue, pelo teorema de Gauss (ou teorema da divergência), que:

$$\int_\Omega \nabla \cdot \mathbf{v} \, \mathrm{d}\Omega = \int_\Gamma \mathbf{v} \cdot \mathbf{n} \, \mathrm{d}\Gamma \tag{7}$$

onde $\mathbf{n}$ representa a normal à superfície Γ.

Sejam ϕ e ψ funções escalares de classe $\mathbb{C}^2$ em Ω. Fazendo $\mathbf{v} = \phi \nabla \psi$, obtém-se:

$$\int_\Omega \nabla \cdot (\phi \nabla \psi) \, \mathrm{d}\Omega = \int_\Gamma \phi \nabla \psi \cdot \mathbf{n} \, \mathrm{d}\Gamma \tag{8}$$

Notando-se que:

$$\nabla \cdot (\phi \nabla \psi) = \nabla \phi \cdot \nabla \psi + \phi \nabla \cdot \nabla \psi$$

chega-se à **primeira identidade de Green**:

$$\int_\Omega \nabla \phi \cdot \nabla \psi \, \mathrm{d}\Omega + \int_\Omega \phi \nabla^2 \psi \, \mathrm{d}\Omega = \int_\Gamma \phi \nabla \psi \cdot \mathbf{n} \, \mathrm{d}\Gamma \tag{9}$$

A **segunda identidade de Green** é obtida aplicando-se o mesmo processo a $\psi \nabla \phi$

e subtraindo as equações simétricas:

$$\int_{\Omega}(\phi\nabla^2\psi - \psi\nabla^2\phi)\,\mathrm{d}\Omega = \int_{\Gamma}(\phi\nabla\psi - \psi\nabla\phi)\cdot\mathbf{n}\,\mathrm{d}\Gamma \tag{10}$$

Partindo da Eq.10, a **terceira identidade de Green** é obtida em dois passos. Primeiro, toma-se $\phi = u$, uma solução da equação de Laplace, e, consequentemente, $\nabla u\cdot\mathbf{n} = q$, resultando :

$$\int_{\Omega} u\nabla^2\psi\,\mathrm{d}\Omega = \int_{\Gamma} u\nabla\psi\cdot\mathbf{n}\,\mathrm{d}\Gamma - \int_{\Gamma}\psi q\,\mathrm{d}\Gamma \tag{11}$$

No segundo passo, como a **solução fundamental** u^* definida pela Eq.5 não é $\mathbb{C}^2$, toma-se $\psi = \frac{1}{r+\epsilon}$, com $\epsilon > 0$, e obtém-se:

$$\int_{\Omega} u(\nabla^2\frac{1}{r+\epsilon})\,\mathrm{d}\Omega = \int_{\Gamma} u\nabla\frac{1}{r+\epsilon}\cdot\mathbf{n}\,\mathrm{d}\Gamma - \int_{\Gamma}\frac{1}{r+\epsilon}q\,\mathrm{d}\Gamma \tag{12}$$

Tendo em mente a propriedade do delta de Dirac

$$\int_{\Omega} u(\mathbf{x})\Delta(\xi,\mathbf{x})\,\mathrm{d}\Omega = u(\xi)$$

e tomando o limite $\epsilon \to 0$ em ambos os lados de 12, como:

$$lim_{\epsilon\to 0}\nabla^2\frac{1}{r+\epsilon} = \nabla^2\frac{1}{r} = -\Delta(\xi,\mathbf{x})$$

chega-se finalmente à versão da **terceira identidade de Green** conhecida na literatura do Método dos Elementos de Contorno como **equação integral para pontos interiores ao domínio:**

$$u(\xi) = \int_{\Gamma} qu^*\,\mathrm{d}\Gamma - \int_{\Gamma} uq^*\,\mathrm{d}\Gamma, \qquad \text{para} \quad \xi \quad \text{em} \quad \Omega \tag{13}$$

Desse modo, se u satisfaz a equação de Laplace, seu valor em qualquer ponto do domínio Ω é completamente determinado pelos valores de u e q no contorno Γ.

2.1.3 Equação integral de contorno

Para tornar a Eq. 13 válida para $\xi \in \Gamma$ é necessário efetuar um processo limite, levando o ponto ξ ao contorno. Esses cálculos são efetuados, por exemplo, em Becker (1992, pp.95-97), retirando-se do contorno Γ uma área semi-esférica de raio ϵ e tomando o limite quando $\epsilon \to 0$, resultando em:

$$\frac{1}{2}u(\xi) = \int_\Gamma qu^* \, \mathrm{d}\Gamma - \int_\Gamma uq^* \, \mathrm{d}\Gamma, \qquad \text{para} \quad \xi \quad \text{em} \quad \Gamma \tag{14}$$

Assim, em um problema de valor de contorno bem formulado, os valores desconhecidos de u e q no contorno podem ser determinados pelos valores inicialmente prescritos $\overline{u}$ e $\overline{q}$. De posse dos valores das funções no contorno, a Eq.13 permite o cálculo da solução em qualquer ponto no interior do domínio.

2.2 SOLUÇÃO NUMÉRICA DAS EQUAÇÕES INTEGRAIS

Como se pode inferir das Eqs. 13 e 14, sua solução numérica requer apenas a discretização do contorno Γ do problema. Essa é uma das vantagens do MEC em relação ao Método dos Elementos Finitos (MEF), que requer a discretização do domínio do problema (COSTABEL, 1987; LIU, 2009).

Esta seção contém três subseções. A primeira subseção descreve a discretização do contorno adotada neste trabalho. A equação integral do contorno é discretizada na segunda seção e a última seção discute os métodos de integração numérica utilizados no programa original.

Como na seção anterior, as referências mais utilizadas nesta seção são Brebbia, Telles e Wrobel (1984) e Becker (1992).

2.2.1 Discretização do contorno do problema por elementos quadrilaterais planos com valores constantes de u e q

No MEC, a discretização do contorno de problemas tridimensionais é efetuada com técnicas análogas às utilizadas para problemas bidimensionais no Método dos Elementos Finitos (BREBBIA; TELLES; WROBEL, 1984). A discretização do contorno Γ pode ser realizada dividindo-se o contorno em um número finito de elementos quadrilaterais ou triangulares. A geometria desses elementos pode ser plana ou interpolada, por exemplo, de forma linear ou quadrática. Analogamente, os valores de u e q em cada elemento podem ser considerados constantes ou sua variação pode ser representada através de interpolações.

A eficácia do método de discretização adotado, quer em precisão dos resultados obtidos, quer em eficiência computacional, só pode ser avaliada no contexto de um problema e suas peculiaridades, tais como a geometria e as características físicas dos materiais envolvidos.

Para os exemplos rodados neste trabalho, a discretização do contorno em elementos quadrilaterais planos, considerando-se u e q constantes em cada elemento, é suficientemente adequada. Nesse caso, cada elemento é definido por seus quatro vértices (denominados **nós geométricos**) e considera-se que os valores de u e q são prescritos (ou calculados) no ponto localizado no centro do elemento (denominado **nó funcional**).

A ordenação dos nós geométricos determina o sentido da normal à superfície (regra da mão direita). Os quatro nós geométricos de um elemento são representados por:

$$\mathbf{x}^{[\mathbf{e}]} = (x_1^{[e]}, x_2^{[e]}, x_3^{[e]}) \qquad e = 1, 2, 3, 4 \tag{15}$$

Em cada nó, é definido um sistema de coordenadas locais (η_1, η_2) com origem no centro do elemento com $\eta_1, \eta_2 \in [-1, +1]$. Assim, o único nó funcional de cada elemento possui coordenadas locais $(\eta_1, \eta_2) = (0, 0)$.

As equações abaixo são as **funções de forma** dos elementos quadrilaterais planos:

$$\begin{aligned} \varphi_1 &= \frac{1}{4}(1-\eta_1)(1-\eta_2), \qquad & \varphi_2 &= \frac{1}{4}(1+\eta_1)(1-\eta_2) \\ \varphi_3 &= \frac{1}{4}(1+\eta_1)(1+\eta_2), \qquad & \varphi_4 &= \frac{1}{4}(1-\eta_1)(1+\eta_2) \end{aligned} \tag{16}$$

Para pontos no interior de um elemento, as equações abaixo transformam coordenadas locais (η_1, η_2) em coordenadas globais $\mathbf{x} = (x_1, x_2, x_3)$:

$$x_i = \sum_{e=1}^{4} \varphi_e x_i^{[e]} \qquad i = 1, 2, 3 \tag{17}$$

O diferencial $\mathrm{d}\Gamma$ é dado por:

$$\mathrm{d}\Gamma = \left| \frac{\partial \mathbf{x}}{\partial \eta_1} \times \frac{\partial \mathbf{x}}{\partial \eta_2} \right| \mathrm{d}\eta_1 \mathrm{d}\eta_2 = |J| \, \mathrm{d}\eta_1 \mathrm{d}\eta_2 \tag{18}$$

onde

$$\frac{\partial \mathbf{x}}{\partial \eta_1} = \left(\frac{\partial x_1}{\partial \eta_1}, \frac{\partial x_2}{\partial \eta_1}, \frac{\partial x_3}{\partial \eta_1} \right), \qquad \frac{\partial \mathbf{x}}{\partial \eta_2} = \left(\frac{\partial x_1}{\partial \eta_2}, \frac{\partial x_2}{\partial \eta_2}, \frac{\partial x_3}{\partial \eta_2} \right) \tag{19}$$

O jacobiano $|J|$ é também o módulo do vetor normal ao elemento:

$$\mathbf{n} = \frac{\partial \mathbf{x}}{\partial \eta_1} \times \frac{\partial \mathbf{x}}{\partial \eta_2} = (n_1, n_2, n_3) \tag{20}$$

onde

$$\begin{aligned} n_1 &= \frac{\partial x_2}{\partial \eta_1}\frac{\partial x_3}{\partial \eta_2} - \frac{\partial x_2}{\partial \eta_2}\frac{\partial x_3}{\partial \eta_1} \\ n_2 &= \frac{\partial x_3}{\partial \eta_1}\frac{\partial x_1}{\partial \eta_2} - \frac{\partial x_1}{\partial \eta_1}\frac{\partial x_3}{\partial \eta_2} \\ n_3 &= \frac{\partial x_1}{\partial \eta_1}\frac{\partial x_2}{\partial \eta_2} - \frac{\partial x_2}{\partial \eta_1}\frac{\partial x_1}{\partial \eta_2} \end{aligned} \tag{21}$$

$$|J| = (n_1^2 + n_2^2 + n_3^2)^{1/2} \tag{22}$$

2.2.2 Discretização da equação integral do contorno

Com o contorno discretizado conforme a seção anterior, a equação integral 14 pode ser aproximada como:

$$\frac{1}{2}u_i + \sum_{j=1}^{N} \int_{\Gamma_j} uq^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \simeq \sum_{j=1}^{N} \int_{\Gamma_j} qu^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \tag{23}$$

onde N é o número de elementos de contorno, o índice i corresponde ao ponto fonte ξ e o índice j corresponde aos elementos de contorno.

Como as funções u e q são constantes em cada elemento, podem ser retiradas das integrais:

$$\frac{1}{2}u_i + \sum_{j=1}^{N} u_j \int_{\Gamma_j} q^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \simeq \sum_{j=1}^{N} q_j \int_{\Gamma_j} u^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \tag{24}$$

Definindo as matrizes $\hat{\mathbf{H}} = \hat{h}_{ij}$ e $\mathbf{G} = g_{ij}$ por:

$$\hat{h}_{ij} = \int_{\Gamma_j} q^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \qquad g_{ij} = \int_{\Gamma_j} u^*(\xi_\mathbf{i}, \mathbf{x})\, \mathrm{d}\Gamma \tag{25}$$

e definindo a matriz $\mathbf{H} = h_{ij}$ por:

$$h_{ij} = \begin{cases} \hat{h}_{ij} & \text{quando } i \neq j, \\ \frac{1}{2} + \hat{h}_{ij} & \text{quando } i = j \end{cases}$$

a Eq.24 se transforma em:

$$\sum_{j=1}^{N} h_{ij} u_j = \sum_{j=1}^{N} g_{ij} q_j \tag{26}$$

Finalmente, transforma-se a Eq.26 em um sistema linear $\mathbf{Ab} = \mathbf{y}$, através de uma transposição de colunas das matrizes $\mathbf{H}$ e $\mathbf{G}$. Mais especificamente, a matriz $\mathbf{A}$ é formada pelas colunas de $\mathbf{H}$ e de $\mathbf{G}$ correspondentes às incógnitas de u e q e o vetor $\mathbf{y}$ é obtido pelo produto da matriz formada de modo análogo para os valores prescritos com o vetor dos valores

prescritos.

O cálculo dos elementos da matriz $\mathbf{A}$ é o assunto da próxima subseção.

2.2.3 Integração numérica

O cálculo dos elementos $\hat{h}_{ij}$ e g_{ij} normalmente é efetuado da seguinte maneira (BREBBIA; TELLES; WROBEL, 1984; BEER; SMITH; DUENSER, 2008; LIU, 2009; COSTA, 2013):

- Se $i \neq j$, então $\xi \neq \mathbf{x}$ e $r > 0$. Logo, q^* e u^* não apresentam singularidade e as integrações numéricas podem ser efetuadas através de uma quadratura Gaussiana.
- Se $i = j$, então q^* e u^* apresentam singularidade para $\mathbf{x} = \xi$. No caso dos elementos planos em problemas tridimensionais, $\hat{h}_{ii}$ são nulos, como será demonstrado abaixo. A singularidade da solução fundamental u^* pode ser contornada no cálculo dos elementos $\hat{g}_{ii}$ utilizando, por exemplo, as ideias de Guiggiani e Gigante (1990).

2.2.3.1 Quadratura gaussiana

Para os elementos que não pertencem à diagonal da matriz do sistema linear, a quadratura Gaussiana implementada no código original pode ser descrita pelas equações (BREBBIA; TELLES; WROBEL, 1984, pp.447):

$$\begin{aligned} \hat{h}_{ij} &= \int_{\Gamma_j} q^*(\xi_{\mathbf{i}}, \mathbf{x})\, \mathrm{d}\Gamma \\ &= \int_{-1}^{1}\int_{-1}^{1} q^*(\xi_{\mathbf{i}}, \eta)|J|\, \mathrm{d}\eta_1 \mathrm{d}\eta_2 \\ &\simeq \sum_{a=1}^{P}\left(\sum_{b=1}^{P} q^*(\xi_{\mathbf{i}}, \eta_{\mathbf{ab}})|J|\, w_b\right) w_a \end{aligned} \tag{27}$$

onde P representa o número de pontos de Gauss utilizado na integração, $\eta_{\mathbf{ab}} = (\eta_a, \eta_b)$ é um ponto de Gauss, w_a e w_b são os pesos do ponto de Gauss, $|J|$ é o jacobiano da equação 22.

Analogamente,

$$\begin{aligned} g_{ij} &= \int_{\Gamma_j} u^*(\xi_{\mathbf{i}}, \mathbf{x})\, \mathrm{d}\Gamma \\ &\simeq \sum_{a=1}^{P} \left(\sum_{b=1}^{P} u^*(\xi_{\mathbf{i}}, \eta_{\mathbf{ab}}) |J|\, w_b \right) w_a \end{aligned} \tag{28}$$

2.2.3.2 Diagonal da matriz do sistema linear

A função $q^*(\xi, \mathbf{x})$ apresenta uma singularidade forte em $\mathbf{x} = \xi$. No entanto, para discretizações com elementos planos, a integração de $q^*(\xi, \mathbf{x})$ no elemento pode ser calculada através de um processo limite análogo ao da seção 2.1.3.

Primeiro, note-se que, como os elementos de contorno são planos, o vetor normal $\mathbf{n}$ é constante em cada elemento. Portanto, dado um elemento Γ_i:

$$\frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}} = 0 \qquad \text{para todo } x \in \Gamma_i \tag{29}$$

Logo, para todo $\mathbf{x} \in \mathbf{\Gamma_i}$, exceto no ponto $\mathbf{x} = \xi_{\mathbf{i}}$:

$$q^*(\xi_{\mathbf{i}}, \mathbf{x}) = \frac{-1}{4\pi r^2} \frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}} = 0 \tag{30}$$

Seja Γ_ϵ uma região circular contida no elemento Γ_i com centro em ξ_i e raio ϵ. Seja $\Gamma_i \setminus \Gamma_\epsilon$ a região do elemento excluída da região circular Γ_ϵ. O elemento $\hat{h}_{ii}$ da matriz do sistema linear pode ser calculado como a seguir:

$$\begin{aligned} \hat{h}_{ii} &= \lim_{\epsilon \to 0} \left(\int_{\Gamma_i \setminus \Gamma_\epsilon} q^*(\xi_{\mathbf{i}}, \mathbf{x})\, \mathrm{d}\Gamma + \int_{\Gamma_\epsilon} q^*(\xi_{\mathbf{i}}, \mathbf{x})\, \mathrm{d}\Gamma \right) \\ &= \lim_{\epsilon \to 0} \left(0 + \int_{\Gamma_\epsilon} q^*(\xi_{\mathbf{i}}, \mathbf{x})\, \mathrm{d}\Gamma \right) \\ &= \lim_{\epsilon \to 0} \left(\int_{\Gamma_\epsilon} \frac{-1}{4\pi r^2} \frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}}\, \mathrm{d}\Gamma \right) \end{aligned} \tag{31}$$

Em coordenadas polares:

$$\begin{aligned}\hat{h}_{ii} &= \lim_{\epsilon\to 0}\left(\int_0^{2\pi}\int_0^{\epsilon}\frac{-1}{4\pi r^2}\frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}}r\,\mathrm{d}r\,\mathrm{d}\theta\right)\\ &= \frac{-1}{4\pi}\int_0^{2\pi}\lim_{\epsilon\to 0}\left(\int_0^{\epsilon}\frac{1}{r}\frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}}\,\mathrm{d}r\right)\mathrm{d}\theta\end{aligned} \tag{32}$$

Agora, note-se que o integrando mais interno da equação anterior coincide com a curvatura da superfície do elemento de contorno e pela segunda identidade de Frenet:

$$\lim_{\epsilon\to 0}\left(\frac{\frac{\mathrm{d}r}{\mathrm{d}\mathbf{n}}}{r}\right) = \frac{1}{\varrho} = \kappa = 0 \tag{33}$$

Portanto, segue de 32 e 33, que $\hat{h}_{ii} = 0$ para todo x em Γ_i.

Para elementos quadrilaterais planos, a singularidade de u^* ocorre no nó funcional $(\eta_1, \eta_2) = (0, 0)$.

Dividindo-se o elemento Γ_i em quatro sub-elementos triangulares delimitados pelas linhas que ligam o nó funcional aos nós geométricos denotados por Γ_{is}, com $s = 1, 2, 3, 4$, obtém-se:

$$\begin{aligned}g_{ii} &= \int_{\Gamma_i} u^*(\xi_{\mathbf{i}}, \mathbf{x})\,\mathrm{d}\Gamma\\ &= \sum_{s=1}^{4}\int_{\Gamma_{is}} u^*(\xi_{\mathbf{i}}, \mathbf{x})\,\mathrm{d}\Gamma\\ &= \sum_{s=1}^{4}\int_0^{\alpha_s}\int_0^{\rho(\theta)} u^*(\xi_{\mathbf{i}}, \mathbf{x})\left|J_{\rho_\theta}\right|\rho\,\mathrm{d}\rho\,\mathrm{d}\theta\\ &\simeq \sum_{s=1}^{4}\left(\sum_{a=1}^{P}\left(\sum_{b=1}^{P} u^*(\xi_{\mathbf{i}}, \eta_{\mathbf{ab}})\left|J_{\rho_\theta}\right|\rho w_\rho\right) w_\theta\right)\end{aligned} \tag{34}$$

onde $\left|J_{\rho\theta}\right|$ é o jacobiano correspondente à transformação de coordenadas retangulares em coordenadas polares, η_{ab} são os pontos de Gauss, w_ρ e w_θ são os pesos de Gauss, retirados de uma tabela específica para integração de elementos triangulares em coordenadas polares (BREBBIA; TELLES; WROBEL, 1984) [p. 450].

As equações que relacionam as coordenadas globais (x_1, x_2, x_3) e locais dos pontos

de Gauss (η_1, η_2) para elementos triangulares são:

$$\begin{aligned} x_1 &= \eta_1\, x_1^{[1]} + \eta_2\, x_1^{[2]} + (1 - \eta_1 - \eta_2)\, x_1^{[3]} \\ x_2 &= \eta_1\, x_2^{[1]} + \eta_2\, x_2^{[2]} + (1 - \eta_1 - \eta_2)\, x_2^{[3]} \\ x_3 &= \eta_1\, x_3^{[1]} + \eta_2\, x_3^{[2]} + (1 - \eta_1 - \eta_2)\, x_3^{[3]} \end{aligned} \tag{35}$$

onde $(x_1^{[e]}, x_2^{[e]}, x_3^{[e]})$ são as coordenadas do e-ésimo vértice do elemento.

2.3 ALGORITMO DO NÚCLEO DO PROGRAMA ORIGINAL

Esta seção possui duas subseções. A primeira descreve a versão de código inicial **s0.0** e a segunda apresenta algoritmos, subrotinas e estruturas de armazenamento de dados dessa versão.

2.3.1 O programa legado inicial e a versão s0.0

A versão do programa legado é composta de aproximadamente 65.000 linhas de código. Dentre as características do programa, incluem-se: capacidade de trabalhar com subregiões finitas ou infinitas, regiões simétricas, corpos delgados, condições de contorno não-lineares, fontes pontuais e cálculo da condutividade do solo com dados obtidos pelo Método de Wenner.

O núcleo do programa, em torno do qual todas as funcionalidades foram construídas, é uma implementação da Formulação Clássica do Método dos Elementos de Contorno para Problemas do Potencial, capaz de lidar com elementos de contorno triangulares ou quadrilaterais, de geometria plana ou interpolada por funções lineares ou quadráticas, com valores das funções nos elementos de contorno constantes ou interpolados linear ou quadraticamente.

