Um Framework para Análise Seqüencial e em Paralelo de Sólidos Elásticos pelo Método dos Elementos Finitos

Bianca de Almeida Dantas

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para obtenção do título de **Mestre em Ciência da Computação.**

ORIENTADOR: Prof. Dr. Paulo Aristarco Pagliosa

CO-ORIENTADOR: Prof. Dr. Henrique Mongelli

Durante parte da elaboração desse projeto a autora recebeu apoio financeiro da CAPES.

Campo Grande - MS
2006

Aos meus pais Adelina e Felix.

E pluribus unum.

Agradecimentos

Em primeiro lugar, agradeço a Deus por ter me dado tudo o que tenho e por ter garantido que eu sempre tivesse as pessoas certas ao meu lado. Existem muitas dessas pessoas a quem preciso agradecer neste momento, espero não me esquecer de nenhuma aqui.

À minha querida mãe Adelina e ao meu querido pai Felix, os amores de minha vida, que durante a minha vida inteira sempre estiveram ao meu lado, ensinando que temos que sempre seguir em frente, enfrentando as adversidades, que me ensinaram a ter força e, ao mesmo tempo, não deixaram de me dar colo quando precisei. Obrigada por existirem!

Ao meu orientador Paulo Aristarco Pagliosa, por ter sido um amigo sempre presente, quem mais confiou em mim em minha vida acadêmica e me deu, desde a graduação, a inspiração e os ensinamentos para conseguir passar por esta etapa. Obrigada por ter sido meu "pai" acadêmico.

Ao professor Henrique Mongelli que co-orientou este trabalho e me forneceu o embasamento necessário para utilização de paralelismo e que esteve sempre presente para minhas eventuais dúvidas durante o desenvolvimento do projeto.

Agradeço à minha tia e madrinha Edilce, por ter sido sempre minha amiga, dando apoio e amizade exemplares, sendo sempre um exemplo de pessoa. Agradeço também à minha prima Laís e ao meu tio Alício por, juntamente com minha tia, terem me apoiado quando cheguei a Campo Grande.

Alguns amigos tiveram papel fundamental durante o meu mestrado e, entre eles, merece destaque a minha amiga Marcia (popularmente conhecida como Marcinha). Obrigada por sempre estar disposta a ouvir minhas reclamações, a dar conselhos e palavras de otimismo e a me fazer sair dos meus momentos "nerds" e me fazer ir ao cinema de vez em quando, sendo minha companhia mais constante durante todo o Mestrado.

Agradecimento especial aos meu amigos Anderson e Claudio, por juntamente com Marcia e eu, formarem "O Quarteto Fantástico" e por terem sido ao longo dos anos fonte de muita descontração e alegria, com suas tiradas sarcásticas e por me fazerem sempre extrair algo bom de todas as situações. Ao meu amigo Carlos Juliano, o B2, que me acompanhou e ajudou em todos os momentos bons e ruins do Mestrado, desde trabalhos em grupo até vagar pelos corredores da UFMS tentando explicar o inexplicável e que, juntamente, com a Amandita, formam um casal de amigos exemplar.

Ao meu tio Jaime pelo constante encorajamento e pelas palavras de humor.

Agradeço aos amigos Anderson (de novo!), Suzana e Maria Tânia por estarem sempre ao meu lado e por me acompanharem, de "livre e espontânea vontade", nos testes realizados durante a noite na UFMS. À Larissa, minha amiga de longa data, que sempre me deu palavras de apoio e me deu acesso aos artigos da UNICAMP, além de me ajudar com as aulas na fase final deste trabalho. À amiga Carulina Metzker por confiar em meu trabalho, sempre dando um grande apoio moral. À professora Sonia Regina Di Giacomo, por ser sempre presente e disposta a tirar minhas dúvidas desde o início da graduação. Aos amigos Christiane Brasil, Christiane Nishibe, Cristiano, Evely, Leonardo (que me auxiliou em diversas dúvidas técnicas), Liana, Marta, Noíza, Rafael, Renatinha Rodrigues, Rodrigo Sacchi e Valéria pelo apoio e companheirismo.

Aos demais companheiros de Mestrado: Adriana, Breno, Cláudia, Delair, Fabiano, Graciela, Graziela Araújo, Kellen, Kelly, Luciana Montera, Marcio e Pedro Bastos. Sucesso a todos.