A primeira tarefa executada neste trabalho foi o isolamento do núcleo do programa. Por simplicidade, foi escolhido apenas um tipo de discretização do contorno: elementos quadrilaterais planos com valores das funções u e q constantes em cada elemento. Desse modo,

das 65.000 linhas de código originais, foram isoladas cerca de 6.700 linhas e 67 subrotinas. Essa nova versão do código foi denominada versão **s0.0**.

2.3.2 Algoritmo do núcleo do programa original

O algoritmo do MEC clássico é ilustrado pelo fluxograma da Fig. 1. A figura também contém a complexidade computacional das subrotinas e o custo de armazenamento das estruturas de dados do programa em memória RAM, com n graus de liberdade.

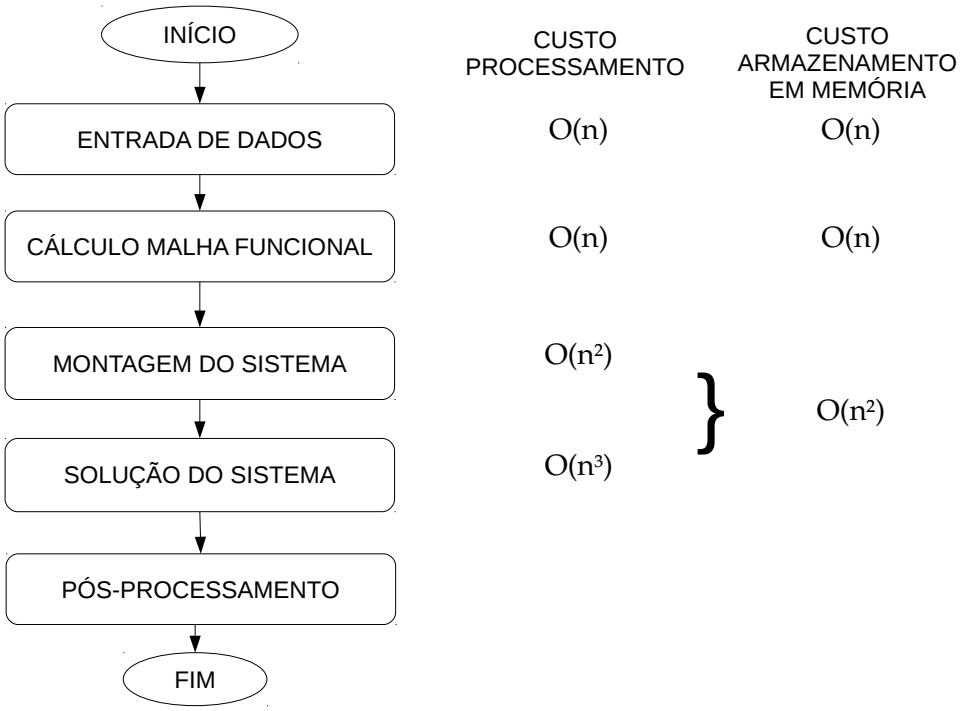


FIGURA 1 – Algoritmo da Formulação Clássica do Método dos Elementos de Contorno

Entrada de Dados:

As principais estruturas de armazenamento de dados utilizadas nessa rotina são: vetor com as coordenadas tridimensionais dos **nós geométricos** (vértices dos elementos de contorno); vetor com **elementos de contorno** e as **incidências** dos nós geométricos nos elementos (lista ordenada dos nós geométricos pertencentes a um determinado elemento); vetores com **tipo de condição de contorno** do elemento (essencial ou natural) e **valores prescritos**.

Embora a versão **s0.0** não inclua as subrotinas do programa legado relacionadas com outras discretizações do contorno, a estrutura de armazenamento de dados original foi mantida, com o objetivo de facilitar uma eventual reinserção de qualquer uma das funcionalidades originais. Desse modo, herdam-se variáveis que indicam a geometria (tringular ou quadrilateral), a funcionalidade (constante, linear ou quadrática) e a continuidade dos lados dos elementos de contorno.

Cálculo da Malha Funcional:

O número de nós funcionais (n) determina a dimensão da matriz do sistema linear. Para a discretização do contorno adotada, o único nó funcional de cada elemento está localizado no centro do elemento e a **malha funcional** é formada por um vetor real contendo as coordenadas globais dos nós funcionais, definidas pelas Eqs. 17 e 16.

Embora contendo informação redundante para malhas de contorno com elementos planos e constantes, outra estrutura de dados herdada do programa legado pela versão **s0.0** é o **vetor das incidências dos nós funcionais**, com o mesmo intuito de permitir uma eventual reinserção da capacidade de trabalhar com outras discretizações.

Montagem do Sistema Linear:

A rotina de montagem do sistema linear do programa legado apresenta peculiaridades que se mostram eficientes mesmo em sua versão sequencial.

Nota-se no Algoritmo 2.1 que a matriz é calculada por colunas, o que apresenta dois pontos positivos. O primeiro tem relação com uma peculiaridade da linguagem Fortran: ao contrário do que acontece em outras linguagens, como C, as matrizes são armazenadas em memória por colunas, isto é, dois elementos consecutivos em uma mesma coluna ($A(i,j)$ e $A(i+1,j)$) são armazenados em endereços consecutivos na memória (como as variáveis reais utilizam precisão dupla, cada elemento da matriz ocupa dois endereços de memória). Assim, a distância entre os endereços acessados a cada iteração (linhas 14 e 17 no Algoritmo 2.1) é a menor possível. Essa distância possui um significado importante na otimização de programas e recebe a denominação técnica de ***loop stride***. O *loop stride* pequeno da rotina permite uma utilização eficiente da hierarquia de memória apresentada na Sec. 3.1.2 . Por ora, note-se que a montagem realizada por linhas apresentaria um *loop stride* n vezes maior que a montagem

Algoritmo 2.1 Montagem da Matriz do Sistema Linear

1: **para** j $\leftarrow$ 1 **até** n **faça**
2: **para** i $\leftarrow$ 1 **até** n **faça**
3: $\xi \leftarrow$ NÓ_FUNCIONAL(i)
4: **se** i = j **então**
5: $h \leftarrow 0$
6: $g \leftarrow \int_{\Gamma_j} u^*(\xi, \mathbf{x})\, \mathrm{d}\Gamma_j$
7: **senão**
8: $P \leftarrow$ NÚMERO_DE_PONTOS_DE_GAUSS(i, j)
9: $h \leftarrow \sum_{a=1}^{P} \sum_{b=1}^{P} q^*(\xi, \eta_{\mathbf{ab}}) |J|\, w_b w_a$
10: $g \leftarrow \sum_{a=1}^{P} \sum_{b=1}^{P} u^*(\xi, \eta_{\mathbf{ab}}) |J|\, w_b w_a$
11: **fim se**
12: **se** potencial prescrito **então**
13: $Y(j) \leftarrow g *$ VALORES_PRESCRITOS(j)
14: $A(i, j) \leftarrow h$
15: **senão**
16: $Y(j) \leftarrow -h *$ VALORES_PRESCRITOS(j)
17: $A(i, j) \leftarrow -g$
18: **fim se**
19: **fim para**
20: **fim para**

por colunas.

O segundo ponto positivo é a possibilidade de reutilização de resultados nas integrações numéricas, já que todas as integrações numéricas referentes a uma coluna j ocorrem no mesmo elemento Γ_j. Supondo inicialmente, por simplicidade, que se utiliza o mesmo número de pontos de Gauss (P^2) para todas as integrações numéricas, os P^2 produtos $|J|\, w_b w_a$ das linhas 9 e 10 do Algoritmo 2.1, bem como as P^2 transformações de coordenadas tridimensionais locais ($\eta_{\mathbf{ab}}$) em globais ($\mathbf{x_{ab}}$) (necessárias para a determinação da distância $r = |\xi_{\mathbf{i}} - \mathbf{x_{ab}}|$, que é utilizada nos cálculos de u^* e q^*) podem ser efetuadas uma única vez e reutilizadas $P^2 2(n-1)$ vezes (não se reutiliza esses valores apenas para os elementos das diagonais que são, ou nulos, ou calculados com quatro integrações, conforme Eq. 34 e linhas 5 e 6 do Algoritmo 2.1). Essa reutilização de valores reduz o processamento para problemas com $n >> P$, embora requeira espaço de armazenamento em memória da ordem de $\mathcal{O}(P^2)$.

A linha 8 do Algoritmo 2.1 é um indício de que a quantidade de pontos de Gauss é variável na rotina. De fato, outra técnica que reduz o esforço computacional da montagem da matriz, sem perda de precisão numérica, é a utilização de números variáveis de pontos de Gauss na integração numérica do elemento Γ_j de acordo com a distância ($D = \left|\xi_{\mathbf{i}} - \xi_{\mathbf{j}}\right|$) entre o ponto fonte ($\xi_i \notin \Gamma_j$) e o centro do elemento Γ_j. Distâncias maiores requerem menos pontos de Gauss. Surge então o problema de classificar essas distâncias, que é resolvido na rotina do programa legado através da razão ($R = D/L$) entre a distância D e o comprimento do maior lado do elemento L. Conforme mostra o Algoritmo 2.2, as integrações dos elementos são feitas com 2^2, 4^2 ou 10^2 pontos de Gauss, de acordo com essa razão.

Mesmo para o menor dos problemas apresentados nos Cap. 3 e 4, o valor máximo de todas as razões R encontradas no problema $max(R) >> 15$. Isso sugere a possibilidade de criação de um novo intervalo na rotina de cálculo do número de pontos de Gauss, o que permitiria a integração numérica com 1 ponto de Gauss e o decorrente ganho de processamento para programas sequenciais e para programas paralelos baseados nas ideias de Beer, Smith e Duenser (2008). No entanto, a rotina não foi modificada até o momento. A definição desse novo intervalo implicaria em novos experimentos numéricos, provavelmente com problemas testes mais diversificados do que os utilizados aqui. Além disso, o ganho dessa modificação é

Algoritmo 2.2 Cálculo do número de pontos de Gauss utilizado

```
1: função NÚMERO_DE_PONTOS_DE_GAUSS(i,j)
2:     D ← |ξ_i − ξ_j|   \\ Distância euclidiana entre elementos
3:     L ← MAIOR_LADO(j)   \\ Maior lado do elemento Γ_j
4:     R ← D/L
5:     se R <= 3.0 então
6:         retorna 10
7:     senão se R <= 15.0 então
8:         retorna 4
9:     senão
10:        retorna 2
11:    fim se
12: fim função
```

reduzido nos programas paralelos que utilizam armazenamento em memória disponível, como será detalhado nos próximos capítulos.

3 METODOLOGIA

Este capítulo contém quatro seções. A primeira seção descreve o ambiente computacional utilizado, o *cluster* do Lactec. As seções seguintes descrevem as ferramentas e medidas adotadas, os testes de larga escala e as alterações efetuadas no programa.

3.1 AMBIENTE COMPUTACIONAL

O *cluster* do Lactec é composto de um nó administrativo e dezoito nós de trabalho. Todos os nós utilizam o sistema operacional Suse Linux Enterprise Server, versão 11.

O nó administrativo, utilizado para submissão, agendamento e controle dos programas em lote (*batch jobs*), possui dois multi-processadores Intel Xeon 5690 com seis núcleos de processamento de 3.64 GHz por *chip*, 96 GB de RAM e 1 TB de disco rígido.

Os nós de trabalho, onde as versões do programa foram executadas, são descritos nas próximas três subseções, organizadas de acordo com os objetivos de trabalho apresentados na Sec. 1.2: processamento, memória e comunicação. A Fig. 2 apresenta o esquema de um nó de trabalho.

3.1.1 Processamento

Os **nós de trabalho** do *cluster* possuem 48 GB de memória RAM, dois multi-processadores Intel Xeon 5650 com seis núcleos de processamento de 2.66 GHz, totalizando doze núcleos de processamento por máquina. Cada núcleo de processamento é um processador completo e independente, com seu banco de registradores (RF - *Register File*, na Fig.2) e unidades lógicas e aritméticas.

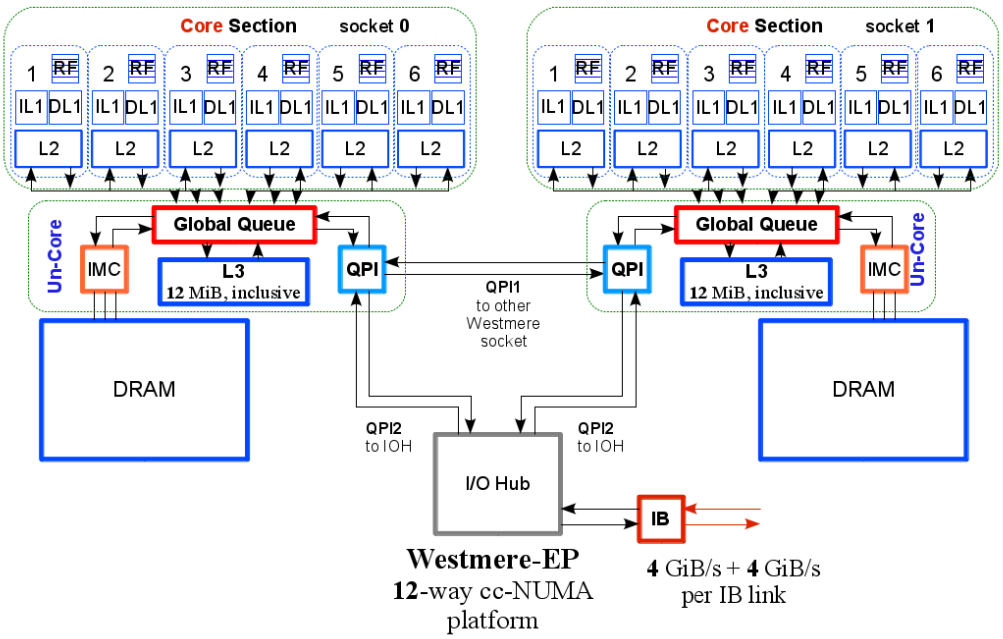


FIGURA 2 – Esquema de uma configuração semelhante a um nó de trabalho do *cluster* do Lactec (retirado de Intel, 2012)

O Intel Xeon 5650 pertence à família de processadores Westmere, que possui um recurso tecnológico chamado *hyperthreading*. Unidades replicadas em cada núcleo de processamento permitem que os seis núcleos de processamento físicos de cada multi-processador possam ser divididos em doze **núcleos de processamento lógicos**. De acordo com *benchmarks* realizados pelo CERN, Organização Europeia para a Pesquisa Nuclear (JARP *et al.*, 2010, p.21), esse recurso pode fornecer, ou um aumento de 12 a 24%, ou redução na velocidade de processamento, dependendo das características de escalabilidade do programa sendo executado. Verificou-se que os testes rodados no presente trabalho se enquadram no segundo caso e por essa razão não se utiliza essa tecnologia. Todas as medidas apresentadas no Cap. 4 foram feitas considerando-se somente a quantidade de núcleos de processamento físicos dos nós de trabalho.

3.1.2 Memória

A Fig. 3 representa valores de referência de um servidor atual para os tempos de acesso da **hierarquia de memória** (HENESSY; PATTERSON, 2012). O tempo de acesso dos registradores é compatível com a duração de um ciclo do relógio do processador. No entanto, a latência da memória RAM, incluindo o tempo de busca e tradução dos endereços virtuais para endereços físicos, chega a ser dezenas ou até centenas de vezes maior que o ciclo do relógio do processador. Para esconder essa latência, o *hardware* possui uma hierarquia de dispositivos de armazenamento mais rápidos, porém com menor capacidade de armazenamento, quanto mais próximos aos núcleos de processamento (os chamados *caches* de memória). Essa técnica explora os **princípios de localidade** espacial e temporal. Uma vez que ocorra acesso a um endereço de memória A qualquer em um programa, o princípio de **localidade espacial** afirma que existe uma probabilidade maior de que os próximos endereços acessados estejam próximos ao endereço A, enquanto o princípio de **localidade temporal** afirma que existe uma probabilidade maior de que o endereço A seja acessado num futuro próximo. Cada nível da hierarquia de memória contém um número inteiro de blocos de memória com endereços contíguos. O tamanho padrão dos blocos de memória para os sistemas linux é 4KB, mas esse tamanho pode ser configurado através do sistema operacional.

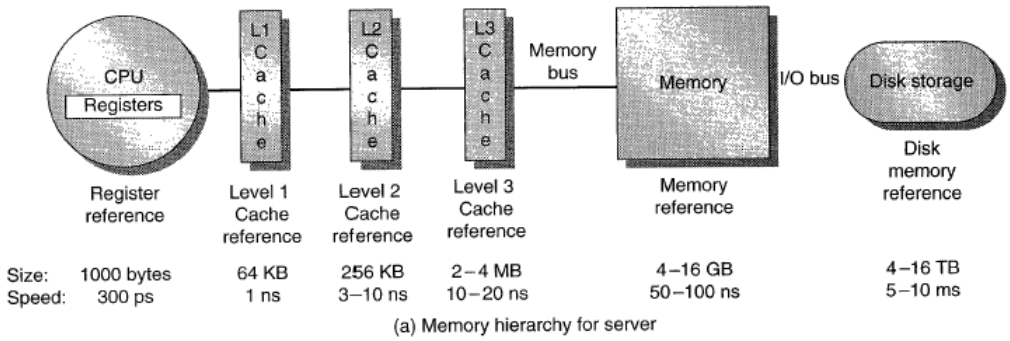


FIGURA 3 – Tempos da hierarquia de memória de um servidor atual (retirado de Henessy and Patterson, 2012)

Voltando à Fig. 2, cada *chip* multiprocessador possui três níveis de *cache* de memória. O primeiro nível (L1) é dedicado a cada núcleo de processamento, possui capacidade de 32 KB e é dividido em cache de dados (DL1) e cache de instruções (IL1). O segundo nível (L2) também é dedicado a cada núcleo de processamento e possui 256 KB. O terceiro nível é

compartilhado por todos os seis núcleos de processamento de um *chip* e possui capacidade de 32 MB.

Uma das razões da ineficiência da tecnologia *hyperthreading*, para as versões do programa apresentadas neste trabalho, é que a divisão de cada núcleo de processamento físico em dois núcleos de processamento lógicos implica no compartilhamento dos níveis L1 e L2 da *cache* de memória entre os dois núcleos lógicos.

Embora todos os núcleos de processamento de um mesmo nó tenham acesso a toda memória física disponível, os tempos de acesso não são iguais. Retornando ao esquema da Fig. 2, os 48 GB de memória RAM são divididos em dois grupos, cada um próximo a um dos *chips* multiprocessadores. Nessa organização de memória, conhecida como NUMA (*Non Uniform Memory Access*), os tempos de acesso dependem da proximidade entre o núcleo de processamento e o dispositivo de memória.

Estudos referentes ao desempenho da memória da arquitetura Nehalem, uma geração anterior à arquitetura Westmere, porém já com uma hierarquia de memória bastante semelhante à apresentada nesta seção, são apresentados por Molka *et al.* (2009).

3.1.3 Comunicação

Os nós do *cluster* estão interligados por duas redes *ethernet GBit* e uma rede Infiniband 4x QDR com capacidade de 10Gb/s. A rede infiniband possui tecnologia especialmente desenvolvida para interconexão de *clusters* de computadores e pode apresentar velocidades 37 vezes superiores às apresentadas pela rede *ethernet* GBit (HPC Advisory Council, 2009).

Esse aumento de velocidade é consequência de melhorias em diversos níveis da comunicação, como por exemplo:

- sem perda de confiabilidade, o protocolo *infiniband* possui menos redundâncias que o protocolo *ethernet*, pois o protocolo *ethernet* foi originalmente concebido para redes de longa distância com altas taxas de perdas de pacotes;
- dispositivos de controle dedicados transferem dados diretamente à memória RAM, sem intervenção dos núcleos de processamento;

- a comunicação é estabelecida ponto-a-ponto através de VLs (*Virtual Lanes*).

3.2 FERRAMENTAS E MEDIDAS

Utilizou-se o compilador Fortran, as bibliotecas numéricas e as implementações MPI e OMP do *Intel Fortran Composer XE 2013 SP1 para Linux*. Os programas foram todos compilados com a *flag* de otimização -O3.

Foram utilizadas as ferramentas de análise de desempenho *Intel VTune Amplifier XE 2013*, *Trace Analyzer and Collector 8.1*, *TAU Performance System 2.23*, desenvolvido pela University of Oregon, *HPCToolkit 5.3.2*, da Rice University, e gprof do GNU, que disponibilizam uma série de métricas de desempenho. Esses dados são obtidos por amostragem estatística, estão diretamente relacionados com a metodologia e os tempos de amostragem configurados para cada análise, as medidas sofrem interferência do processo de medição e, em consequência, ferramentas diferentes fornecem números diferentes para a mesma métrica (GRAHAM; KESSLER; MCKUSICK, 2003; Intel Corp., 2013c, 2013b; Los Alamos National Laboratory, 2012; Rice University, 2013). Apesar disso, essas métricas são úteis como indicativos de gargalos no código, apontando também indícios de suas causas e formas de contorná-los.

Segundo Henessy e Patterson (2009), "o tempo de execução é a única medida de performance válida e inquestionável" (p.54). E essas são as medidas apresentadas nos resultados experimentais do Cap. 4. O sistema operacional fornece três tempos para a execução de cada programa:

- **tempo total decorrido** (*Elapsed time*): tempo decorrido entre o início e o fim do programa;
- **tempo de sistema** (*System time*): tempo da execução do programa em modo privilegiado (por exemplo, tempo para carregar na memória uma subrotina de uma biblioteca chamada pelo programa, tempo para operações de E/S: disco, rede e outros dispositivos);

- **tempo de usuário** (*User time*): tempo da execução do programa em modo usuário.

É possível abstrair a comunicação via rede e o acesso ao disco do tempo de execução, considerando somente o tempo de usuário. Mas neste trabalho, como a comunicação eficiente entre os processos é parte do desafio da programação paralela, considera-se sempre o **tempo total decorrido**, mesmo que esse tempo inclua processos administrativos do ambiente computacional e outros eventuais processos alheios ao programa em questão. Aliás, a inclusão das inevitáveis interferências do ambiente computacional nas medidas permite validar estratégias para diminuir seus efeitos.

Para maior precisão dos resultados, cada medida foi executada dez vezes e foram desconsideradas as duas medidas com desvios maiores.

3.2.1 Comparação entre medidas de tempo

Referências para os conceitos apresentados nesta subseção são Rauber e Rünger (2010) e Henessy e Patterson (2012).

O **ganho no tempo de processamento** (*speedup*) fornecido por uma alteração em um programa é definido como a razão entre os tempos de execução:

$$GANHO = \frac{TEMPO_{anterior}}{TEMPO_{novo}} \tag{36}$$

Assim, se $TEMPO_{novo} = (TEMPO_{anterior}/S)$, o ganho obtido pela alteração é S, isto é, a nova versão é S vezes mais rápida que a anterior.

Quando a alteração efetuada é a paralelização de um programa em NP processadores (ou linhas de execução) e $GANHO = NP$, isto é, $TEMPO_{paralelo} = (TEMPO_{sequencial}/NP)$, o ganho obtido é denominado **ganho linear** (*linear speedup*).

A **eficiência paralela** (EP) leva em conta o número de processadores (NP):

$$EP = \frac{TEMPO_{sequencial}}{TEMPO_{paralelo} \times NP} \tag{37}$$

Considerando-se somente o processamento da computação, isto é, abstraindo-se as

latências da memória e da comunicação, a **máxima eficiência paralela** é obtida quando o ganho é linear:

$$EP_{maximo} = \frac{TEMPO_{sequencial}}{(TEMPO_{sequencial}/NP) \times NP} = 100\%. \tag{38}$$

Embora o *overhead* de comunicação e gerenciamento dos processos paralelos usualmente resultem em $GANHO < NP$, há paralelizações que produzem $GANHO > NP$. Nesse caso, o ganho e a eficiência paralela são denominados **ganho superlinear** (*superlinear speedup*) e **eficiência superlinear**. Esse resultado, surpreendente a princípio, geralmente é devido a uma melhor utilização dos *caches* de memória quando os dados do problema são divididos entre os diversos processos paralelos (RAUBER; RÜNGER, 2010, p.163). Algumas versões de código deste trabalho apresentam resultados superlineares.

3.2.2 Leis de Amdahl e de Gustafson

Todo programa paralelo possui partes sequenciais. A ideia de que a fração sequencial do programa impõe um limite ao ganho obtido com a paralelização do código foi apresentada por Amdahl (1967). Atualmente essa ideia é conhecida como a **Lei de Amdahl** e é geralmente formulada como segue. Sejam f_s e f_p as frações sequencial e paralela de um dado programa, tais que $f_s + f_p = 1$. Se $TEMPO_{seq}$ é o tempo de execução do programa processado sequencialmente, então o tempo de execução da parte serial é $f_s \times TEMPO_{seq}$ e, supondo que a parte paralela seja linearmente paralelizável em NP processadores, o tempo de execução da parte paralela é $(f_p \times TEMPO_{seq})$. Portanto, o ganho obtido com a paralelização é dado por:

$$\begin{aligned} GANHO_{Amdahl} &= \frac{TEMPO_{seq}}{f_s \times TEMPO_{seq} + \frac{f_p \times TEMPO_{seq}}{NP}} \\ &= \frac{1}{f_s + \frac{1-f_s}{NP}} < \frac{1}{f_s} \end{aligned} \tag{39}$$

Ainda segundo Amdahl, valores típicos da fração serial para a época em que o artigo

foi escrito giravam em torno de 30 a 40%. Lembrando que a entrada de dados era realizada através de cartões perfurados, incluindo também a leitura dos programas, e a saída de dados utilizava impressoras matriciais, esses valores parecem razoáveis. Aplicando o menor desses valores à Eq. 39, se $f_s = 30/100$, então $GANHO \to 3,\bar{3}$ quando $NP \to \infty$. Isto é, mesmo considerando que a fração paralela do código seja idealmente paralelizável, uma fração serial de 30% imporia um limite de $3,\bar{3}$ ao ganho adquirido através da paralelização, não importando o número de processadores disponíveis.

Duas décadas mais tarde, Gustafson (1988) afirma ter conseguido, entre outros resultados semelhantes, um ganho de 1021 utilizando 1024 processadores para um programa elastoestático. O autor argumenta que a condição modelada pela Lei de Amdahl, ou seja, a utilização de mais processadores para a solução do mesmo problema, teria interesse puramente acadêmico e a motivação prática para a adição de processadores a um sistema computacional seria a solução de problemas maiores. Além disso, a fração sequencial diminuiria com o aumento do problema. Vale lembrar que essa hipótese é razoável para o MEC clássico, com complexidade $\mathcal{O}(n)$ para as rotinas de entrada e saída e $\mathcal{O}(n^2)$ e $\mathcal{O}(n^3)$ para as rotinas dos cálculos. Uma nova formulação, denominada **ganho em escala** (*scaled speedup*) pelo autor e hoje conhecida como **Lei de Gustafson**, foi apresentada. Sejam $TEMPO(NP)$ o tempo de execução do programa paralelo normalizado com NP processadores e F_S a fração sequencial, tais que $TEMPO(NP) = F_S + (1 - F_S) = 1$. O tempo de execução do programa sequencial é dado por $TEMPO_{SEQ} = F_S + (1 - F_S) \times NP$.

$$\begin{aligned} GANHO_{Gustafson} &= \frac{TEMPO_{SEQ}}{TEMPO(NP)} \\ &= \frac{F_S + (1 - F_S) \times NP}{1} \\ &= F_S + (1 - F_S) \times NP \end{aligned} \tag{40}$$

Gustafson conclui mostrando que, se o tamanho do problema aumentar com o número de processadores de modo a manter o tempo de execução constante, a fração serial diminui e o programa apresenta um ganho quase linear.

Atualmente, os pontos de vista de ambos autores são utilizados. Denomina-se a vari-

ação do ganho fornecido pelo programa paralelo de **escalabilidade forte** quando o programa é usado na solução de um mesmo problema com um número variável de processadores e **escalabilidade fraca** quando o tamanho do problema aumenta com o número de processadores.

Shi (1996) mostra que as frações sequenciais da lei de Amdahl (f_s) e da lei de Gustason (F_s) não são equivalentes e se a relação

$$f_s = \frac{1}{1 + \frac{(1-F_S)\times NP}{F_S}} \tag{41}$$

é definida, ambas as leis são matematicamente equivalentes.

3.3 TESTES DE LARGA ESCALA

Como teste, foi escolhido um problema simples, com o intuito de simplificar a verificação dos resultados e o refinamento arbitrário da malha de contorno. O domínio do problema é um cubo e as condições de contorno são potencial 0 e 1 em duas faces opostas e fluxo 0 nas demais faces, conforme a Fig. 4. O número de elementos de contorno varia de 3.456 a 194.400. A discretização do domínio é realizada com elementos quadrilaterais planos e os valores das funções são considerados constantes em cada elemento. Assim, como discutido no Cap. 2, a dimensão do sistema linear obtido é equivalente ao número de elementos do contorno.

A ferramenta utilizada para a geração dos problemas foi o GiD 9.0.6.

3.4 ALTERAÇÕES NO PROGRAMA E CONVENÇÃO DAS VERSÕES DE CÓDIGO

Esta seção descreve a progressão das versões de código geradas neste trabalho e possui cinco subseções, dedicadas a cada uma das etapas de programação. Além de algoritmos, são apresentados também resultados obtidos com ferramentas de análise de desempenho.

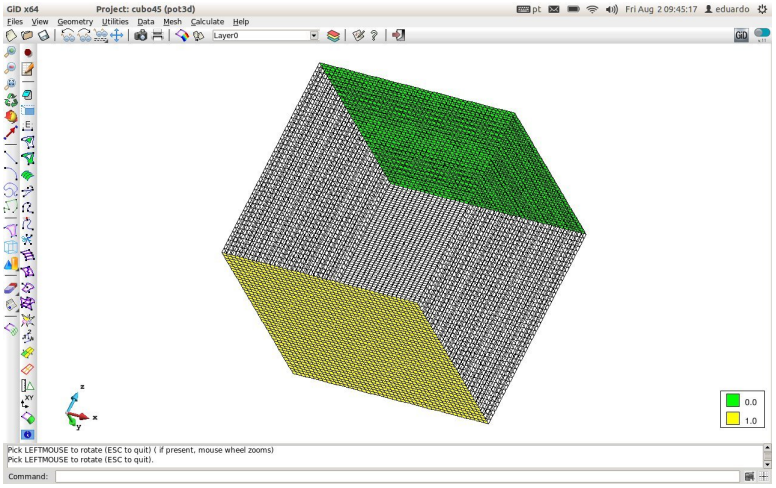


FIGURA 4 – Formulação do problema dos experimentos

As alterações no programa foram realizadas nas seguintes etapas:

1. alterações na solução do sistema linear,
2. alterações na montagem do sistema linear,
3. paralelização MPI,
4. armazenamento em memória disponível,
5. paralelização híbrida.

As três primeiras etapas são replicação do trabalho de Beer, Smith e Duenser (2008), que foca a montagem elemento-por-elemento e a paralelização MPI de um problema elastostático [cap.8], com as diferenças detalhadas adiante. As duas últimas etapas são específicas deste trabalho.

Durante o desenvolvimento foram geradas quinze versões do programa. Foi adotada a seguinte convenção para nomenclatura das versões de código :

- Uma letra inicial indica se o código é sequencial (s), paralelo com OpenMP (p), paralelo com MPI (P) ou híbrido MPI e OpenMP (h).
- O primeiro número, antes do ponto, indica uma alteração maior realizada no código, correspondente a uma das cinco fases do trabalho.

- O segundo número, após o ponto, indica uma alteração menor, correspondente a um passo dentro de uma fase.
- Uma letra após o último número indica apenas uma otimização de uma versão.

A Tab.1 contém as versões de código de cada etapa.

Entre as duas versões de código geradas na quarta etapa (**P3.0** e **P3.2**), foram realizados alguns experimentos em código sequencial, que são apresentados nas versões **s3.1.0**, **s3.1.1** e **s3.1.2**.

TABELA 1 – tabela de versões do código

Etapa	Versão	Comentário
Solução	s0.0	versão original
	p0.1	biblioteca numérica LAPACK
	s0.2	método iterativo BiCGStab
Montagem	s1.0	montagem elemento-por-elemento
	s1.0a	otimização no acesso às estruturas de dados
Paralelização MPI	P2.0	sem load balance
	P2.0.1	load balance simples
	P2.0.2	load balance adaptativo
	P2.1	cálculos do lado direito da equação, da malha funcional e da diagonal paralelizados
Armazenamento em Memória Disponível	P3.0	área de armazenamento geral
	s3.1.0	teste serial: colunas armazenadas
	s3.1.1	teste serial: AXPY BLAS
	s3.1.2	teste serial: colunas armazenadas (sem cópia)
	P3.2	área de armazenamento local
Paralelização Híbrida	h4.0	paralelização híbrida (OMP + MPI)

3.4.1 Alterações na solução do sistema linear

No primeiro passo da paralelização de Beer, Smith e Duenser (2008)[p.206], a rotina de solução direta do sistema linear é substituída pela rotina iterativa BiCGStab, que é adequada para a montagem da matriz elemento-por-elemento do segundo passo.

O objetivo desta etapa é comprovar o bom desempenho desta rotina. Para isso são realizadas comparações entre três rotinas de solução do sistema linear: a solução direta

original, uma solução direta otimizada e a rotina fornecida por Beer et al. A cada rotina de solução corresponde uma versão do programa.

3.4.1.1 s0.0: versão original

A versão original utiliza uma rotina de solução direta através de **eliminação de Gauss** e apresenta complexidade computacional $\mathcal{O}(n^3)$.

Os dados da Tab.2 foram obtidos com a ferramenta *paraprof* da suíte TAU e representam as porcentagens do tempo de execução do programa utilizado pelas rotinas de entrada e saída, montagem e solução do sistema linear. Esses números justificam o foco inicial do trabalho na rotina de solução do sistema.

As duas últimas linhas da Tab.2 sugerem a aplicação da Lei de Ahmdahl para achar o ganho máximo de $333,3$ com a paralelização da rotina de solução devido aos $0,3\%$ da rotina de montagem, mas medidas com outras ferramentas de análise mostram que essas porcentagens são apenas bons indicativos do comportamento das rotinas e não devem ser considerados mais do que isso.

TABELA 2 – v0.0 - Perfil de Execução: Porcentagem do tempo de execução do programa utilizada pelos grupos de subrotinas

Elementos	Entrada e Saída (%)	Montagem Sist. Lin. (%)	Solução Sist. Lin. (%)
3456	0.4	4.1	95.5
5400	0.1	1.8	98.1
12150	0.0	0.5	99.5
13824	0.0	0.3	99.7
24576	0.0	0.3	99.7

3.4.1.2 p0.1: solução com biblioteca numérica

Nessa versão, a rotina de solução original é substituída pela rotina DGESV da **biblioteca numérica LAPACK** (Linear Algebra PACKage). O LAPACK, por sua vez, utiliza a biblioteca BLAS (Basic Linear Algebra Subprograms) para operações entre vetores e matrizes. O BLAS é cuidadosamente otimizado para uma ampla variedade de plataformas e

paralelizado via OMP (ANDERSON *et al.*, 1999; Intel Corp., 2013a). Foram utilizadas as implementações do LAPACK e do BLAS fornecidas com a biblioteca numérica Intel MKL v11.1 (Math Kernel Library). A rotina DGESV é uma implementação do método direto e também possui complexidade computacional $\mathcal{O}(n^3)$.

Alguns dados obtidos com a ferramenta VTune mostram o comportamento da nova rotina de solução:

- para o problema de 3456 (3K) elementos de contorno, a rotina de solução gera 12 linhas de execução que são executadas paralelamente em 500 milisegundos (aproximadamente 10% do tempo de execução total do programa).
- para o problema de 38400 (38K) elementos, a rotina de solução gera 24 linhas de execução que são executadas paralelamente em 150 segundos (27% do tempo de execução total).

Observa-se que o número de linhas de execução geradas depende do tamanho do problema a ser resolvido e que, como esperado, o custo administrativo adicional decresce com o tamanho do problema. Lembrando que esses números são apenas indicativos, nota-se também que a porcentagem do tempo de execução da solução é significativamente menor que os 90% da versão anterior.

3.4.1.3 s0.2: solução iterativa

Na versão **s0.2** foi realizada a substituição da rotina de solução do sistema pelo **método iterativo BiCGStab(l)**, com a implementação em Fortran fornecida por Beer, Smith e Duenser (2008). O BiCGStab(l) pertence à família Krylov, cuja compatibilidade com as matrizes densas do BEM tem sido demonstrada em diversos trabalhos (BARRA *et al.*, 1992; GRAMA; KUMAR; SAMEH, 1996; VALENTE; PINA, 2006). O esforço computacional desses métodos é proporcional ao quadrado da dimensão da matriz do sistema linear n^2 e ao número de iterações i (XIAO; CHEN, 2007). Embora o número de iterações não seja refletido na complexidade do algoritmo, pois $i << n$ e $\mathcal{O}(in^2) = \mathcal{O}(n^2)$.

Os métodos da família Krylov são derivados do Teorema de Cayley-Hamilton, que

afirma que a inversa da matriz A é uma combinação linear das n potências de A, $\{A^0 = I, A^1 = A, \ldots, A^{n-1}\}$. Dada uma matriz A de dimensão $n \times n$ e posto n e um vetor $\mathbf{b}$ não nulo de dimensão n, define-se o **espaço Krylov de ordem** r, denotado por K_r, $1 \leq r \leq n$, como o espaço linear gerado pelo conjunto $\{A^0\mathbf{b} = \mathbf{b}, \mathbf{Ab}, \mathbf{A^2b}, \ldots, \mathbf{A^{r-1}b}\}$. A cada iteração r, é realizada uma busca da solução no espaço K_r. A diferença entre os métodos dessa família é o algoritmo de busca. Teoricamente a solução seria encontrada em, no máximo, n iterações. No entanto, a solução aproximada é geralmente obtida com precisão suficiente com um número de iterações i muito menor que a dimensão n do sistema. Por outro lado, acúmulos de erros de aproximação numérica podem prejudicar a convergência desses métodos em alguns casos especiais (KELLEY, 1995).

O método dos Gradientes Conjugados é utilizado apenas para sistemas lineares com matrizes simétricas. Os métodos BiCG (*Bi-Conjugate Gradient*), GMRES (*Generalized Minimal Residual*) e CGS (*Conjugate Gradient Squared*) são utilizados para a solução de sistemas com matrizes não-simétricas. O método BiCGStab (*Bi-Conjugate Gradient Stabilized Method*), criado por Vorst (1992), pode ser descrito como uma combinação dos métodos BiCG e GMRES. O método BiGCStab(l) é uma generalização do BiCGStab, que com $l = 1$ corresponde exatamente ao BiCGStab e com $l = 4$ produz bons resultados em muitos casos em que o BiCGStab apresenta estagnação, por exemplo, para sistemas com matrizes reais e autovalores puramente imaginários, ou com autovalores cuja parte real seja muito próxima de zero (SLEIJPEN; VORST; FOKKEMA, 1994).

O Algoritmo 3.1 foi retirado de Sleijpen e Fokkema (1993) e descreve o método BiCGStab(l). Cada iteração do algoritmo possui duas partes: uma parte MR (*Minimal Residual*), que corrige a direção de busca $\mathbf{u}$ com o método dos resíduos mínimos, e uma parte BiCG, que possui l iterações internas e fornece um polinômio MR de grau l a cada l-ésima iteração. Na parte BiCG, há duas multiplicações da matriz A por um vetor. A primeira multiplicação (linha 17) ocorre na atualização do vetor de busca $u_{k+1} = Au_k$ e a segunda multiplicação (linha 22) ocorre no cálculo do resíduo $r_{k+1} = Ar_k$. Isso implica em **oito multiplicações** da matriz do sistema pelos vetores u e r a cada iteração. Esse número será importante para determinar o aumento do processamento causado pela montagem elemento-por-elemento da

próxima seção.

3.4.2 Montagem do sistema elemento-por-elemento

O segundo passo na paralelização de Beer, Smith e Duenser (2008) é a montagem do sistema elemento-por-elemento. A rotina BiCGStab utiliza a matriz $A_{n\times n}$ apenas nos produtos $v = Ap$ e $t = As$, mas o cálculo desses produtos não exige que a matriz esteja completamente armazenada em memória. Os autores modificam a parte do código que efetua esses produtos para que utilize uma linha da matriz de cada vez e cada linha seja novamente montada a cada vez que for utilizada. Desse modo, o requisito de armazenamento do novo programa poderia possuir complexidade linear, da ordem de $O(n)$. No entanto, como os autores armazenam os coeficientes das matrizes dos elementos $H_{n\times n}$ e $G_{n\times n}$, o custo de armazenamento da matriz do sistema é duplicado. Portanto, a implementação da montagem elemento-por-elemento dos autores **aumenta o custo de armazenamento do programa**, apesar de facilitar a paralelização.

Nesta segunda etapa do trabalho, não se armazena as matrizes H e G e o custo de armazenamento do sistema é da ordem de $\mathcal{O}(n)$. No entanto, se i é o número de iterações necessárias para a convergência do BiCGStab(l) e $l = 4$ é o grau do polinômio de busca, o custo computacional da montagem da matriz é multiplicado por $2li = 8i$. Para os problemas rodados, o número de iterações i oscilou entre 20 e 30. Desse modo, $160 \leq 8i \leq 240$, isto é, durante os experimentos, a matriz do sistema foi montada entre 160 a 240 vezes a cada execução do programa. Esse aumento no custo da montagem permitiu otimizar a rotina de montagem do sistema, que na etapa anterior tinha pouca visibilidade nas ferramentas de análise de desempenho.

Portanto, esta etapa produziu duas versões:

- **s1.0**: montagem elemento-por-elemento.
- **s1.0a**: otimização do código.