Agradeço também aos professores Edson Cáceres, Katia França, Marcelo Henriques, Marcelo Siqueira e Nalvo Franco, do Departamento de Computação e Estatística que, de alguma forma, influenciaram na concretização deste trabalho. Ao chefe do DCT, Fabio Viduani, que forneceu a infra-estrutura necessária para os testes. Aos funcionários da secretaria, sempre prestativos, Beth, Giselda, Joacir, Joel, Leodir e Osmano.

Agradeço à CAPES pela bolsa de estudos fornecida.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Programas

Resumo

Dantas, B.A. *Um Framework para Análise Seqüencial e em Paralelo de Sólidos Elásticos pelo Método dos Elementos Finitos.* Campo Grande, 2006. Dissertação (Mestrado) – Universidade Federal de Mato Grosso do Sul.

O objetivo geral deste trabalho é o desenvolvimento de um framework em C++ para análise seqüencial ou em paralelo de sólidos elásticos através do método dos elementos finitos (MEF). O MEF baseia-se na discretização do domínio do sólido sendo analisado em uma malha de elementos, o que resulta na transformação das equações diferenciais que regem a elasticidade em sistemas de equações lineares que, para problemas tridimensionais, podem conter milhares de equações. Uma vez que a montagem e resolução de tais sistemas podem ser computacionalmente intensivas, faz sentido a busca por alternativas para tornar a análise através do MEF mais eficiente, destacando-se entre elas a computação paralela.

A análise em paralelo é realizada por p processadores, cada qual responsável por executar uma parte da análise. Para tal, o sólido é subdividido em p partes ou subdomínios, sendo cada subdomínio enviado a um processador e analisado separadamente. Nesta etapa, os pontos das fronteiras entre os subdomínios são considerados fixos. Após a análise de cada subdomínio em paralelo, os resultados globais da análise nas fronteiras podem ser determinados a partir da técnica da subestruturação. Os resultados internos em cada subdomínio são então calculados em paralelo a partir dos resultados nas fronteiras. Dentre as alternativas possíveis para decomposição de domínio, adotou-se neste trabalho uma baseada no particionamento de um grafo cujos vértices são os elementos finitos da malha decorrente da discretização do sólido; as arestas conectam vértices correspondentes a elementos adjacentes na malha. Para particionamento de grafos utilizou-se o pacote METIS, cujo código fonte foi encapsulado em uma classe C++ e anexado ao framework. Para envio de objetos entre os processadores durante a análise, implementou-se um esquema de serialização para transformar objetos em um fluxo contíguo de bytes e, inversamente, transformar um fluxo nos objetos que lhe deram origem. Fluxos são enviados entre os processadores com auxílio da biblioteca de comunicação MPI (*message passing interface*).

Palavras-chave: *mecânica computacional*, *computação paralela*, *programação orientada a objetos*, *decomposição de domínio*, *método dos elementos finitos*.

Abstract

Dantas, B.A. *Um Framework para Análise Seqüencial e em Paralelo de Sólidos Elásticos pelo Método dos Elementos Finitos*. Campo Grande, 2006. Dissertação (Mestrado) - Universidade Federal de Mato Grosso do Sul.

The objective of this work is the development of a C++ framework for finite element method (FEM) that provides sequential and parallel analysis of elastic solids. FEM is based on the discretization of the domain analyzed into a mesh, resulting on the transformation of the differential equations that define elasticity into a linear equations system which for tridimensional problems may contain thousands of equations. Since obtaining and solving those systems are computation intensive tasks, it makes sense searching for alternatives that make FEM analysis more efficient, emphasizing parallel computing.

Parallel analysis is performed through p processors, each one responsible for analyzing a part of the domain. In order to make it possible, the solid is split into p parts or subdomains. Each part is sent to a different processor and separately analyzed. In this phase the boundary points among subdomains are considered fixed. After each subdomain is analyzed in parallel, the global analysis results in the boundary nodes can be calculated using a substructuring technique. Internal results in each subdomain can then be obtained in parallel with the boundary nodes results. Among different possible alternatives for domain decomposition, we chose to use a technique based on partitioning the graph whose vertexes are equivalent to the mesh elements and edges connect vertexes which represent adjacent elements of the mesh. METIS package was chosen as the partitioning graphs tool, aiming that its component functions were encapsulated in a C++ class and attached to the framework. In order to send objects through different processors during analysis, one implemented a serialization scheme which transforms an object into a contiguous byte flow and, inversely, obtains the original object which originated a byte flow. MPI (message passing interface) communication library was used to grant sending and receiving byte flows.