Algoritmo 3.1 BiCGStab(l) de SLEIJPEN, 1993

1: $\mathbf{x_0} \leftarrow \mathbf{0}$ \\ aproximação inicial
2: $\mathbf{r_0} \leftarrow \mathbf{b}$ \\ resíduo inicial
3: $\mathbf{u_{-1}} \leftarrow \mathbf{0}$ \\ u é direção de busca
4: $\rho_0 \leftarrow \omega \leftarrow 1$
5: $\alpha \leftarrow 0$
6: $i \leftarrow 0$ \\ número de iterações externas
7: $k \leftarrow -l$
8: **enquanto** $(\mathbf{r_{k-l}} \bullet \mathbf{r_{k-l}}) > \mathbf{tolerancia}$ **faça**
9: $i \leftarrow i + 1; k \leftarrow k + l$
10: $\hat{u_0} \leftarrow \mathbf{u}_{k-1}; \quad \hat{r_0} \leftarrow \mathbf{r}_k; \quad \hat{x_0} \leftarrow \mathbf{x}_k$
11: $\rho_0 \leftarrow -\omega\rho_0$
\\ PARTE Bi-CG
12: **para** $j \leftarrow 0$ **até** $l - 1$ **faça**
13: $\rho_1 = \hat{r}_j \bullet \tilde{r}_0; \quad \beta \leftarrow \alpha(\rho_1/\rho_0); \quad \rho_0 \leftarrow \rho_1$
14: **para** $i \leftarrow 0$ **até** j **faça**
15: $\hat{u}_i \leftarrow \hat{r}_i - \beta\hat{u}_i$
16: **fim para**
17: $\hat{u}_{j+1} \leftarrow A\hat{u}_j$ \\ Primeiro produto da matriz A
18: $\gamma \leftarrow \hat{u}_{j+1} \bullet \tilde{r}_0; \quad \alpha \leftarrow \rho_0/\gamma$
19: **para** $i \leftarrow 0$ **até** j **faça**
20: $\hat{r}_i \leftarrow \hat{r}_i - \alpha\hat{u}_{i+1}$
21: **fim para**
22: $\hat{r}_{j+1} \leftarrow A\hat{r}_j$ \\ Segundo produto da matriz A
23: **fim para**
\\ PARTE MR
24: **para** $j \leftarrow 1$ **até** l **faça**
25: **para** $i \leftarrow 1$ **até** $j - 1$ **faça**
26: $\tau_{ij} \leftarrow (\hat{r}_j \bullet \hat{r}_i)/\sigma_i$
27: $\hat{r}_j \leftarrow \hat{r}_j - \tau_i j\hat{r}_i$
28: **fim para**
29: $\sigma_j \leftarrow (\hat{r}_j \bullet \hat{r}_j); \quad \gamma'_j \leftarrow (\hat{r}_0 \bullet \hat{r}_j)/\sigma_j;$
30: **fim para**
31: $\gamma_l \leftarrow \gamma'_l; \quad \omega \leftarrow \gamma_l$
32: **para** $j \leftarrow l - 1$ **até** 1 **faça**
33: $\gamma_j \leftarrow \gamma'_l - \sum_{i=j-1}^{l-1} \tau_{ij}\gamma_i$
34: **fim para**
35: **para** $j \leftarrow 1$ **até** $l - 1$ **faça**
36: $\gamma''_j \leftarrow \gamma_{j+1} + \sum_{i=j+1}^{l-1} \tau_{ji}\gamma_{i+1}$
37: **fim para**
38: $\hat{x}_0 \leftarrow \hat{x}_0 + \gamma_1\hat{r}_0; \quad \hat{r}_0 \leftarrow \hat{r}_0 - \gamma'_1\hat{r}_l; \quad \hat{u}_0 \leftarrow \hat{u}_0 - \gamma_1\hat{u}_l$
39: **para** $j \leftarrow 1$ **até** $l - 1$ **faça**
40: $\hat{u}_0 \leftarrow \hat{u}_0 - \gamma_j\hat{u}_j$
41: $\hat{x}_0 \leftarrow \hat{x}_0 + \gamma''_j\hat{r}_j$
42: $\hat{r}_0 \leftarrow \hat{r}_0 - \gamma'_j\hat{r}_j$
43: **fim para**
44: $\mathbf{u}_{k+l-1} \leftarrow \hat{u}_0; \quad \mathbf{r}_{k-l} \leftarrow \hat{r}_0; \quad \mathbf{x}_{k-l} \leftarrow \hat{x}_0$
45: **fim enquanto**

3.4.2.1 s1.0: montagem elemento-por-elemento

A implementação de Beer, Smith e Duenser (2008) realiza o cálculo por linhas, enquanto a versão **s1.0** herda os pontos fortes da implementação original **s0.0** expostos na seção 2.3.2, calculando a matriz por colunas. Além disso, como já foi mencionado, as matrizes H e G não são armazenadas e os elementos da matriz do sistema são efetivamente calculados neste trabalho.

Conforme mostra a Fig. 5, a rotina de montagem do sistema é dividida em três subrotinas: cálculo do lado direito da equação, cálculo da diagonal de A e cálculo de uma coluna de A. Somente a última é chamada $2l$ vezes a cada iteração.

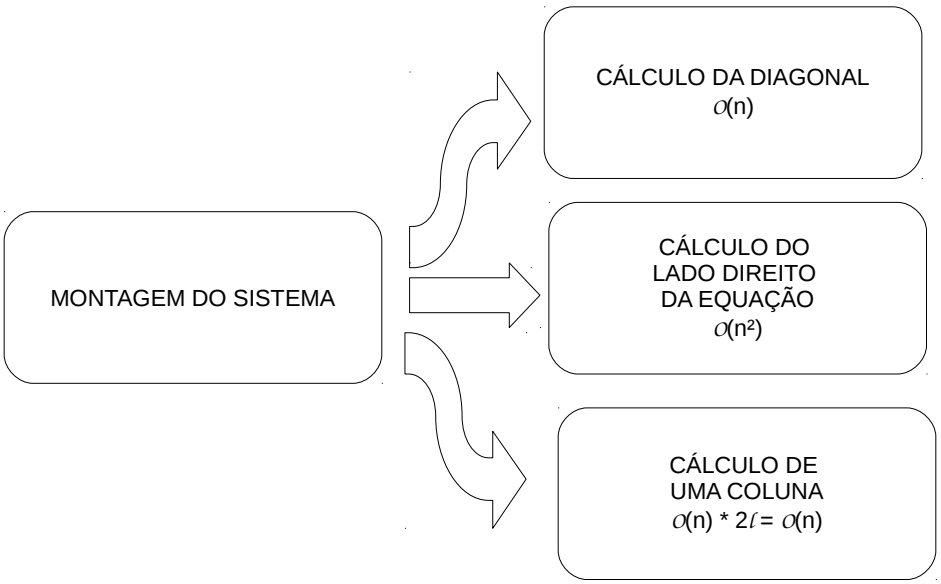


FIGURA 5 – Subrotina de montagem é dividida em três subrotinas: cálculo da diagonal, cálculo do lado direito da equação e cálculo da i-ésima coluna.

Essa divisão é bastante conveniente, pois simplifica a rotina de montagem original do Algoritmo 2.1: como a diagonal é calculada apenas uma vez e armazenada, a montagem da coluna apenas copia o valor da diagonal e não necessita chamar a rotina de integração numérica correspondente à Eq.34, que subdivide o elemento quadrilateral para contornar o ponto de singularidade no cálculo de g_{ii}; como o lado direito da equação já foi calculado a montagem da coluna j necessita calcular apenas um dos valores h_{ij} ou g_{ij}, dependendo do

tipo de condição de contorno.

O Algoritmo 3.2 descreve simplificadamente a rotina de montagem da j-ésima coluna, já retirados os cálculos da diagonal e do lado direito da equação. Na linha 2, a subrotina *Constantes(j)* calcula as constantes referentes ao j-ésimo elemento de contorno e que são reutilizadas nos cálculos de cada elemento da coluna, conforme discutido na Seção 2.3.2. Na linha 13, a posição $C(i)$ recebe o valor armazenado na i-ésima posição do vetor $DIAG_n$ que armazena os elementos diagonais da matriz do sistema linear.

Algoritmo 3.2 Subrotina da Montagem de Uma Coluna da Matriz do Sistema Linear

1: **função** MONTA_COLUNA(j)
2: CONSTANTES(j) \\ Cálculo das constantes η_{ab},$|J|\, w_b w_a$
3: **para** i $\leftarrow$ 1 **até** n **faça**
4: **se** $i \neq j$ **então**
5: $\xi \leftarrow$ NÓ_FUNCIONAL(i)
6: $P \leftarrow$ NÚMERO_DE_PONTOS_DE_GAUSS(i, j)
7: **se** potencial prescrito **então**
8: $C(i) \leftarrow \sum_{a=1}^{P}\sum_{b=1}^{P} q^*(\xi, \eta_{\mathbf{ab}})|J|\, w_b w_a$
9: **senão**
10: $C(i) \leftarrow \sum_{a=1}^{P}\sum_{b=1}^{P} u^*(\xi, \eta_{\mathbf{ab}})|J|\, w_b w_a$
11: **fim se**
12: **senão**
13: $C(i) \leftarrow DIAG(i)$
14: **fim se**
15: **fim para**
16: **retorna** C
17: **fim função**

Finalmente, a rotina da solução do sistema linear foi modificada para o cálculo dos produtos $u_{k+1} = Au$ e $r_{k+1} = Ar$, conforme as equações abaixo, onde $u(i)$ e $r(i)$ são os i-ésimos elementos dos vetores u e r e $A(:, i)$ representa a i-ésima coluna de A, de modo que $u(i)A(:, i)$ e $r(i)A(:, i)$ são produtos de escalar e vetor:

$$u_{k+1} = \sum_{i=1}^{n} u(i)A(:, i) \qquad r_{K+1} = \sum_{i=1}^{n} r(i)A(:, i) \tag{42}$$

Essas mudanças na rotina de solução podem ser vistas nos algoritmos 3a e 3b. Nota-se que o Algoritmo 3b está pronto para ser paralelizado, pois os produtos $u(i)A(:, i)$ podem

(a) s0.2: BiCGStab

```
1: função BICGSTAB_L
        ⋮
        ⋮
2:      u ← Au
        ⋮
        ⋮
        ⋮
3:      r ← Ar
        ⋮
        ⋮
        ⋮
4: fim função
```

(b) s1.0: BiCGStab

```
1: função BICGSTAB_L
        ⋮
2:      para i ← 1 até n faça
3:          Col ← MONTA_COLUNA(i)
4:          u1 ← u1 + u(i) * Col
5:      fim para
        ⋮
6:      para i ← 1 até n faça
7:          Col ← MONTA_COLUNA(i)
8:          r1 ← r1 + r(i) * Col
9:      fim para
        ⋮
10: fim função
```

TABELA 3 – modificação na rotina BiCGStab para utilizar montagem elemento-por-elemento

ser calculados independentemente e somados em qualquer ordem.

3.4.2.2 s1.0a: otimização da versão s1.0

Dois fatores justificam o processo de otimização após a implementação da montagem elemento-por-elemento. Em primeiro lugar, somente nesta etapa a montagem passa a ocupar uma porcentagem significativa do tempo de processamento. Nas etapas anteriores, a solução ocupava a maior parte do tempo de processamento e as rotinas de montagem tinham pouquíssima visibilidade nas ferramentas de análise de desempenho. Em segundo lugar, é uma boa prática otimizar um código antes de paralelizá-lo, pois, dentre outras razões, as áreas que se beneficiariam de otimizações são muito mais difíceis de serem detectadas em códigos paralelos (GRAMA *et al.*, 2003; RAUBER; RÜNGER, 2010).

A versão **s1.0a** foi obtida com otimizações manualmente introduzidas no código e indicadas pelas análises da versão **s1.0** realizadas com as ferramentas de desempenho. As alterações mais relevantes, responsáveis por um ganho de até $1,5$ no tempo de execução, foram efetuadas na indexação e na forma de acesso das estruturas de armazenamento dos cálculos prévios das coordenadas globais dos pontos de Gauss η_{ab} e dos produtos do jacobiano pelos pesos dos pontos de Gauss $|J|\, w_b w_a$ na montagem das colunas, conforme a Seção 2.3.2. Uma simples troca da ordem dos índices diminuiu o *loop stride* das integrações numéricas.

Porém outras técnicas manuais de otimização foram também empregadas, como, por exemplo, desenrolamento de laços (*loop unrolling*), redução de complexidade de operações (*strength reduction*), linearização (*inlining*). Chellappa, Franchetti e Puschel (2008) fornecem uma boa descrição dessas técnicas.

O resultado final do processo de otimização foi um ganho de até $2,42$ no tempo de processamento.

Essa segunda etapa resultou, portanto, em uma versão otimizada e paralelizável do código. Além disso, o custo de armazenamento foi reduzido para $\mathcal{O}(n)$, o que possibilita a resolução de problemas maiores.

3.4.3 Paralelização MPI

O terceiro e último passo no trabalho de Beer, Smith e Duenser (2008) [cap.8] é a paralelização do código através de MPI. Na versão dos autores não há hierarquia entre os processos e todos os processos realizam as mesmas tarefas, exceto pela escrita no arquivo de saída, que é sempre realizada pelo processo de menor número (*mpi rank*). Os elementos de contorno são divididos o mais igualmente possível entre os processos e durante as rotinas de montagem cada processo calcula somente as linhas que lhes compete. Antes e depois das rotinas de montagem, os nós realizam uma operação MPI_ALL_GATHER, não muito eficiente, que concatena os resultados em cada um dos processos.

O código de Beer, Smith e Duenser (2008) é simples e pode ser compreendido sem dificuldades por um programador acostumado com algoritmos sequenciais, como afirmam os autores. No entanto, não se ganha nada com a redundância das operações e a economia obtida no envio dos dados de entrada, da malha funcional, da diagonal e do lado direito da equação (todos com tamanho de ordem $\mathcal{O}(n)$) é pequena comparada com o gasto extra de se transmitir e acumular cada um dos resultados intermediários em todos os processos (MPI_ALL_GATHER). Além disso, as rotinas de entrada concorrem entre si pelos mesmos recursos e uma estratégia melhor seria fazer com que apenas uma rotina realizasse a leitura dos dados e distribuísse esses dados entre os demais processos.

A estratégia adotada neste trabalho foi a inversa. Em vez de se paralelizar indistintamente todas as rotinas, paralelizou-se individualmente cada rotina, iniciando-se pela rotina de montagem, visivelmente mais onerosa. Posteriormente, foram paralelizadas as demais rotinas: cálculo da malha funcional, montagem do lado direito da equação e montagem da diagonal. A estratégia de comunicação foi planejada e medida durante cada modificação do código. Além disso, somente um dos processos é responsável pela execução das partes sequenciais e pela distribuição de tarefas, o que permitiu a realização de ensaios sobre balanceamento de carga.

Desse modo, nesta etapa, são produzidas duas versões principais (P2.0 e P2.1) e duas versões secundárias (P2.0.1 e P2.0.2), conforme a lista abaixo:

- **P2.0**: Paralelização MPI do Cálculo das Colunas
  - **P2.0.1**: Balanceamento de Carga Simples
  - **P2.0.2**: Balanceamento de Carga Adaptativo
- **P2.1**: Paralelização MPI dos Cálculos do Vetor Resultado, da Diagonal e da Malha Funcional

3.4.3.1 P2.0: paralelização MPI do cálculo das colunas

Nesta versão do código é utilizado o modelo conhecido na literatura como "companheiros" (*peers*), uma variação do modelo "mestre-escravo", em que o "mestre", além de delegar tarefas aos "escravos", também executa parte das tarefas.

A Fig. 6 representa simplificadamente a implementação paralela da versão **P2.0**.

O processo principal executa sequencialmente as rotinas de leitura de dados, cálculo da malha funcional, cálculo da diagonal da matriz e do lado direito da equação. Os dados lidos e calculados são transmitidos aos outros processos através da operação $MPI\_BROADCAST$. Essa operação transmite dados a todos os processos utilizando uma estrutura em árvore, isto é, iniciando com o processo principal, cada processo transmite os dados somente a v processos vizinhos até que todos os nós recebam os dados. Durante a rotina de solução do sistema linear, o processo principal envia a todos os processos um dos vetores direção $\mathbf{u}$ ou resíduo $\mathbf{r}$ através da operação $MPI\_BROADCAST$. Em seguida, envia através da operação MPI_SEND

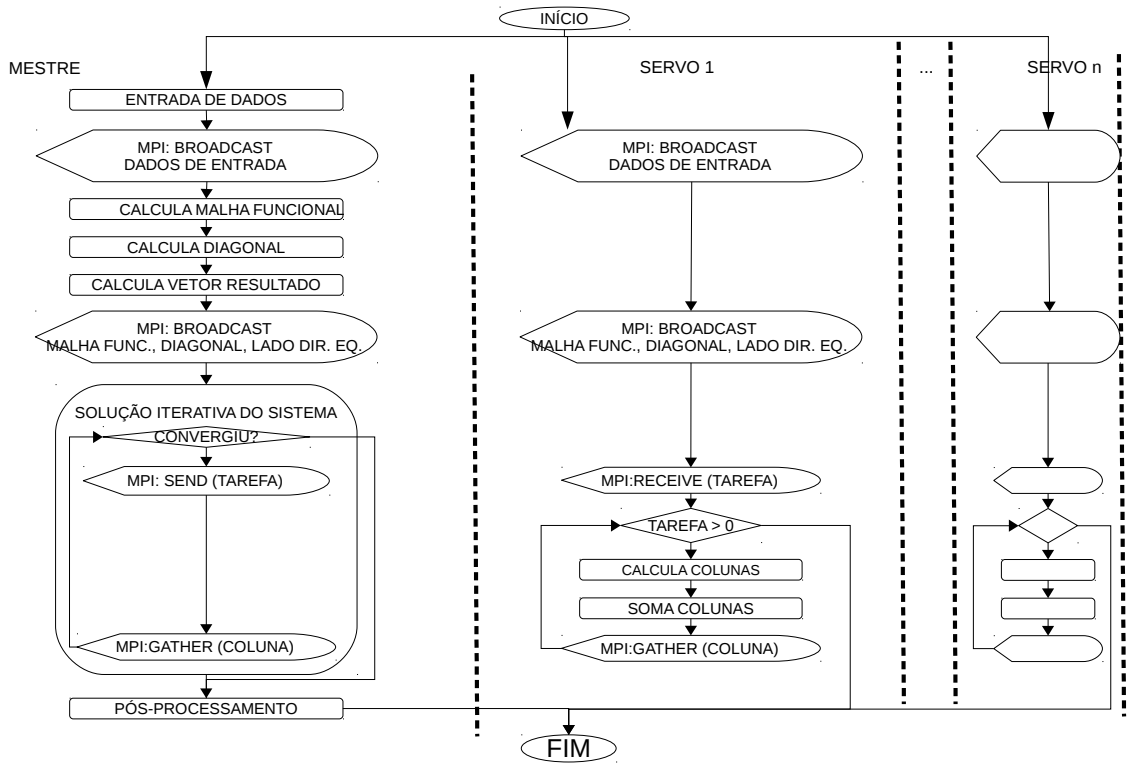


FIGURA 6 – P2.0: Paralelização do Cálculo das Colunas

um **pacote de tarefas** a cada processo, que consiste em dois inteiros, indicando o intervalo de colunas a ser calculado. Cada processo calcula cada coluna do seu pacote de tarefas, multiplica-as pelos elementos correspondentes do vetor direção ou resíduo e acumula a soma desses valores em um vetor, como nas Eqs. 42. Finalmente, esses vetores são acumulados em um único vetor no processo principal através da operação MPI_GATHER. A operação MPI_GATHER acumula os resultados parciais de cada um dos processos no processo principal, usando algoritmos otimizados de comunicação, o que é mais eficiente que transmitir todos resultados parciais a um dos nós e somá-los, pois isso sobrecarregaria os recursos de processamento, armazenamento e comunicação desse nó.

Note-se que a operação MPI_GATHER aqui utilizada é bem menos onerosa que a operação MPI_ALL_GATHER, utilizada por Beer, Smith e Duenser (2008), que distribui e acumula os resultados em cada um dos processos.

3.4.3.2 P2.0.1: balanceamento de carga simples

A versão secundária **P2.0.1** é derivada da versão **P2.0** e implementa uma estratégia simples de balanceamento de carga. A efetividade do balanceamento de carga é baseada em dois fatos. Primeiro, devido ao número variável de pontos de Gauss nas integrações numéricas, a montagem das colunas só é homogênea em um domínio esférico. Isso se comprova, mesmo para os experimentos com domínio em forma de cubo e com uma malha de elementos de contorno quadrados e de mesma dimensão. Segundo, em qualquer ambiente computacional, parte do processamento é sempre ocupado por processos do sistema operacional e outros eventuais processos concorrentes.

Nesses ensaios, considera-se esses dois fatos já no projeto do algoritmo.

A ideia do balanceamento de carga é medir, em tempo de execução, o tempo de processamento de cada lote de tarefas e ajustar as próximas divisões de tarefa de acordo com os tempos de execução informados. Assim, a primeira divisão de tarefas é feita de forma homogênea, mas as divisões seguintes levam em conta o tempo de processamento enviado por cada um dos processos concorrentes.

O Algoritmo 3.3 descreve simplificadamente essa ideia.

Algoritmo 3.3 Balanceamento de Carga Simples

```
1:  função BALANCEAMENTO_SIMPLES(LOTE, TEMPOS)
        \\ Cálculo da velocidade de processamento (colunas/ns)
2:      para i ← 1 até P faça
3:          VELOCIDADE(i) = LOTE(i)/TEMPOS(i)
4:          TEMPO_TOTAL= TEMPO_TOTAL + TEMPOS(i)
5:          VELOCIDADE_TOTAL= VELOCIDADE_TOTAL + VELOCIDADE(i)
6:      fim para
        \\ Cálculo da velocidade média
7:      VELOCIDADE_MEDIA = N/TEMPO_TOTAL
        \\ Cálculo do tempo ótimo
8:      TEMPO_OTIMO = N / VELOCIDADE_TOTAL
        \\ Nova distribuição de tarefas
9:      para i ← 1 até P faça
10:         NOVO_LOTE(i) = TEMPO_OTIMO * VELOCIDADE(i)
11:     fim para
12:     retorna NOVO_LOTE
13: fim função
```

O gráfico da Fig. 7 mostra um teste simples que comprova a efetividade do método.

O eixo horizontal indica cada uma das montagens paralelas e o eixo vertical indica o número de colunas atribuídas a cada processo. Na resolução do problema com 864 elementos com 12 processos paralelos, arbitrariamente fixou-se a distribuição de tarefas inicial com 100 colunas para os primeiros dois processos e 0 colunas para os dois últimos processos. Já na terceira montagem das colunas, verifica-se uma estabilização das cargas delegadas a cada processo paralelo.

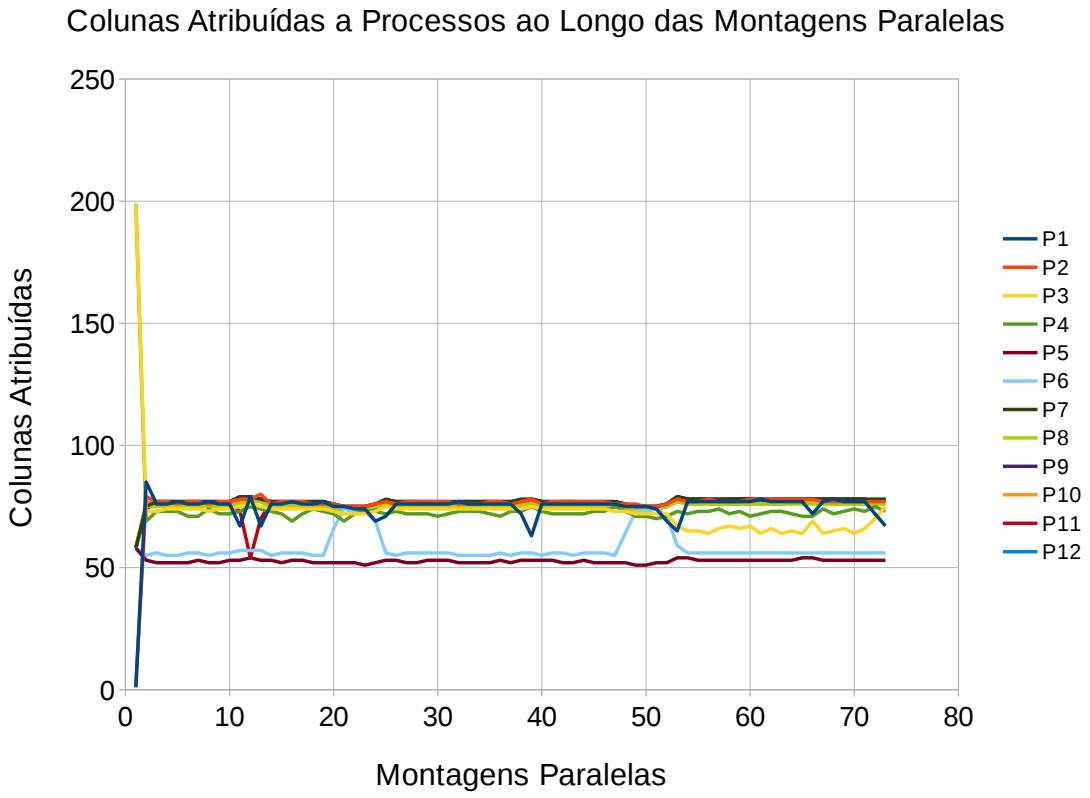


FIGURA 7 – P2.1: Distribuição de Tarefas 864 Elementos 12 processos: Cada curva representa o número de colunas atribuídas a cada processo ao longo das montagens paralelas durante a execução do programa.

3.4.3.3 P2.0.2: balanceamento de carga adaptativo

A versão **P2.0.2** é uma evolução da versão **P2.0.1**. Na versão anterior, a redistribuição de colunas era feita a cada distribuição de colunas. No entanto, nota-se que o código pode ser otimizado quando se define os conceitos de perturbação e estabilidade.

A cada medição de velocidade de processamento, armazena-se o maior desvio entre a

velocidade de processamento média e as velocidades de processamento dos processos. Dessa forma, os h últimos maiores desvios são sempre armazenados. Uma execução paralela é **estável** se o maior desvio for menor ou igual a cada um dos desvios do histórico, com uma tolerância de t. Uma **perturbação** é detectada quando o maior desvio da última medição ultrapassar algum dos desvios do histórico.

Enquanto as velocidades de processamento permanecerem estáveis, mantém-se a mesma distribuição de tarefas. Somente quando uma perturbação é detectada, a distribuição de tarefas é recalculada.

Essa versão forneceu um ganho de até 1.2 nos tempos de execução para o problema de 117.600 elementos.

3.4.3.4 P2.1: paralelização MPI dos cálculos do vetor resultado, da diagonal e da malha funcional

A primeira rotina paralelizada na versão **P2.1** foi a rotina de cálculo do lado direito da equação, que também possui complexidade $\mathcal{O}(n^2)$.

Uma vez que as rotinas com complexidade quadrática são paralelizadas, as rotinas de complexidade linear começam a ocupar um tempo de CPU visível. Em seguida foram paralelizados, portanto, esse segundo grupo de rotinas.

Conforme ilustra a Fig. 8, utilizou-se a mesma estratégia da versão **2.1** para distribuição de tarefas.

3.4.4 Armazenamento em memória disponível

A versão da etapa anterior **P2.1** é uma versão do código paralelizada, com eficiência paralela oscilando entre 83% e 115% e com custo de armazenamento linear.

De acordo com as medições obtidas, os problemas com 194.400 elementos de contorno, que produzem uma matriz que necessitaria 281 GB para serem inteiramente armazenadas, utilizam cerca de 15 MB da memória RAM disponível em cada processo. Com doze processos por nó do *cluster* de computadores, o programa utilizaria apenas cerca de 150 MB

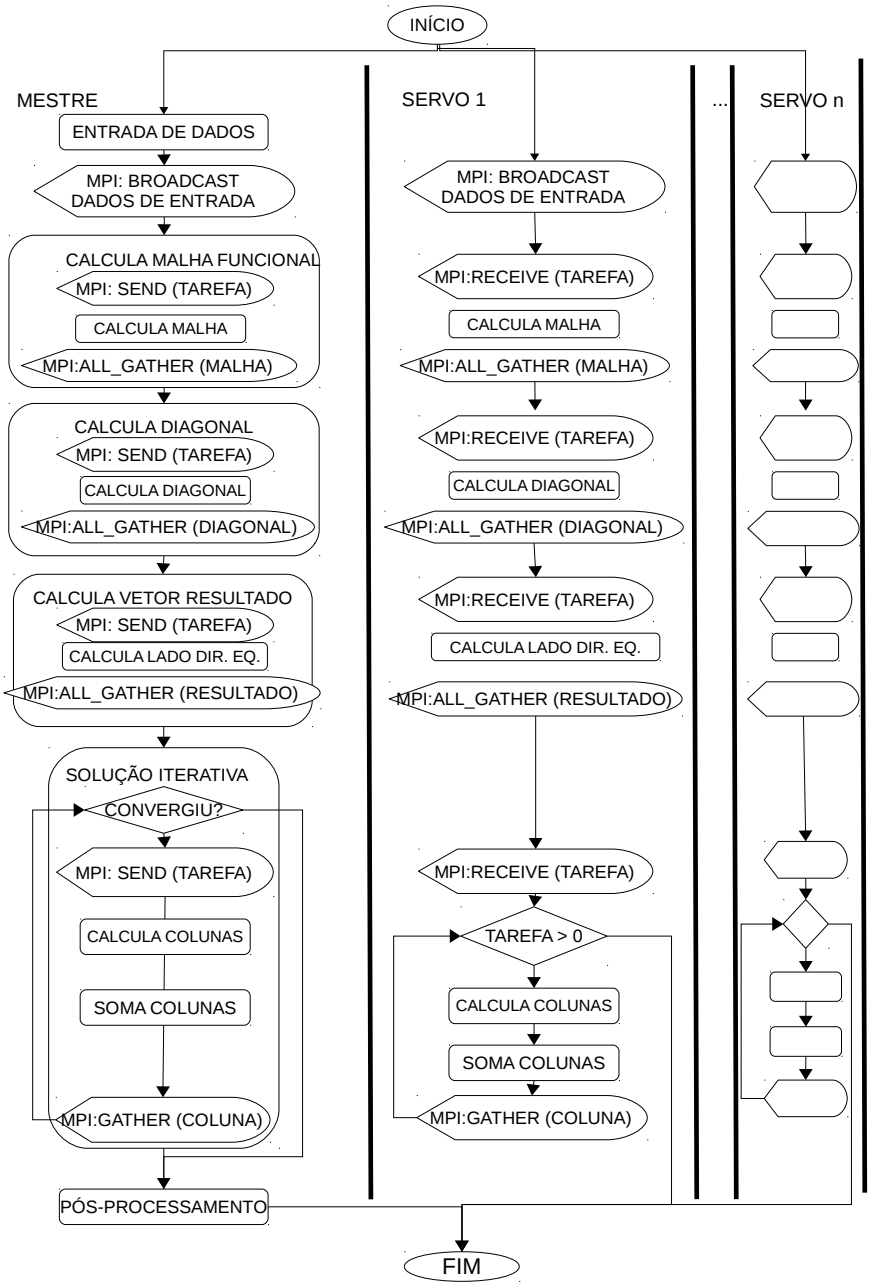


FIGURA 8 – P2.1: Paralelização MPI dos cálculos do lado direito da equação, da diagonal da matriz e da malha funcional

dos 48 GB de memória RAM de cada nó.

A questão desta quarta etapa de trabalho é como utilizar a memória disponível de forma a diminuir o tempo de execução.

Como exposto na Seção 2.3.2, as integrações numéricas utilizam 4, 16 ou 100 pontos de Gauss. A primeira ideia é armazenar os elementos da matriz cuja integração utiliza 100 e 16 pontos de Gauss e que portanto requerem 100 e 16 iterações do laço mais interno da

rotina de montagem da coluna. Por brevidade, esses elementos da matriz são denominados neste trabalho **G10x10** e **G4x4**. O armazenamento desses pontos foi implementado na versão **P3.0**.

Embora a primeira ideia tenha dado resultados positivos, ainda assim uma pequena parte da memória física disponível é utilizada pelo programa. Antes de implantar o armazenamento dos elementos G2x2, bem mais abundantes na matriz, foram realizados diversos testes em versão sequencial para esclarecer os custos de montagem e acesso à memória envolvidos. Esses testes permitiram experimentar mais alternativas, pois tanto a programação quanto as análises de desempenho são mais ágeis no código sequencial. Para ilustrar os caminhos possíveis, foram escolhidas três ideias que deram origem às versões **s3.1.0**, **s3.1.1** e **s3.1.2**.

Finalmente, os resultados dos testes sequenciais foram aplicados na versão paralela **P3.2**.

3.4.4.1 P3.0: armazenamento global

O processo de otimização da versão **s1.0a** mostrou a importância de uma estrutura de dados eficiente e de uma indexação simples para que o armazenamento de dados produza bons resultados. Sabe-se que a integração numérica requer mais pontos de Gauss quando os pontos fonte ξ estão próximos ao elemento de contorno sendo integrado. No entanto, como a numeração dos elementos de contorno é feita de forma aleatória pela ferramenta que gerou as malhas dos testes, não há nenhum padrão para se encontrar os elementos G10x10 de uma coluna. A questão central é como armazenar e indexar elementos distribuídos de forma esparsa e aleatória em uma matriz quadrada de forma que possam ser acessados centenas de vezes de forma eficiente. Essa questão possui certa semelhança com o armazenamento de matrizes esparsas e a obra de Saad (2003) foi uma boa referência para esse assunto.

Uma diferença entre o armazenamento de matrizes esparsas e de elementos G10x10 é que não há necessidade de se armazenar os índices dos elementos, pois o cálculo do número de pontos de Gauss já faz parte da montagem das colunas. Se a montagem da matriz fosse sequencial, o armazenamento dos índices seria completamente desnecessário, pois bastaria armazenar os elementos G10x10 sequencialmente e estes seriam sempre acessados na mesma

ordem. Transpondo essa ideia para o processamento paralelo, em que cada processo recebe um intervalo de colunas para serem montadas, um vetor de n posições indicando a posição do primeiro elemento G10x10 de cada coluna é suficiente.

A estrutura de dados final é bastante compacta e pode ser distribuída a todos os processos com um custo pequeno. Aliás, mesmo acrescentando o armazenamento dos elementos G4x4, o custo da comunicação e do acesso à estrutura permanece menor que o custo do cálculo dos elementos. Por simplicidade, essa área de armazenamento é aqui chamada de **armazenamento global**, embora em realidade exista uma cópia desses dados em cada processo, assim como há uma cópia dos dados de entrada, da malha funcional, da diagonal da matriz e do lado direito da equação.

Dessa forma, a estratégia de armazenamento dos pontos G10x10 pode ser descrita assim:

1. na primeira montagem, cada processo possui:
   - um **vetor real local** para armazenar os valores dos elementos G10x10
   - um **vetor inteiro local** para armazenar as posições no vetor real dos primeiros elementos G10x10 de cada coluna
2. ao final da primeira montagem, utiliza-se a operação MPI_ALL_GATHER para concatenar os vetores locais em um **vetor real global** e um **vetor inteiro global**, de forma que cada processo possui uma cópia de todos os valores dos elementos G10x10 e os índices dos primeiros elementos de cada coluna.
3. nas montagens subsequentes, existe um **contador de pontos G10x10** que é zerado a cada montagem da coluna. A soma do índice do primeiro elemento G10x10 da coluna e do contador fornece o índice para acessar o valor do elemento armazenado.

Com o armazenamento dos elementos G10x10 e G4x4, os resultados experimentais desta versão mostraram diminuições de até 40% no tempo de processamento dos problemas resolvidos.

3.4.4.2 s3.1.0, s3.1.1, s3.1.2: testes sequenciais para armazenamento local

O objetivo dos testes sequenciais é, partindo da versão **s1.0a**, montagem elemento-por-elemento otimizada, realizar alterações no código que permitam a utilização da memória disponível para diminuir o tempo de execução do programa.

O primeiro problema encontrado é determinar a quantidade de memória disponível. Os programas enxergam a memória do sistema como um espaço de memória virtual contínuo e limitado apenas pela capacidade de endereçamento do sistema. Isso não significa que não ocorram erros de alocação por falta de memória em tempo de execução, mas esses erros dependem de fatores que estão fora da área de atuação do programador como: o modo como a área de dados do programa é organizada pelo compilador, o modo como o sistema operacional aloca recursos aos processos, configurações de performance do sistema operacional definidas pelo administrador do sistema, etc. Portanto esse problema foi resolvido de forma empírica. Considerou-se como a quantidade de **memória disponível em cada nó do cluster** a memória utilizada (*Maximum Resident Memory Set Size*) pelo maior problema executado sem paginação no disco rígido. Esse valor corresponde a cerca de 39.27 GB e é suficiente para armazenar uma matriz quadrada de 72.600^2 elementos. Esse valor é passado como um parâmetro de entrada ao programa.

3.4.4.3 teste sequencial 1 (s3.1.0): colunas copiadas, em vez de calculadas

Uma vez estabelecido o espaço disponível, o segundo passo é avaliar a estrutura e a forma de indexação. Revendo os resultados anteriores, procura-se uma solução que:

1. matenha a simplicidade do algoritmo de montagem: deve ser evitada a inserção de uma instrução condicional para verificar se cada elemento da coluna já está armazenado ou precisa ser recalculado.

2. seja indexada por apenas um inteiro, acessado por laços com *stride* unitário.

A versão **s3.1.0** foi baseada nesses dois pontos e as seguintes alterações no código foram efetuadas:

- armazena-se colunas inteiras, de modo que cada coluna tenha dois estados: armazenada ou calculada.
- na rotina de entrada de dados, uma vez que o número de colunas da matriz é determinado, divide-se a área de armazenamento pelo tamanho da coluna, obtendo-se o **número de colunas armazenadas** (C).
- o estado das colunas (armazenada ou calculada) não precisa ser armazenado, pois pode ser verificado com apenas uma comparação ($j \leq C?$).
- na rotina de montagem, se a coluna está armazenada, copia-se a coluna da área de armazenamento.

Essas alterações foram efetivas, com ganhos de até 45 em relação aos tempos da versão **s1.0a**, e as ferramentas indicaram que o gargalho do programa migrou da rotina de montagem da coluna para a acumulação dos produtos das colunas pelos escalares $S = S + A(:, j) * u(j)$ ou $S = S + A(:, j) * r(j)$.

3.4.4.4 teste sequencial 2 (s3.1.1): BLAS

Uma vez que o gargalo do programa passou a ser a acumulação dos produtos das colunas pelos vetores de busca, essa versão utiliza a rotina AXPY da biblioteca BLAS na versão **s3.1.1**, que implementa essa operação em diversas linhas de execução com OMP. De forma surpreendente, não houve ganho notável com essa mudança. Do contrário, em 80% dos testes realizados, essa versão mostrou desempenho pior que a versão **s3.1.0** anterior. As ferramentas mostraram que é mais eficiente utilizar toda a hierarquia de memória para um único processo executando a acumulação vetorial do que dividi-la entre linhas de execução paralelas executando essas operações.

3.4.4.5 teste sequencial 3 (s3.1.2): colunas armazenadas em memória

As análises da versão anterior mostraram o quanto a estratégia de armazenamento estava sobrecarregando a hierarquia de memória. Isso deu origem à última versão serial **s3.1.2**, que parte da versão **3.1.0** com as seguintes alterações:

- em vez de realizar o teste $j \leq C$ na rotina de montagem da coluna, esse teste é efetuado na rotina de solução do sistema, que decide se a rotina de montagem da coluna precisa ou não ser chamada. Dessa forma, a rotina de montagem volta à simplicidade da versão **s1.0a** .

- os dados não são copiados da área de armazenamento para um vetor antes da multiplicação, mas a instrução de multiplicação tem como operando a própria área de armazenamento com os índices apropriados.

O segundo item utiliza recursos do Fortran 2005 que permitem a multiplicação de um escalar por um vetor em uma única instrução e referência a partes de matriz ou vetor através de intervalos de indexação. Por exemplo, supondo que todas as colunas estão armazenadas sequencialmente em um grande vetor chamado $AREA$, o índice do primeiro elemento da coluna j é $IC = 1 + (j-1) * n$ e a coluna j inteira pode ser referenciada por $AREA(IC : IC + (n-1))$. Dessa forma, sem a utilização de recursos como ponteiros da linguagem C, é possível utilizar os valores armazenados em uma área da memória sem copiá-los.

A versão **3.1.2** produziu ganhos de desempenho de até 1.8 vezes com relação à versão **3.1.0** e de até 73 vezes, com relação à versão **s1.0a** (montagem elemento-por-elemento otimizada).

3.4.4.6 P3.2: armazenamento local

A versão **P3.2**, utilizando as ideias dos testes sequenciais, apresenta as seguintes alterações à versão **P2.1**:

- cada processo possui sua própria área de dados, cujo tamanho é determinado por um parâmetro de entrada do programa. Calculou-se esse parâmetro empiricamente, utilizando $1/12$ da memória consumida pelo maior problema resolvido pela versão sequencial **s0.2** sem paginação em disco.

- na rotina de entrada de dados, o **número de colunas armazenadas por processo** (C) é obtido pela divisão do tamanho da área de armazenamento de um processo pelo tamanho da coluna.

- a rotina que recebe o intervalo de colunas a serem montadas determina quais colunas estão armazenadas e quais colunas devem ser calculadas.

- os dados armazenados não são copiados, mas a instrução de multiplicação tem como operando a própria área de armazenamento com os índices apropriados.

- as demais rotinas, incluindo as rotinas de montagem das colunas, não foram alteradas.

Os resultados experimentais da versão **3.2** foram surpreendentes:

- a execução do programa em 10 nós e 12 processos por nó (n10ppn12) obteve ganhos de até 10.47 vezes relativos à versão **P2.2**, valor bem menor que o ganho de 73 entre as versões sequenciais.

- ainda com n10ppn12, a configuração do tamanho da área de armazenamento necessita ser redimensionada a partir de 117.600 elementos para que a execução do programa deixe de fazer paginação da memória em disco.

- no entanto, a execução com apenas 2 processos por nó (n10ppn2) produziu ganhos semelhantes, em alguns casos maiores, à execução com 12 processos. Além disso, a configuração do tamanho da área de armazenamento foi mantida para a execução de todos os problemas, sem paginação da memória.

- como era de se esperar, a execução com 1 processo por nó (n10ppn1) resultou em tempos dobrados com relação à n10ppn2.

- pela primeira vez a comunicação entre os processos passou a apresentar um custo significativo e os resultados obtidos com a rede *infiniband* foram notavelmente superiores (até 40%) aos resultados obtidos com a rede *ethernet*.

Portanto, para o tamanho de área de armazenamento adotado, essa versão é apropriada para a execução em 2 processos por nó. Isso faz sentido quando se tem em mente a arquitetura da hierarquia de memória da Fig.2, que possui acesso à memória não-uniforme (NUMA) e L3 dedicado a cada multiprocessador.

Como bem advertem Kennedy e Allen (2001), deve-se desconfiar dos ganhos grandes, pois são obtidos somente quando os recursos não estavam sendo bem utilizados. À medida que os recursos passam a ser melhor utilizados, os ganhos passam a ser menores e dependentes de fatores mais complexos.

Finalmente, embora este trabalho não tenha implementado o algoritmo de Beer, Smith e Duenser (2008) e isso não tenha sido demonstrado diretamente, considera-se provado indiretamente que o armazenamento das matrizes H e G dificilmente traria ganhos significativos ao desempenho da rotina de montagem elemento-por-elemento da matriz do sistema paralelizada com MPI para problemas com graus de liberdade entre 100.000 e 200.000 e L3 com 32 MB. Basta comparar o trabalho realizado, por um lado, pelo acesso do valor armazenado em memória e sem cópia para variáveis intermediárias como o da versão **P3.2**, com o trabalho de decidir entre quais das matrizes acessar, acessar esse valor em uma matriz de dimensão $n \times n$ com *loop stride* de n (devido à montagem por linha) e decidir entre multiplicar esse valor por -1 ou não.

3.4.5 Paralelização híbrida

Finalmente, nesta última etapa de trabalho, procura-se utilizar os outros 10 processadores que ficam ociosos com execuções MPI n10ppn2. Isso foi realizado adicionando-se diretivas OMP, que paralelizam os processos por meio da criação de **linhas de execução** (*threads*).

As linhas de execução de um mesmo processo, também conhecidas como "processos leves", compartilham o mesmo espaço de memória do processo mas cada linha possui um conjunto independente de estados dos registradores, incluindo o estado do ponteiro de instrução. O sistema operacional aloca tempo de processamento a cada linha de execução como se fosse um processo e linhas de execução de um mesmo processo podem ser designadas a processadores diferentes.