Keywords: *computational mechanics*, *parallel computing*, *object-oriented programming*, *domain decomposition*, *finite element method*.

CAPÍTULO 1

Introdução

1.1 Motivação e Justificativas

A *simulação* em computador de um fenômeno físico envolve a implementação de *modelos* que representam os aspectos relevantes da estrutura e do comportamento dos objetos envolvidos no fenômeno. A simulação pode ser caracterizada por um pipeline onde as três fases principais são: *modelagem*, *análise* e *visualização* (Figura 1.1).

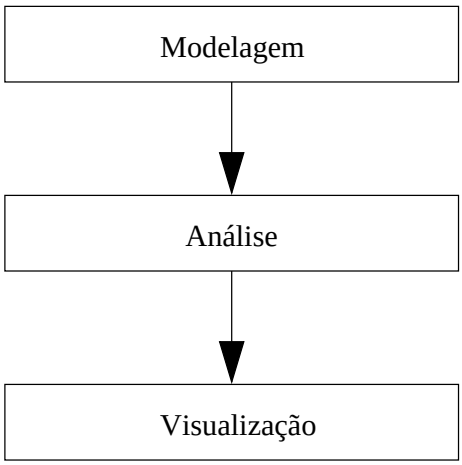


Figura 1.1: Pipeline de simulação.

Na fase de modelagem, ou pré-processamento, são construídos os modelos *geométrico*, *matemático* e *de análise* usados na simulação [PAG98]. Um modelo geométrico descreve a posição, dimensão e forma de um objeto, entre outros atributos (texturas, materiais, etc.). Um modelo matemático de um objeto é, em geral, definido por um conjunto de equações diferenciais formuladas a partir de hipóteses simplificadoras sobre a natureza dos materiais que compõem o objeto. Em mecânica computacional, um modelo matemático representa quantitativamente o comportamento de um corpo em termos de variáveis tais como deslocamentos, deformações e tensões. Além dessas

equações, o modelo define um conjunto mínimo de condições de contorno que garantem a unicidade da solução, tais como forças ou deslocamentos prescritos em determinadas regiões do contorno do corpo. Para resolver as equações diferenciais do modelo matemático, é necessário, para os casos gerais de geometria e de condições de contorno, a utilização de algum método numérico de análise, por exemplo, o método dos elementos finitos (MEF) e/ou o método dos elementos de contorno (MEC). Um modelo de análise é definido por uma malha de elementos (finitos e/ou de contorno) obtida a partir da discretização do volume e/ou da superfície de um objeto, em conjunto com as especificações das condições de contorno e das ações, ou carregamentos, nele aplicados. Ambas as formulações, MEF e MEC, transformam as equações diferenciais em um sistema de equações algébricas que, uma vez resolvido, fornece a solução do modelo matemático em pontos da malha chamados nós (geralmente pontos de interconexão de elementos). A fase de análise, ou processamento, toma como entrada um modelo de análise e produz como saída um conjunto de números que representam a solução discreta do modelo matemático nos pontos nodais da malha. A fase de visualização [SCH96], ou pós-processamento, transforma os dados de saída da análise em primitivos gráficos a partir dos quais podem ser sintetizadas imagens que permitem uma compreensão mais imediata dos efeitos do fenômeno sendo simulado.

O método dos elementos finitos é o mais amplamente utilizado em mecânica computacional [ASS03, ZIE94a, ZIE94b]. A técnica, matematicamente fundamentada em princípios variacionais, ou mais genericamente, em expressões de resíduos ponderados, consiste na divisão do volume de um sólido em um número finito de regiões ou células chamadas elementos finitos, os quais são interconectados através de um número finito de pontos nodais ou nós. A Figura 1.2 ilustra a divisão de um domínio bidimensional Ω em elementos finitos triangulares. O comportamento isolado de um elemento finito é especificado por um conjunto discreto de parâmetros normalmente associados às variáveis do problema físico em questão (deslocamentos, etc.). O comportamento global do domínio, igualmente especificado em função de tais parâmetros, pode ser determinado pela solução de sistemas de equações algébricas definidas a partir das contribuições individuais de todos os elementos finitos.