A versão **h4.0** implementa paralelismo híbrido MPI e OMP. O trabalho computacional de cada processo é dividido entre seis linhas de execução, cada uma sendo executada em um

núcleo de processamento diferente. Desse modo, todos os doze núcleos de processamento físicos de cada um dos dez nós do cluster são utilizados.

Embora o modelo de programação paralela OMP também coloque à disposição do programador a possibilidade de criação de tarefas concorrentes, nesta fase somente o recurso de paralelização de laços (*loops*) foi utilizado. Através de diretivas no código, o compilador é instruído a dividir um conjunto de iterações em L linhas de execução. Há diversos algoritmos para essa divisão de tarefas. Por exemplo, supondo um laço com I iterações, a **atribuição estática** instrui que cada linha de execução execute I/L iterações consecutivas e a **atribuição em tempo de execução** (*runtime schedule*) instrui que as linhas de execução processem pequenos lotes de P iterações por vez, retirando mais pequenos lotes de um "jarro de tarefas" conforme os lotes forem sendo completados. Além disso, como as variáveis do processo são a princípio compartilhadas por todas as linhas de execução, há necessidade de se especificar quais variáveis são copiadas para variáveis locais, acessíveis a uma única linha de execução.

O custo adicional imposto pela paralelização OMP é composto pela criação e destruição das linhas de execução, alocação de memória e atribuição de valores às variáveis locais, distribuição de tarefas e, finalmente, síntese dos resultados ao final das tarefas paralelas. Esse custo adicional é geralmente diluído para laços com muitas iterações e poucas variáveis locais.

Foram manualmente testadas as paralelizações de diversas partes do código, com diversas configurações. Os melhores resultados foram obtidos com a paralelização dos laços:

a) da acumulação dos resultados da multiplicação das colunas da matriz do sistema pelos vetores direção e resíduo $S = S + A(:, j) * u(j)$ e $S = S + A(:, j) * r(j)$,

b) da rotina de cálculo do lado direito da equação do sistema linear (que possui complexidade $\mathcal{O}(n^2)$) e

c) da rotina de montagem das colunas (que é chamada $8in$ vezes).

A paralelização dos laços dos cálculos da malha funcional e da diagonal não produziram bons resultados.

A versão **h4.0** n10ppn2 com 6 linhas de execução por processo resultou em ganhos de até 2.15, relativos à versão **P3.2** n10ppn2.

4 RESULTADOS EXPERIMENTAIS

Este capítulo possui sete seções, sendo as primeiras cinco correspondentes às cinco etapas do trabalho. A sexta seção compara os tempos de execução obtidos com a utilização das redes *infiniband* e *ethernet* e a sétima seção contém discussões gerais sobre os resultados.

Para facilitar a comparação entre os resultados, os primeiros gráficos das primeiras cinco seções são apresentados na mesma escala. Nesses cinco gráficos, os tempos de execução estão em escala logarítmica e os número de elementos de contorno, que aqui correspondem à dimensão n do sistema linear, estão em escala linear.

Cada medida de tempo de execução foi efetuada dez vezes, com um limite de tempo de execução de 30 horas.

4.1 ALTERAÇÕES NA SOLUÇÃO DO SISTEMA LINEAR

O gráfico 9 apresenta os tempos de execução para as três versões de código obtidas na fase de alterações da rotina de solução do sistema linear. Nesta fase, os testes ainda estão confinados a um único nó do cluster, pois nenhuma das versões exibe processamento paralelo com MPI. A versão **s0.0** (solução original) exibe processamento completamente sequencial, a versão **p0.1** (biblioteca numérica LAPACK) apresenta processamento paralelo em doze ou em vinte e quatro linhas de execução somente na rotina de solução do sistema linear, e, finalmente, na versão **s0.2** (solução iterativa BiCGStab) retorna-se ao processamento sequencial, porém sem aumento significativo no tempo de execução.

O maior problema executado pela versão **s0.0** no limite de tempo especificado possui 24.576 elementos de contorno (24K elem.), com tempo de execução de 28:30:00 (28 horas e 30 minutos) e desvio padrão (D.P.) de 00:07:00 (7 minutos). O menor problema rodado para

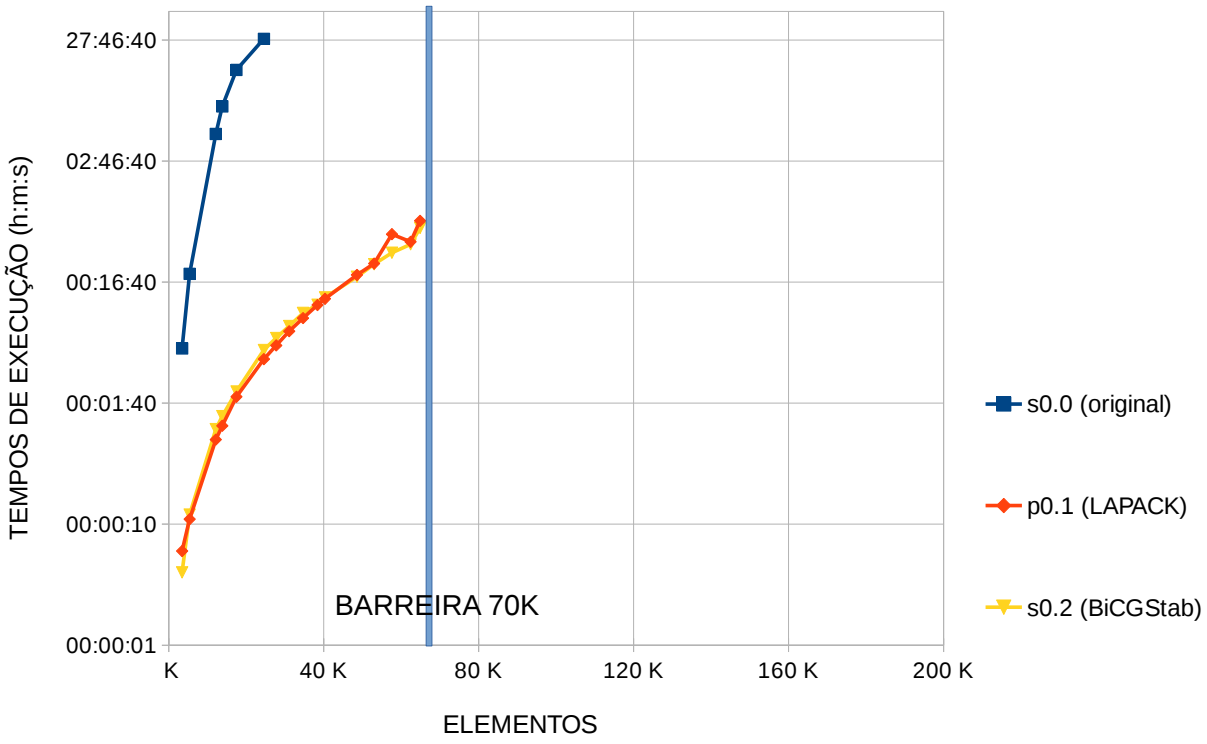


FIGURA 9 – Etapa 1 (s0.0, p0.1 e s0.2): solução do sistema linear

essa versão possui 3.456 elementos (3K elem.), com tempo de execução de 00:04:43 (D.P. 00:00:10). Os *Maximum Residente Set Size* (MRSS) registrados para esses problemas são, respectivamente, 4.5 GB e 23 MB. Note-se que o maior problema executado nessa versão utiliza apenas cerca de 10% da memória física de um nó de trabalho.

Os mesmos problemas rodados na versão **p0.1** demandam tempos de execução de 00:03:52 (D.P. 0:00:01) e 00:00:05 (D.P. 00:00:0.3). Esses números representam ganhos de velocidade de 442 vezes para 24K elem. e 50 vezes para 3K elem.

A versão **s0.2**, que utiliza a rotina de solução iterativa, apresenta tempos de execução bastante semelhantes à versão **p0.1** para esse conjunto de problemas, levemente superiores para problemas abaixo de 40K elementos e levemente inferiores para problemas acima de 40K, conforme se pode verificar no gráfico 9. Além disso, ao contrário da versão **p0.1**, a versão **s0.2** não apresenta nenhum tipo de paralelismo explícito e todo processamento é efetuado em apenas um dos doze núcleos de processamento do nó. Esses resultados confirmam as

conclusões de, por exemplo, (BARRA *et al.*, 1992; VALENTE; PINA, 2006; XIAO; CHEN, 2007), que demonstram a eficiência dos métodos iterativos da família Krylov para sistemas lineares decorrentes do BEM.

Enquanto na versão **s0.0** o recurso que limita o tamanho dos problemas é o tempo de processamento, nas versões **p0.1** e **s0.2** esse recurso é a memória RAM disponível. O maior problema executado pelas versões **p0.1** e **s0.2** possui 72K elementos e demanda 00:53:05 (D.P. 00:00:12) e 00:47:09 (D.P. 00:00:14). Um cálculo simples mostra que a matriz de $(72K)^2$ elementos reais com precisão dupla requer armazenamento de 39,27 GB e, de fato, a memória máxima utilizada para esse problema é de 39,36 GB para a versão **p0.1** e 39,31 GB para a versão **s0.2**.

Além do limite de memória física:

Problemas que demandam espaços de armazenamento maiores que a memória física disponível utilizam paginação da memória em disco para suprir esse déficit. Como o tempo de resposta do disco é muito maior que o tempo de resposta da memória RAM o tempo de execução desses problemas são obviamente muito maiores que o tempo obtido para problemas com requisitos de memória completamente atendidos pela memória física do nó de trabalho. É interessante notar que a dispersão das medidas também sofre um aumento significativo. Como ilustração desse fato para as versões **p0.1** e **s0.2**, enquanto problemas até 72K demandam menos de uma hora de processamento, com desvio padrão de alguns segundos em dez medidas, quatro execuções do problema de 86K demandaram de 7 a 20 vezes esse tempo, conforme ilustra a Tab.4.

TABELA 4 – Tempos de Execução além da barreira da memória física (p0.1 86K)

Medida	Tempo de Execução (hh:mm:ss)	Desvio (hh:mm:ss)
1	10:09:55	02:54:33
2	20:29:14	-07:24:46
3	07:53:17	05:11:11
4	13:45:25	-00:40:57
	Média	D.P.
Resultado	13:04:28	09:31:39

4.2 MONTAGEM DA MATRIZ ELEMENTO-POR-ELEMENTO

Esta fase do trabalho permite que o programa resolva, sem paginação da memória em disco, problemas com mais de 72K elementos e contém duas versões de código, ambas derivadas da versão **s0.2**: a primeira versão com montagem elemento-por-elemento (**s1.0**) e uma otimização dessa versão (**s1.0a**).

As curvas correspondentes a essas versões são adicionadas ao gráfico de tempos de execução na Fig. 10. Nota-se que, apesar dos elevados custos de processamento, a versão **s0.1a** ultrapassa a **barreira de 70K elementos** com tempos de execução de 27:45:40 (D.P. 00:07:35) para o problema com 78K elementos de contorno. A pequena dispersão da medida indica que não há paginação da memória em disco.

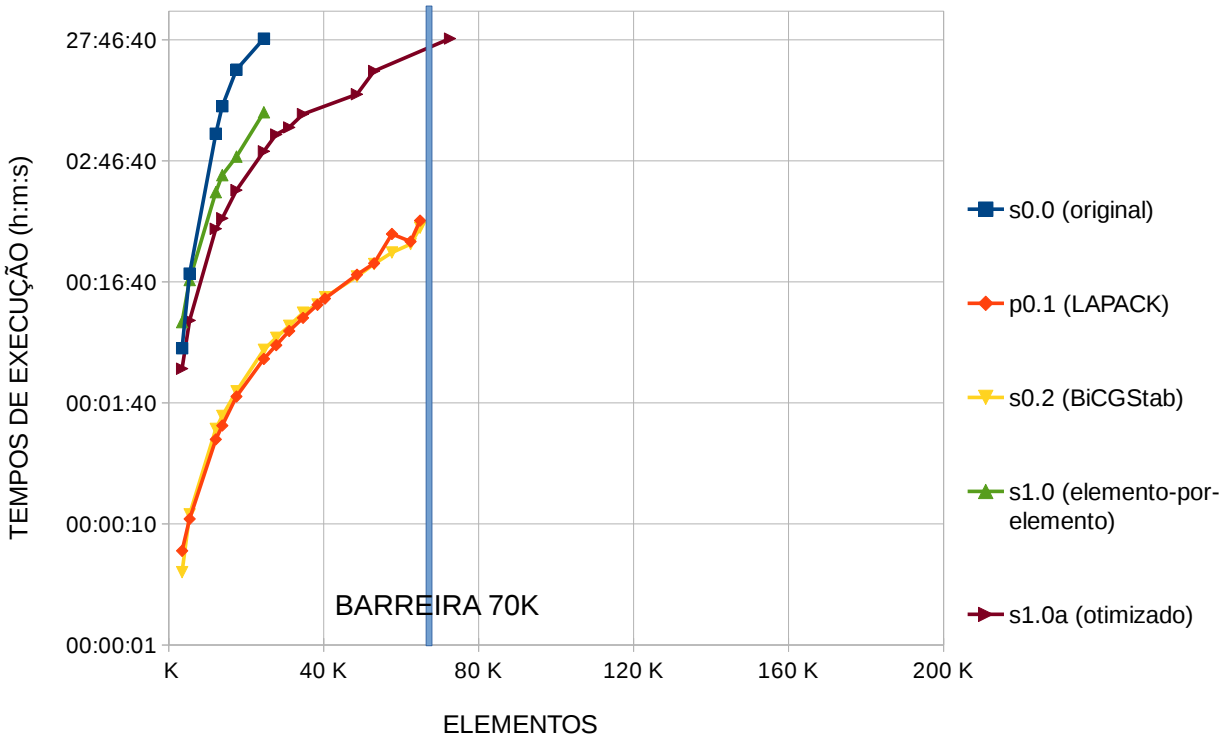


FIGURA 10 – Etapa 2 (s1.0, s1.0a): montagem elemento-por-elemento

De fato, observa-se que a memória utilizada pela versão elemento-por-elemento (*ma-*

ximum resident set size) é cerca de 10 vezes o tamanho teórico de uma coluna da matriz ($8n$ bytes, utilizando variáveis reais com precisão dupla), enquanto para versões da etapa anterior esses valores são, obviamente, bem mais próximos do tamanho teórico da matriz inteira ($8n^2$ bytes). A Tab. 5 compara os tamanhos da matriz e de uma coluna da matriz na memória com as memória utilizadas pelas versões **s1.0a** e **s0.2**.

TABELA 5 – Etapa 2: Tamanhos de matriz e coluna na memória e Memória Residente Máxima das versões s0.1 e s1.1a

	Valores Calculados		*Max. Resident Set Size*	
	Matriz	Coluna	s0.2	s1.0a
Elementos (n)	(GB)	(MB)	(GB)	(MB)
34656	8.95	0.26	8.97	2.69
38400	10.99	0.29	11.01	2.95
40344	12.13	0.31	12.15	3.08
77976	45.30	0.59	*****	5.91

O preço pago pelo baixo consumo de memória é o processamento da rotina de montagem, chamada oito vezes a cada iteração da solução do sistema. A Tab. 6 compara o número de montagens da matriz realizadas pelas versões **s1.0** e **s1.0a** com o **aumento** dos tempos de execução em relação à versão **s0.2**.

TABELA 6 – Etapa 2: Montagens da matriz e aumentos no tempo de execução.

elementos	iterações	montagens	$\mathbf{ganho^{-1}}$	
	(i)	$(8i)$	$\left(\frac{s1.0}{s0.2}\right)$	$\left(\frac{s1.0a}{s0.2}\right)$
5400	15	120	83.5	38.7
12150	19	152	91.5	45.6
13824	21	168	96.8	42.7
17496	19	168	88.3	46.7
24576	23	184	92.8	44.2

4.3 PARALELIZAÇÃO MPI

Esta etapa adiciona duas curvas ao gráfico dos tempos de execução: **P2.0** paralelização da rotina de montagem e **P2.1**, que inclui a paralelização das rotinas de cálculos da

malha funcional, diagonal da matriz e lado direito da equação do sistema linear, ambas as versões executadas em 10 nós do cluster e 12 processos por nó (Fig.11).

Novamente, destaca-se a possibilidade de análise de problemas com dimensões superiores à barreira de 70K, porém, com tempos de processamento mais razoáveis que os obtidos com as versões da etapa anterior.

O maior problema executado possui 194 K elementos com tempos de execução 3:57:29 (D.P. 0:00:25) para **P2.0** e 2:41:37 (D.P. 0:00:18) para **P2.1**.

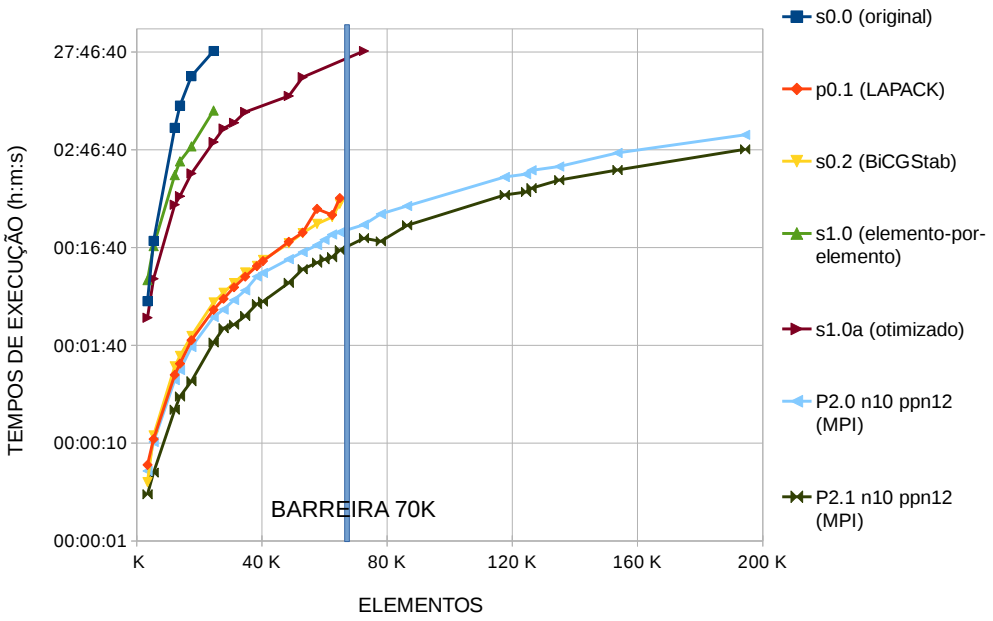


FIGURA 11 – Etapa 3 (P2.0, P2.1): paralelização MPI

Embora a comparação definitiva de ganho e eficiência paralela deva ser contra a melhor versão sequencial (**s0.2**), apenas com o objetivo de avaliar a paralelização e a otimização desta etapa, foram realizados os cálculos do ganho e da eficiência paralela em relação à versão **s1.0a**, que é a versão sequencial que foi paralelizada nesta etapa. Esses resultados intermediários são mostrados na Tab.7. Resultados finais são apresentados na Sec. 4.7.2.

TABELA 7 – Etapa 3: Resultados Intermediários: Ganho e Eficiência Paralela relativos a s1.0a

ELEMENTOS	P2.0		P2.1	
	ganho	E.P.	ganho	E.P.
17496	59,58	50,00%	138,24	115,20%
24576	61,58	51,32%	116,64	97,20%
27744	71,15	59,30%	126,15	105,12%
31104	64,98	54,15%	99,77	83,14%
34656	66,57	55,47%	109,42	91,18%
53016	61,35	51,13%	105,74	88,12%
72600	82,92	69,10%	126,22	105,18%

4.4 ARMAZENAMENTO EM MEMÓRIA DISPONÍVEL

Esta seção adiciona duas curvas ao gráfico de tempos de execução, ambos referentes à versão **P3.2** executada em 10 nós do cluster com 12 e 2 processos por nó. Nota-se claramente o melhor desempenho apresentado pela segunda curva (n10ppn2).

A curva para **P3.2** n10ppn12 oscila (alguns problemas ligeiramente maiores apresentam tempos de execução menores que os problemas vizinhos) e termina abruptamente no problema com 86 K elementos. Para problemas maiores, a execução do programa com o mesmo parâmetro de configuração para o tamanho da área de armazenamento das colunas passa a fazer paginação em disco. Conforme já mencionado na Seção 3.4.4.6, para continuar a resolver problemas maiores, esse parâmetro requer recalibração. A diminuição de memória disponível é devida, principalmente, aos *buffers* de memória utilizados pela rede *infiniband*, cujo tamanho aumenta proporcionalmente ao tamanho dos pacotes de dados que trafegam entre os processos. Esse fato foi confirmado com a análise do mapeamento de memória virtual dos processos fornecido pelo sistema operacional e é um dos principais motivos que torna a execução com ppn2 mais eficiente, já que dessa forma a comunicação entre processos de um mesmo nó do *cluster* é bastante reduzida. Outro motivo para a eficiência da execução com ppn2 parece ser a configuração da hierarquia de memória: cada um dos dois processos MPI é fixado em um *chip* diferente e usufrui dos 3 níveis de *cache* de memória.

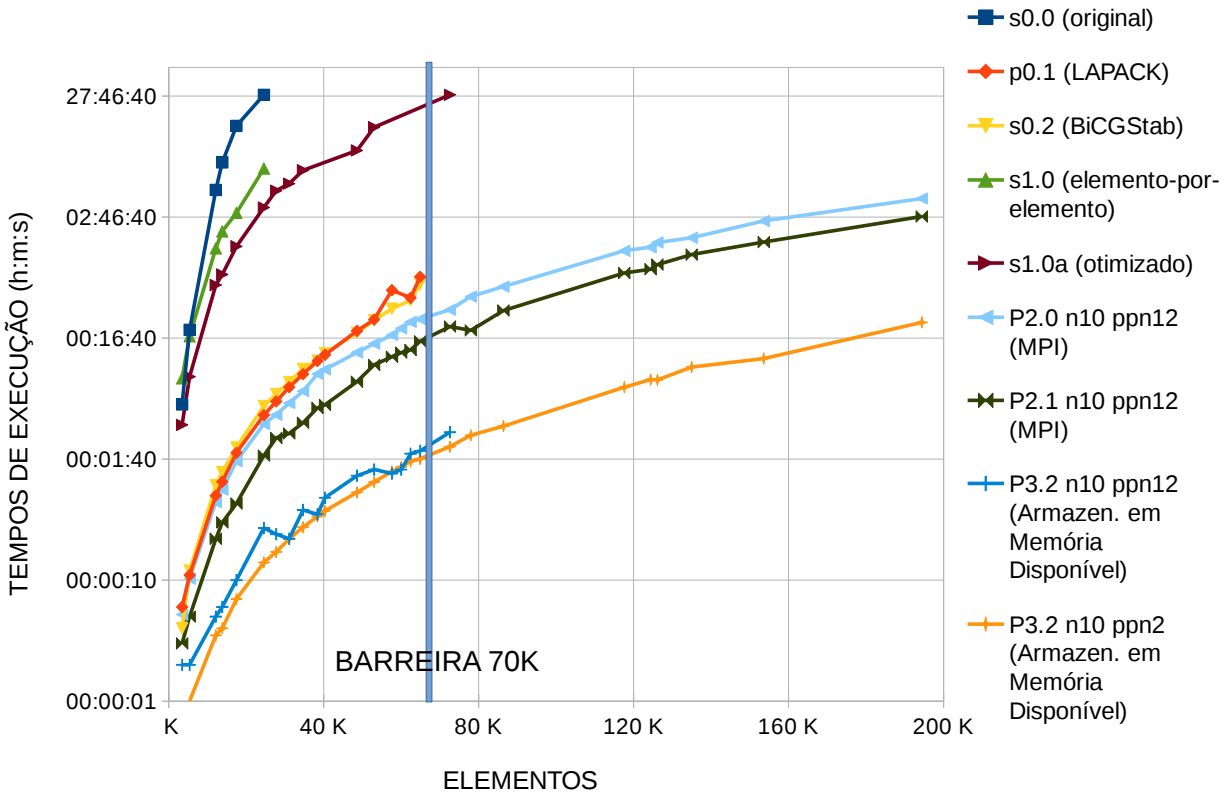


FIGURA 12 – Etapa 4 (P3.2): Armazenamento em Memória Disponível

4.5 PARALELIZAÇÃO HÍBRIDA

Esta seção adiciona a última curva ao gráfico de tempos de execução, referente à versão **h4.0** executada em 10 nós do cluster com 2 processos por nó e 6 linhas de execução por processo, de modo que os 12 núcleos de processamento de cada um dos 10 nós são utilizados.

Nota-se o bom desempenho oferecido pela versão **h4.0** em relação a todas as demais versões.

A fig. 14 apresenta em maior detalhe os tempos de execução das duas últimas versões do código **P3.2** n10ppn2 e **h4.0** n10ppn2 6 linhas de execução por processo. A resolução e a escala linear do tempo de execução desse gráfico permitem a visualização das barras de desvio. Nota-se menor dispersão dos resultados no processo híbrido.

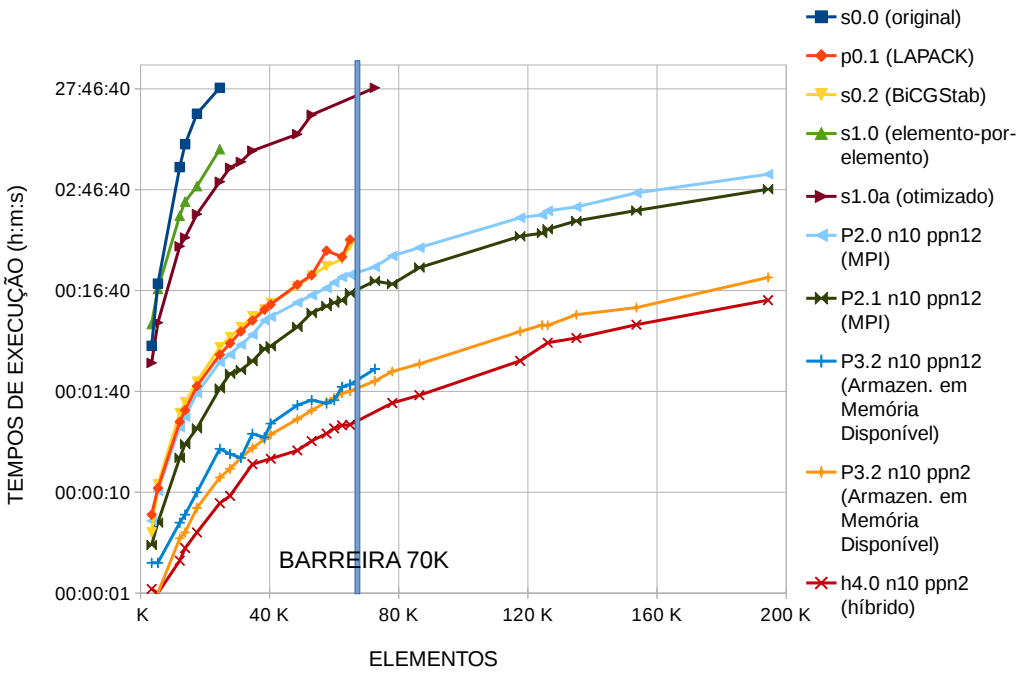


FIGURA 13 – Etapa 5 (h4.0): Paralelização Híbrida

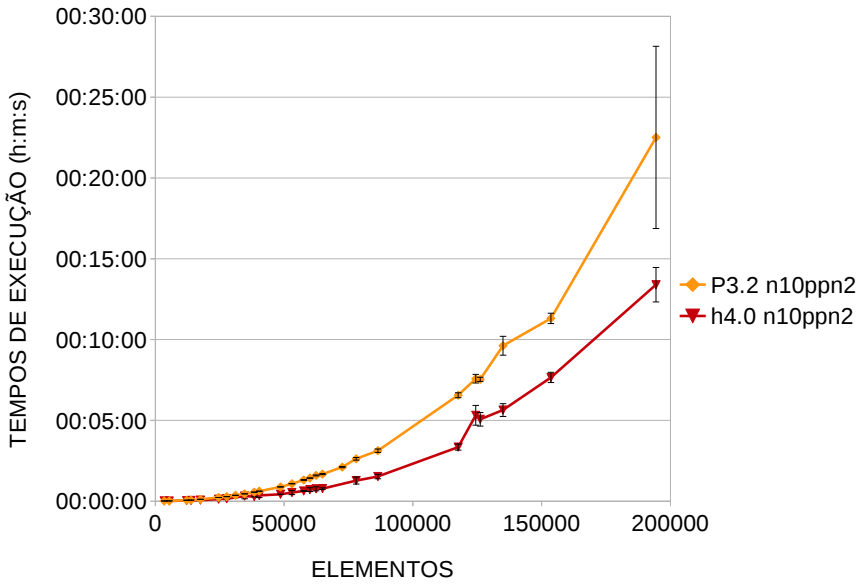


FIGURA 14 – Comparação dos Tempos de Execução: P3.2 e h4.0

4.6 COMPARAÇÃO *infiniband* e *ethernet*

Embora ofereça maior desempenho que a rede *ethernet*, o benefício trazido pela rede *infiniband* é praticamente imperceptível nas primeiras paralelizações MPI, que correspondem

às versões **P2.0** e **P2.1**. No entanto, esse benefício fica evidente nas versões seguintes, conforme o tempo de processamento é reduzido.

Nesta seção são apresentados dois experimentos comparando os tempos de execução obtidos com as redes *infiniband* e *ethernet*.

O primeiro gráfico (Fig. 15) apresenta os tempos da versão paralelizada com MPI **P2.1** e não oferece diferenças significativas. De fato, as ferramentas de análise de desempenho indicam que a soma das porcentagens de tempo de CPU utilizadas por todas as rotinas MPI não ultrapassava 10%.

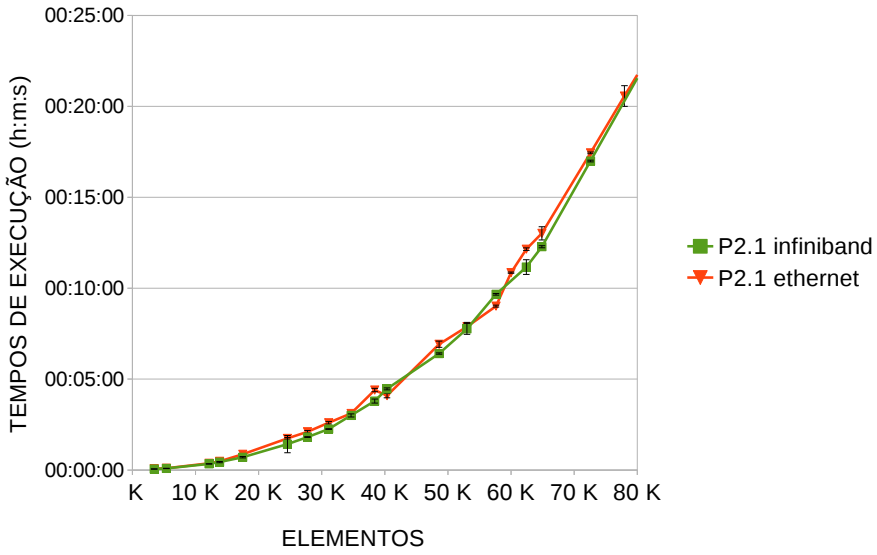


FIGURA 15 – Comparação dos Tempos de Execução: P2.1 *infiniband* e *ethernet*

Por outro lado, na versão híbrida **h4.0**, que apresenta tempos de processamento cerca de dez vezes menores que a versão **P2.1**, as diferenças dos tempos obtidos com as duas redes é relevante, conforme se verifica no gráfico da Fig. 16). As ferramentas indicam que as rotinas MPI utilizam cerca de 40% do tempo de CPU.

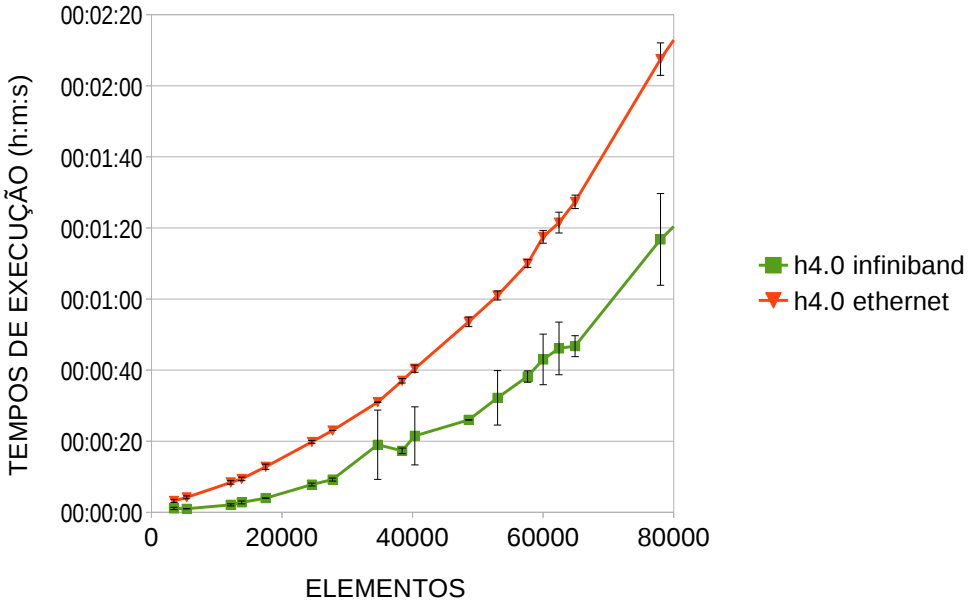


FIGURA 16 – Comparação dos Tempos de Execução: h4.0 *infiniband* e *ethernet*

4.7 DISCUSSÃO GERAL

A Tab.8 apresenta os ganhos máximos obtidos nas versões principais de cada etapa. Estão ausentes as versões de testes intermediários e são considerados os tempos provenientes das melhores configurações: utilização da rede *infiniband*, melhor configuração de nós e processos por nós (**n ppn**) do MPI e maior área de armazenamento. Os ganhors relativos são tomados com relação à versão da linha anterior da tabela e os ganhos absolutos são tomados em relação à versão **s0.0**.

As próximas subseções avaliam os resultados obtidos em dois aspectos: comparação entre as versões paralelas e desempenho da última versão híbrida **4.0** do programa.

4.7.1 Comparação das versões paralelas

As três últimas etapas atingiram bons resultados em termos de eficiência paralela e ganhos acumulados.

TABELA 8 – tabela de ganhos máximos com relação à versão anterior

Etapa	Versão	Comentário	Ganho Relativo	Ganho Absoluto
Solução	s0.2	BiCGStab	377.80	377.80
Montagem	s1.0	elemento-por-elemento	0.01	4.07
	s1.0a	otimizações manuais	2.42	8.55
Paralelização MPI	P2.0	sem balanceamento	71.15	526.42
	P2.0.1	balanceamento simples	1.11	541.99
	P2.0.2	balanceamento adaptativo	1.20	542.86
	P2.1	lado dir. da eq., da m. func. e da diag. paralelizados	3.45	997.08
Armazenamento em Memória Disponível	P3.0	armazenamento geral	1.40	1201.41
	P3.2	armazenamento local	19.63	7328.57
Paralelização Híbrida	h4.0	paralelização híbrida	2.15	13153.85

A última versão da etapa de paralelização MPI (**P2.1**) alcança eficiência paralela entre 80% e 115% em relação à versão elemento-por-elemento otimizada **s1.0a** e os ganhos da última versão **h4.0** em relação à versão **p2.1** variam entre $22,3$ e $35,7$ para problemas na faixa de 120K a 194K elementos de contorno.

O gráfico da Fig. 17 recapitula esses resultados para as últimas versões paralelas de cada etapa com tempos de execução em escala linear.

4.7.2 Desempenho da última versão do programa

A seção anterior mostra que, uma vez que se tenha adotado a montagem do sistema elemento-por-elemento, as etapas posteriores atingiram bons resultados.

Entretanto, uma vez que a montagem elemento-por-elemento aliada à solução iterativa aumenta centenas de vezes o tempo de montagem, uma questão válida seria determinar se essa estratégia realmente produz bons resultados se comparada com a montagem clássica aliada a uma solução do sistema paralelizada em diversos nós retirada de uma biblioteca numérica otimizada, como a biblioteca ScaLapack.

Embora este trabalho não contenha uma implementação com essa biblioteca, pode-se

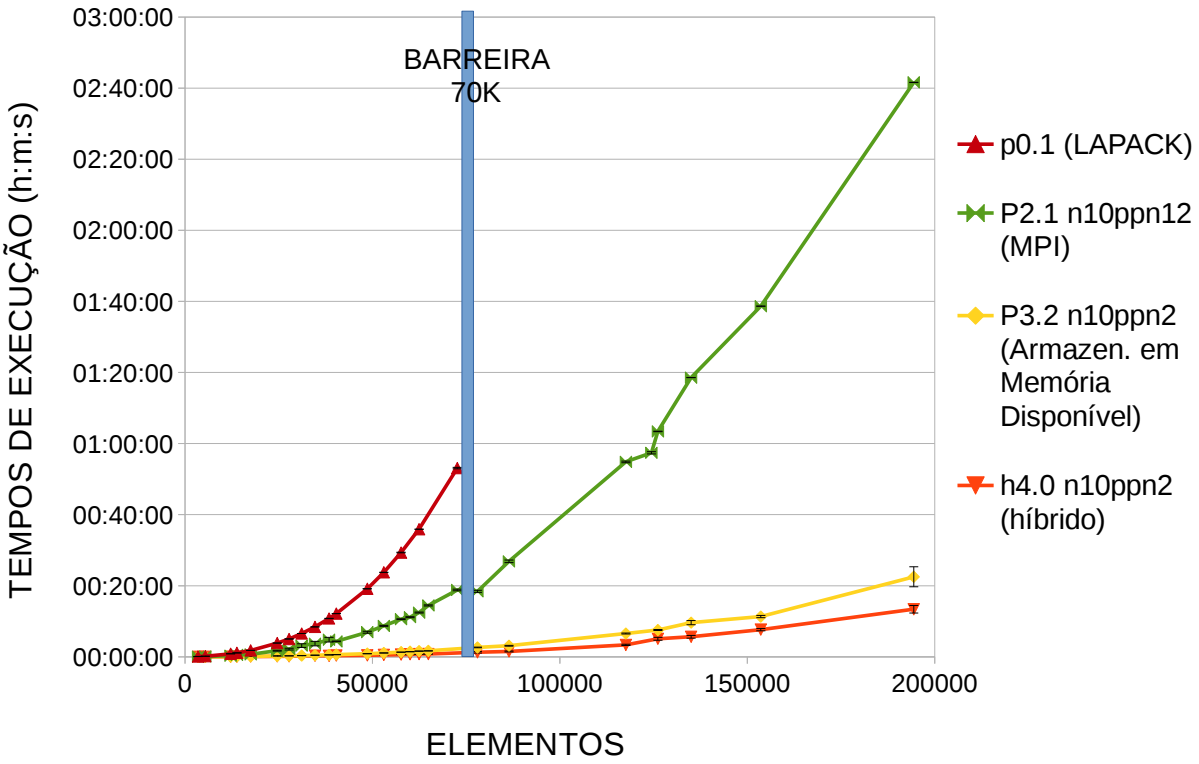


FIGURA 17 – Tempos de Execução das Versões Paralelas: p0.1, P2.1, P3.2 e h4.0

inferir indiretamente a qualidade dos resultados obtidos aqui através de uma comparação com os resultados obtidos com a biblioteca numérica Lapack.

Em primeiro lugar, lembrando que o Lapack utiliza todos os núcleos de processamento de um nó do cluster, dividir os resultados obtidos na versão **p0.1** por dez é o mesmo que admitir que essa versão seja perfeitamente paralelizável, sem nenhum custo administrativo ou de comunicação, entre os dez nós e os cento e vinte núcleos de processamento utilizados. Além disso, também se está admitindo que todas as rotinas são perfeitamente paralelizadas, não somente a rotina de solução, mas também as rotinas de montagem e de entrada e saída de dados. É, portanto, admissível supor que os valores obtidos com uma paralelização da solução com uma biblioteca otimizada em dez nós não seriam menores que um décimo dos valores obtidos na versão **p0.1**. Por simplicidade, essa versão ideal será chamada de versão **(1/10 * p0.1)**.

O gráfico da Fig. 18 compara os tempos de execução das versões **(1/10 * p0.1)** e

h4.0 para todos os problemas anteriores à "barreira de memória física" de 70K.

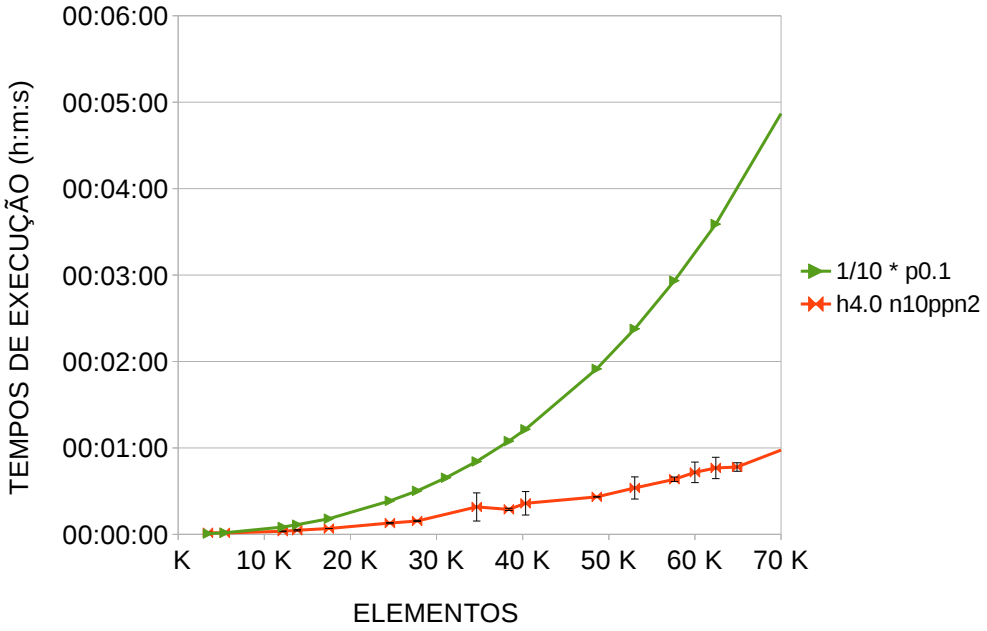


FIGURA 18 – Comparação dos Tempos de Execução para problemas anteriores à barreira de memória: (1/10 * p0.1) e h4.0

Interpolando-se os tempos de execução da versão **(1/10 * p0.1)** através de um polinômio de ordem cúbica, os resultados conhecidos são reproduzidos com cerca de um segundo de tolerância. Por extrapolação, pode-se encontrar os tempos de execução para os problemas além da "barreira de 70K". Os dados obtidos dessa maneira são comparados como os tempos de execução medidos da versão **h4.0** na Fig. 19.

Não se considerou problemas com mais de 194 K elementos de contorno porque a matriz resultante ultrapassaria a soma das memórias físicas dos nós do cluster e, como a versão **(1/10 * p0.1)** necessita da matriz inteira armazenada em memória, a execução de problemas maiores só seria possível com paginação da memória em disco.