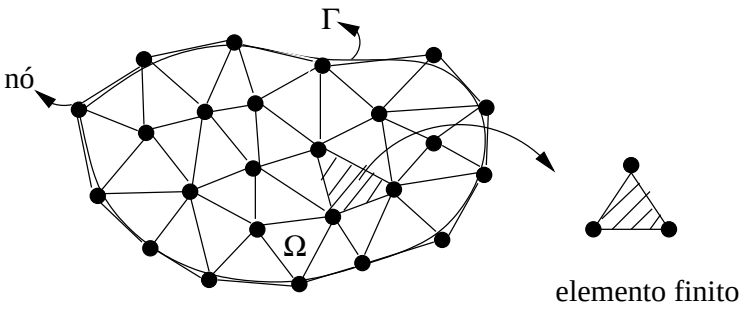


Figura 1.2: Domínio discretizado em elementos finitos.

A solução do sistema linear obtido a partir da formulação do MEF fornece respostas, para as variáveis que governam o problema em questão, nos pontos nodais do domínio discretizado. A partir dos valores nodais, pode-se estabelecer o valor em qualquer ponto do domínio do problema pela interpolação desses valores. Para esse propósito, cada elemento finito é dotado de *funções de interpolação* que possibilitam a determinação

de uma grandeza a partir dos valores dessa grandeza nos pontos nodais nos quais o elemento incide. Para conhecer o valor de alguma grandeza resultante do processo de análise numérica de um ponto P qualquer do domínio Ω, primeiramente, é necessário determinar a qual elemento finito o ponto pertence e, a partir daí, usar as funções de interpolação do elemento para obter o valor desejado.

Um método alternativo bastante empregado em mecânica computacional é o método dos elementos de contorno [BRE78, BRE84, KAN94], o qual se baseia na transformação das equações diferenciais que descrevem o comportamento mecânico no interior e na superfície, ou contorno, de um sólido em equações integrais definidas apenas em termos de valores no contorno. As equações integrais são resolvidas numericamente, a partir da discretização da superfície do sólido em células chamadas elementos de contorno. Um elemento de contorno é um trecho da superfície do contorno, definido geometricamente pela seqüência ordenada de nós sobre os quais o elemento incide. A Figura 1.3 ilustra a divisão de um domínio bidimensional Ω em elementos de contorno lineares. Na figura também são mostrados alguns pontos internos, comentados posteriormente.

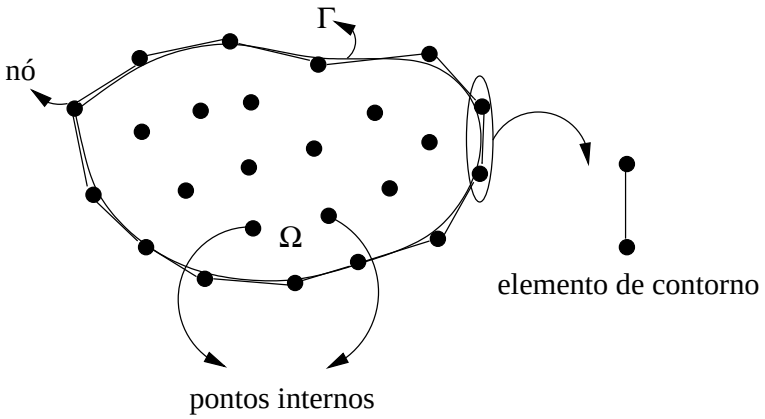


Figura 1.3: Domínio discretizado em elementos de contorno.

Uma das principais vantagens do MEC com relação ao MEF consiste na redução do número de equações do sistema de equações resultante. Esse fato pode ser confirmado levando em consideração que o MEC envolve a discretização apenas do contorno do sólido, ao contrário do MEF, no qual todo o volume é discretizado.

Os dados resultantes da análise numérica de um problema com a utilização do MEC fornecem respostas somente nos pontos nodais do contorno do sólido. Dado que cada elemento de contorno é dotado de funções de interpolação, pode-se, com o uso de tais funções e a partir dos valores nos pontos nodais, obter o valor de uma grandeza em quaisquer pontos da superfície do sólido. Para obtenção de respostas em um ponto P qualquer do contorno do sólido, realizam-se passos semelhantes àqueles executados no MEF. Primeiramente, determina-se sobre qual elemento de contorno P incide e, a partir dos valores nodais e das funções de interpolação desse elemento, calcula-se o valor desejado. Com o MEC também podem ser obtidas respostas em pontos discretos no interior do sólido, Figura 1.3, com o uso de fórmulas de integração numérica envolvendo os resultados obtidos no contorno.