Os ganhos da versão **h4.0** em relação à versão **(1/10 * p0.1)** são mostrados na Fig. 20. Nota-se na Fig. 20a que, para problemas com $10K \leq n < 70K$, o ganho varia entre $2,4$ e $4,7$, oscilando em torno de $4,5$ para problemas com $50K \leq n < 70K$. Além da barreira física (Fig. 20b), com $70K \leq n \leq 194K$, o ganho oscila de $4,6$ a $5,8$. A clara tendência crescente da curva do ganho indica que o *hardware* é melhor utilizado quanto maior

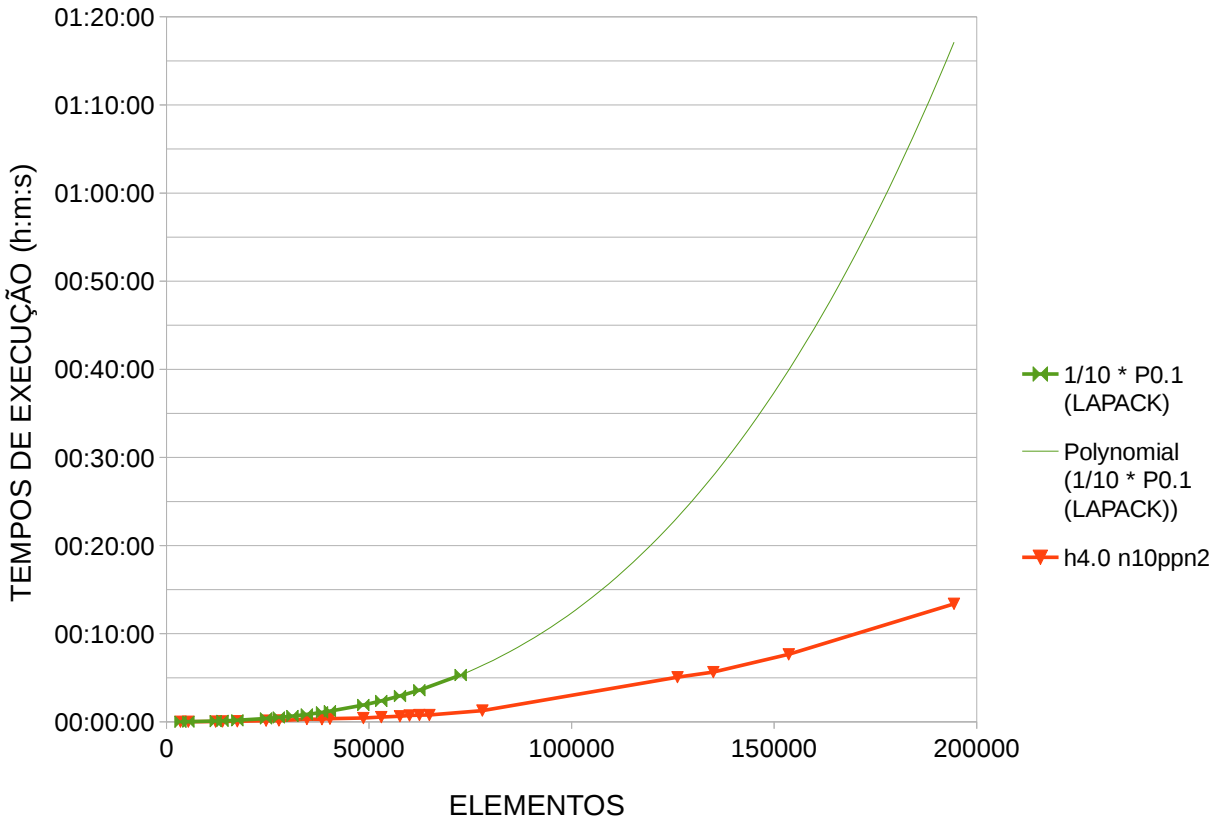
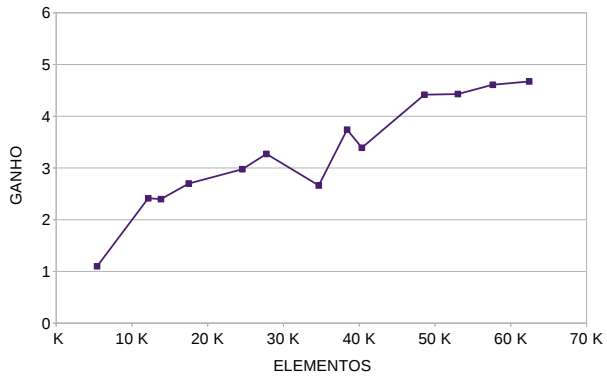


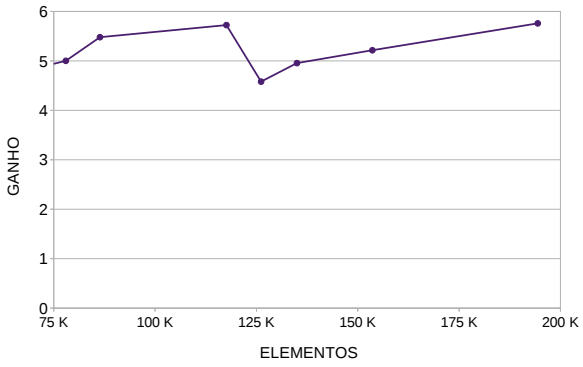
FIGURA 19 – Comparação dos Tempos de Execução: 1/10 * p0.1 e h4.0

o tamanho do problema.

Portanto, apesar do aumento do processamento da montagem do sistema elemento-por-elemento, a versão **h4.0**, com paralelismo híbrido e armazenamento das colunas, fornece ganhos consideráveis com relação aos tempos de execução obtidos com uma implementação ideal de uma solução direta de biblioteca numérica paralela e os maiores ganhos são obtidos para problemas na faixa de 70 K a 194 K elementos de contorno. Ao contrário da versão **((1/10 * p0.1))**, a versão **h4.0** resolve problemas com mais de 195 K elementos de contorno calculando as colunas que não podem ser armazenadas e sem efetuar paginação em memória.



(a) Antes da barreira de memória.



(b) Além da barreira de memória.

FIGURA 20 – Ganhos no tempo de execução: (1/10 * p0.1) / h4.0

5 CONCLUSÃO

No início do trabalho, definiu-se como objetivo a alteração do programa legado, cujo núcleo consiste na formulação clássica do MEC para problemas de potencial 3D, de modo a utilizar de forma mais eficiente os recursos de **processamento**, **armazenamento** e **comunicação** oferecidos por um *cluster* de computadores. A utilização dos recursos oferecidos pelo ambiente computacional varia bastante nas duas primeiras etapas e se torna gradativamente mais equilibrada ao longo das últimas três etapas do processo de trabalho.

Na primeira etapa, há três situações distintas. A versão original **s0.0** utiliza somente um dos doze núcleos de processamento disponíveis em um dos nós do cluster e uma pequena fração da memória física disponível. O recurso sobrecarregado dessa versão é o processamento decorrente da complexidade cúbica da solução direta. A versão **P0.1**, com rotina de solução de biblioteca numérica, distribui melhor o processamento da solução entre os doze núcleos e utiliza totalmente a memória física disponível para armazenar a matriz do sistema. O recurso sobrecarregado dessa versão passa a ser a memória, devido à ordem quadrática do tamanho da matriz do sistema. A versão **s0.2**, com uma rotina de solução iterativa, diminui o grau da complexidade da solução para quadrático e também utiliza totalmente o recurso de armazenamento com tempos de execução semelhantes à versão **P0.1** para a mesma gama de problemas, porém utilizando apenas um dos núcleos de processamento. O recurso sobrecarregado dessa versão também é a memória.

Na segunda etapa, ainda utilizando apenas um dos núcleos de processamento, a montagem elemento-por-elemento reduz a complexidade do requisito de armazenamento para linear e multiplica o trabalho da rotina de montagem por $8i$, onde i é o número de iterações da solução. O recurso sobrecarregado dessa fase é o processamento.

Na terceira etapa, com a paralelização MPI, tanto a rede como todos os núcleos de processamento dos dez nós utilizados passam a contribuir no processamento do programa pela primeira vez. Paralelizando-se todas as rotinas, com exceção das rotinas de distribuição de

tarefas e entrada e saída, a eficiência paralela oscila entre 83% e 115%. Porém a ociosidade da rede é evidente, pois a utilização de redes com desempenhos diferentes produz resultados semelhantes. Além disso, boa parte da memória não é utilizada. O recurso sobrecarregado dessa etapa é o processamento.

Na quarta etapa, a memória física disponível é utilizada para o armazenamento das colunas da matriz do sistema e o tempo de execução é drasticamente reduzido. A rede começa a ser melhor utilizada. A utilização da rede *infiniband* produz ganhos de até $1,4$ com relação ao uso da rede *ethernet*. Configurando o tamanho da área de armazenamento como o tamanho da matriz do maior problema resolvido na primeira etapa, a execução MPI n10ppn2 é mais eficiente que a execução MPI n10ppn12. Para n10ppn2, os únicos recursos claramente ociosos são os dez núcleos de processamento de cada nó do *cluster* que não participam da execução MPI.

Na quinta etapa, os dez processadores anteriormente ociosos passam a ser utilizados pelas linhas de execução OpenMP. A rede também passa a ser melhor utilizada e os tempos de execução com o uso da rede *infiniband* chegam a ser várias vezes maiores que os tempos com a rede *ethernet*.

A última versão híbrida do programa **h4.0** apresenta boa utilização da memória física e da rede do ambiente computacional. Além disso, todos os doze núcleos de cada um dos dez nós utilizados do cluster participam do processamento.

Para problemas cujas matrizes possam ser armazenadas na soma das memórias disponíveis pelos nós utilizados, a implementação apresentada na versão **h4.0** produz bons resultados se comparada com a utilização de uma biblioteca numérica idealizada de solução direta. Além disso, enquanto a versão com a biblioteca numérica idealizada efetuaria paginação em disco com sistemas lineares maiores que a soma das memórias físicas dos nós, a versão **h4.0** é capaz de resolver esses problemas calculando as colunas que não podem ser armazenadas.

5.1 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

Uma continuação natural deste trabalho é agregar à última versão **h4.0** as antigas funcionalidades do programa legado e resolver alguma classe dos problemas práticos para os quais o programa foi originalmente escrito.

Um próximo trabalho seria a utilização de outra formulação do MEC, com complexidade de montagem $\mathcal{O}(n \log n)$, como ACA ou MP, para a resolução de problemas maiores que 250 K.

Outras possíveis opções de trabalho seriam: otimizações mais agressivas, adequadas aos tamanhos das caches, conforme (CHELLAPPA; FRANCHETTI; PUSCHEL, 2008); testes de outras combinações entre as configurações do MPI e OpenMP, como fixação de linhas de execução e processos a núcleos de processamento específicos, algoritmos diferentes de distribuição de dados (*broadcast*); utilização das centenas de processadores da GPU disponível em um dos nós do *cluster* do Lactec; utilização de bibliotecas numéricas que forneçam soluções iterativas de sistemas lineares, como a biblioteca PETSc, que permitiria a comparação entre diversos métodos de solução Krylov.

REFERÊNCIAS

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**, ACM, New York, NY, USA, p. 483–485, 1967.

ANDERSON, E. *et al.* **LAPACK Users' Guide**. Philadelphia, PA: SIAM, 1999.

ARAUJO, F.; AZEVEDO, E.; GRAY, L. Boundary-element parallel-computing algorithm for the microstructural analysis of general composites. **Computers and Structures**, v. 88, p. 773–784, 2010.

ATASEVEN, Y. **Parallel Implementation of the Boundary Element Method for Electromagnetic Source Imaging of the Human Brain**. Tese (Doutorado) — Middle East Technical University, 2005.

BARRA, L.; COUTINHO, A.; MANSUR, W.; TELLES, J. Iterative solution of BEM equations by GMRES algorithm. **Computers & Structures**, v. 44, n. 6, p. 1249–1253, 1992.

BEBENDORF, M.; KRIEMANN, R. Fast parallel solution of boundary integral equations and related problems. **Comput. Vis. Sci.**, Springer-Verlag, Berlin, Heidelberg, v. 8, n. 3-4, p. 121–135, 2005.

BEBENDORF, M.; RJASANOW, S. Adaptive low-rank approximation of collocation matrices. **Computing**, Springer-Verlag, v. 70, n. 1, p. 1–24, 2003.

BECKER, A. A. **The Boundary Element Method in Engineering - A Complete Course**. Cambridge, UK: McGraw-Hill, 1992.

BEER, G.; SMITH, I.; DUENSER, C. **The Boundary Element Method with Programming**. Wien: Springer-Verlag, 2008.

BHARADWAJ, R.; WINDEMUTH, A.; SRIDHARAN, S.; HONIG, B.; NICHOLLS, A. The fast multipole boundary element method for molecular electrostatics: An optimal approach for large systems. **Journal of Computational Chemistry**, John Wiley & Sons, Inc., v. 16-7, p. 898–913, 1995.

BINEGAR, B. **Partial Differential Equations: Lecture Notes**. Oklahoma, OK, USA: Oklahoma State University, 1996. Disponível em: <https://www.math.okstate.edu/ binegar/5233-S96/5233-l20.pdf>.

BREBBIA, C.; TELLES, J.; WROBEL, L. **Boundary Element Techniques: Theory and Applications in Engineering**. Berlin, Heidelberg: Springer-Verlag, 1984.

CARRAZEDO, R.; LACERDA, L. A. Identificação da resistividade do solo em meios estratificados utilizando método dos elementos de contorno e algoritmos genéticos. **XXX CILAMCE - Congresso Ibero Latino Americano de Métodos Computacionais em Engenharia**, 2009.

CHANDRA, R. *et al.* **Parallel Programming in OpenMP**. San Francisco, CA: Morgan Kaufmann Publishers, 2001.

CHELLAPPA, S.; FRANCHETTI, F.; PUSCHEL, M. How to write fast numerical code: A small introduction. **Lecture Notes in Computer Science**, Springer, v. 5235, p. 196–259, 2008.

COSTA, E. **Aplicação dos Métodos dos Elementos de Contorno e das Soluções Fundamentais Usando Funções de Green para Problemas Acústicos**. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2013.

COSTABEL, M. Principles of boundary element methods. **Computer Physics Reports**, Elsevier, v. 6, p. 243–274, 1987.

CUNHA, M.; TELLES, J.; COUTINHO, A. A portable parallel implementation of a boundary element elastostatic code for shared and distributed memory systems. **Advances in Engineering Software**, v. 35, n. 7, p. 453–460, 2004.

DAVIES, A. J. The boundary element method on the ICL DAP. **Parallel Computing**, v. 8, p. 335–343, 1988. Proceedings of the International Conference on Vector and Parallel Processors in Computational Science III.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. gprof: a Call Graph Execution Profiler. **20 Years of the ACM/SIGPLAN Conference on Programming Languages Design and Implementation (1979-1999): A Selection**, ACM - Association of Computing Machinery, p. 49–57, 2003.

GRAMA, A.; GUPTA, A.; KARYPIS, G.; KUMAR, V. **Introduction to Parallel Computing**. 2nd. ed. Harlow - England: Addison-Wesley, 2003.

GRAMA, A.; KUMAR, V.; SAMEH, A. Parallel hierarchical solvers and preconditioners for boundary element methods. **Proceedings of the 1996 ACM/IEEE Conference on Supercomputing**, IEEE Computer Society, Washington, DC, USA, v. 34, 1996.

GREENGARD, L.; ROKHLIN, V. A new version of the fast multipole method for the laplace equation in three dimensions. **Acta Numerica**, Cambridge University Press, v. 6, p. 229–269, 1997.

GUIGGIANI, M.; GIGANTE, A. A general algorithm for multidimensional Cauchy principal value integrals in the boundary element method. **ASME Journal of Applied Mechanics**, v. 57, p. 906–915, 1990.

GUSTAFSON, J. L. Reevaluating Amdahl's law. **Commun. ACM**, ACM, New York, NY, USA, v. 31, n. 5, p. 532–533, may 1988.

HENESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design: The Hardware/Software Interface**. 4th. ed. Waltham - USA: Elsevier, 2009.

HENESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A quantitative approach**. 5th. ed. Waltham - USA: Elsevier, 2012.

HERNÁNDEZ, J. E. **An Indirect Boundary Element Formulation Based On Multipolar Expansion Technique For An Efficient Solution For Large Fluid Flow Problems**. Tese (Doutorado) — Wessex Institute of Technology, University of Wales, 1998.

HPC Advisory Council. **Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing**. USA, 2009.

Intel Corp. **Intel Math Kernel Library 11.1 for Linux OS User's Guide**. USA, 2013.

Intel Corp. **Intel Trace Analyzer 8.1 Reference Guide**. USA, 2013.

Intel Corp. **Intel VTune Amplifier 2013**. USA, 2013.

JARP, S.; LAZZARO, A.; LEDUC, J.; NOWAK, A. Evaluation of the Intel Westmere-EP server processor. **CERN Openlab 2010**, CERN: European Organization for Nuclear Research, Geneva, Swiss, 2010.

KAMIYA, N.; IWASE, H.; KITA, E. Performance evaluation of parallel boundary element analysis by domain decomposition method. **Engineering Analysis with Boundary Elements**, v. 18, n. 3, p. 217–222, 1996. Parallel BEM and Related Methods and Algorithms.

KELLEY, C. **Iterative Methods for Linear and Nonlinear Equations**. Philadelphia, PA: SIAM, 1995.

KENNEDY, K.; ALLEN, R. **Optmizing Compilers for Modern Architectures: A Dependence-based Approach**. 1st. ed. San Francisco - USA: Morgan Kaufmann, 2001.

KREIENMEYER, M.; STEIN, E. Efficient parallel solvers for boundary element equations using data decomposition. **Engineering Analysis with Boundary Elements**, v. 19, n. 1, p. 33–39, 1997.

LABAKI, J.; FERREIRA, L. O. S.; MESQUITA, E. Constant boundary elements on graphics hardware: a GPU-CPU complementary implementation. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, Scielo, v. 33, p. 475–482, 2011.

LACERDA, L. A.; SILVA, J. M. d. A dual BEM genetic algorithm scheme for the identification of polarization curves of buried slender structures. **Computer Modeling in Engineering & Sciences**, v. 14, p. 153–160, 2006.

LAGE, C.; SCHWAB, C. Wavelet Galerkin algorithms for boundary integral equations. **Journal on Scientific Computing**, SIAM, v. 20, n. 6, p. 2195–2222, 1999.

LIU, Y. **Fast Multipole Boundary Element Method: Theory and Applications in Engineering**. New York: Cambridge University Press, 2009.

Los Alamos National Laboratory. **TAU Reference Guide**. Oregon, USA, 2012.

MAERTEN, F. Adaptive Cross-Approximation applied to the solution of system of equations and post-processing for 3d elastostatic problems using the boundary element method. **Engineering Analysis with Boundary Elements**, v. 34, n. 5, p. 483–491, 2010.

MOLKA, D.; HACKENBERG, D.; SCHONE, R.; MULLER, M. Memory performance and cache coherency effects on an Intel nehalem multiprocessor system. **Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on**, p. 261–270, Sept 2009. ISSN 1089-795X.

QUEVEDO, L.; HANSRA, B.; MORRA, G.; BUTTERWORTH, N.; MILLER, R. Oblique mid ocean ridge subduction modelling with the parallel fast multipole boundary element method. **Computational Mechanics**, Springer-Verlag, v. 51, p. 455–463, 2012.

RAUBER, T.; RÜNGER, G. **Parallel Programming for Multicore and Cluster Systems**. Berlin: Springer-Verlag, 2010.

Rice University. **User's Manual For HPCToolkit 5.3.2**. Houston, TX, 2013.

RJASANOW, S.; STEINBACH, O. **The Fast Solution of Boundary Integral Equations**. New York, NY: Springer, 2007.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. 2nd. ed. Philadelphia, PA: SIAM, 2003.

SHI, Y. Reevaluating Amdahl's law and Gustafson's law. **Computer Sciences Department, Temple University (MS: 38-24)**, 1996.

SINGER, J. K. **The Parallel Fast Multipole Method in Three Dimensions**. Tese (Doutorado) — University of Houston, 1995.

SLEIJPEN, G.; FOKKEMA, D. BiCGstab(l) for linear equations involving unsymmetric matrices wiht complex spectrum. **Electronic Transactions on Numerical Analysis**, Kent State University, v. 1, p. 11–32, 1993.

SLEIJPEN, G.; VORST, H. V. D.; FOKKEMA, D. BiCGstab(l) and other hybrid Bi-CG methods. **Numerical Algorithms**, Baltzer Science Publishers, Baarn/Kluwer Academic Publishers, v. 7, n. 1, p. 75–109, 1994.

SYMM, G. T. Boundary elements on a distributed array processor. **Engineering Analysis**, v. 1, n. 3, p. 162–165, 1984.

VALENTE, F.; PINA, H. Conjugate gradient methods for three-dimensional bem systems of equations. **Engineering Analysis with Boundary Elements**, Elsevier, v. 30, p. 441–449, 2006.

VORST, H. V. D. Bi-CGStab: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. **Journal of Science and Statistics Computation**, SIAM, v. 13, p. 631–44, 1992.

WOLF, W. R.; LELE, S. K. Wideband fast multipole boundary element method: Application to acoustic scattering from aerodynamic bodies. **International Journal for Numerical Methods in Fluids**, John Wiley & Sons, Ltd., v. 67, p. 2108–2129, 2011.

XIAO, H.; CHEN, Z. Numerical experiments of preconditioned Krylov subspace methods solving the dense non-symmetric systems arising from BEM. **Engineering Analysis with Boundary Elements**, ELSEVIER, v. 31, p. 1013–1023, 2007.

ZHAO, Y.; TONG, C.; JU, Z. The Multithreading Parallel ACA Algorithm Based on OpenMP. **Business, Economics, Financial Sciences, and Management**, v. 143, p. 607–614, 2012.