Teoricamente, a solução numérica aproximada obtida com quaisquer dos métodos torna-se mais precisa com o uso de um maior número de elementos discretos. Entre-

tanto, à medida que crescem os números de pontos e de elementos do sistema discreto, também aumenta a dimensão do sistema de equações algébricas resultante. Aumentar o sistema ocasiona maior consumo de memória, bem como demanda maior capacidade de processamento para a montagem e resolução do sistema. Para exemplificar, considere o exemplo meramente acadêmico de análise elastostática pelo MEC de uma esfera oca com raio interno de 60 cm, raio externo de 80 cm e material com módulo de elasticidade igual 1000 MPa/cm^2 e coeficiente de Poisson $0,3$. A Figura 1.4 mostra a geometria da esfera, à esquerda; a malha de elementos de contorno é mostrada à direita. As células em vermelho nas regiões polares da esfera representam trechos de superfície engastados, ou seja, nos quais os deslocamentos nodais são totalmente restringidos; as células em azul denotam os trechos de superfície aos quais é aplicada uma pressão interna de 1 MPa/cm^2.

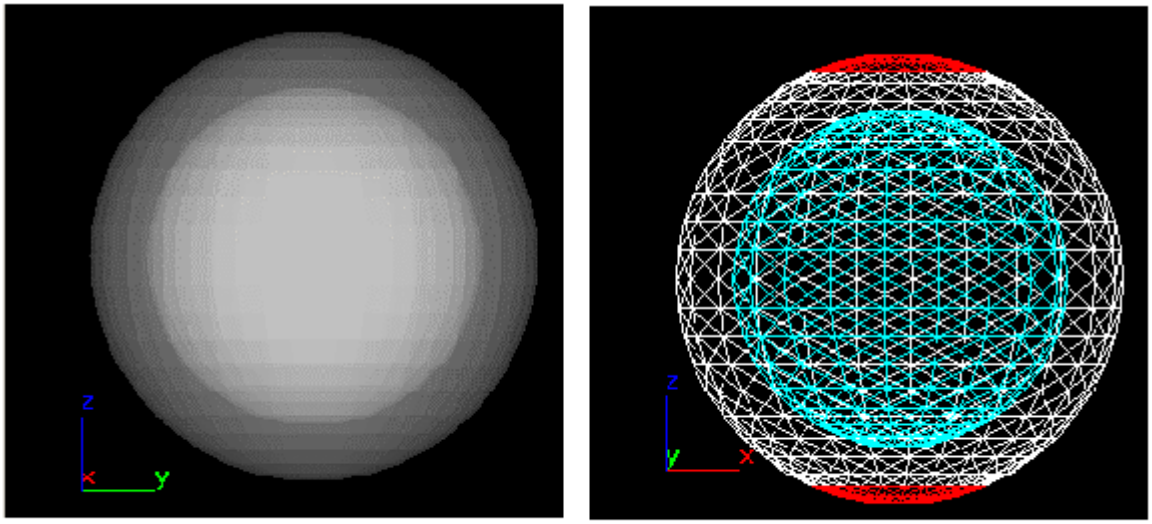


Figura 1.4: Esfera oca submetida a pressão intena.

A análise numérica do exemplo foi efetuada em um computador com processador Pentium 4 de $1,8$ GHz e com 256 MB de memória RAM. A malha resultante da discretização do exemplo é constituída de 613 nós e 1.134 elementos de contorno. Em cada nó é conhecido, para cada uma das três direções principais do sistema de coordenadas, o componente do vetor de deslocamento ou do vetor de força de superfície, mas não ambos. Em cada nó, portanto, há 3 incógnitas, denominadas *graus de liberdade* nodais. Com isso, o sistema de equações lineares resultante da análise com a utilização do MEC possui 1.839 incógnitas, ou seja, a matriz de coeficientes do sistema possui 3.381.921 números reais. Dado que um número real com precisão dupla ocupa 8 bytes, o sistema de equações ocupa aproximadamente $25,8$ MB de memória. Além disso, se cada elemento de contorno mantiver em memória suas duas *matrizes de influência*, cada qual com dimensão 9×9 — o que totaliza, para todo o contorno, 183.708 números reais —, haverá um consumo adicional de aproximadamente $1,4$ MB de memória. Por esses dados, tem-se que $27,2$ MB de memória são necessários para armazenar as matrizes utilizadas pelo MEC, o que corresponde a, aproximadamente, $10,6\%$ da memória RAM da máquina em que a análise foi realizada. O programa executou durante cerca de 3

horas, sendo as proporções dos tempos gastos com a execução das principais fases do MEC apresentadas na Tabela 1.1.

Fase	Porcentagem do tempo total
Montagem do sistema	87%
Resolução do sistema	9%
Outras	4%

Tabela 1.1: Tempos de execução do MEC para a esfera oca.

Para problemas reais, pode-se ter modelos com dezenas de milhares de nós e de elementos, resultando em sistemas lineares com milhões de equações. Com isso, justifica-se, em mecânica computacional, a busca por técnicas que diminuam o tempo de execução da análise numérica, sem diminuição da qualidade dos resultados obtidos. Dentre essas técnicas, destaca-se a computação paralela, a qual tem possibilitade de aumentar o desempenho de várias aplicações como, por exemplo, na solução de sistemas de equações lineares [SCO03] — uma das fases constituintes de ambos os métodos apresentados anteriormente. Estudos realizados também mostraram que a computação paralela pode ser utilizada em outras fases de ambos os métodos, tais como a geração da malha e a montagem dos sistemas de equações. Pelo exemplo apresentado, percebe-se que a maior parte do tempo de execução foi gasta com a montagem do sistema, o que motiva a paralelização dessa fase do processo, em primeiro lugar. Em um primeiro momento, uma das pretensões deste trabalho era a paralelização do MEF e do MEC. Para o MEF, pode-se encontrar na literatura vários trabalhos abordando esse tema [TSE86, DOR95, GRA97]. O mesmo não ocorre com o MEC, devido a sua inerente dependência do conhecimento de valores sobre todo o contorno, o que justificou a escolha pelo estudo do MEF apenas.

Muitos estudos vêm sendo realizados no sentido de paralelizar os métodos para solução de sistemas de equações lineares, tanto diretos quanto iterativos. A solução de tais sistemas, muitas vezes formados por um grande número de equações, é importante em aplicações nas mais diversas áreas, como por exemplo, dinâmica de fluidos, previsões meteorológicas, análise de redes eletrônicas [SMI90], além da análise mecânica de sólidos abordada neste trabalho. Pion e outros [PIO83] foram alguns dos primeiros a enfatizar a possibilidade de utilizar o paralelismo na fase de solução do sistema de equações lineares, com o intuito de aumentar o desempenho do MEF. Até então, já eram conhecidas as vantagens da paralelização da montagem do sistema.

Um dos paradigmas utilizados com o intuito de resolver sistemas de equações lineares é o da implementação de algoritmos seqüenciais sobre arquiteturas paralelas, com o uso de processadores disponíveis comercialmente [WIL96]. A utilização de métodos paralelos para a solução de sistemas lineares não é sempre uma boa alternativa, pois, dependendo do tamanho do sistema a ser resolvido, tais métodos não possuem desempenho melhor do que os algoritmos seqüenciais. Para tornar viável a utilização da computação paralela, a quantidade de processamento a ser realizado deve ser, de acordo com alguma métrica, significativa em comparação à quantidade de comunicação necessária entre os processadores [SCH00]. Geralmente, os métodos paralelos são adaptações dos métodos seqüenciais existentes como é o caso da fatoração LU paralela apresentada em [MUR95, MUR96]. Neste trabalho, optou-se, a princípio, por utilizar um método

seqüencial para resolução do sistema de equações lineares, deixando a implementação de métodos paralelos para uma futura expansão do sistema. Algumas alternativas para a solução de sistemas de equações com diferentes características podem ser encontrados em [SCO03].

Na paralelização de um problema, um dos aspectos fundamentais é a maneira como é feita a distribuição da *carga de trabalho* entre os processadores. Carga de trabalho é, informalmente, a quantidade de processamento que deve ser realizado por um processador para execução de determinada atividade. Idealmente, essa distribuição deve ser a mais balanceada possível, ou seja, cada processador deve realizar aproximadamente a mesma quantidade de processamento. No caso da fase de montagem do sistema de equações do MEF, a carga de trabalho é função do número de elementos da malha com os quais cada processador deve trabalhar. As diferentes implementações paralelas do MEF encontradas na literatura diferem, geralmente, nas técnicas adotadas para a distribuição da carga de trabalho — e, como conseqüência, da maneira através da qual será montado o sistema de equações lineares — e para a solução do sistema de equações. As principais técnicas de paralelização do MEF são baseadas em *decomposição do domínio* [GOP96], adotada no trabalho, a qual consiste na divisão da malha de elementos entre os processadores de forma a tentar garantir o balanceamento dessa divisão.

1.2 Objetivos e Contribuições

O objetivo geral deste trabalho é o desenvolvimento de um framework em C++ para análise seqüencial ou em paralelo de sólidos elásticos através do método dos elementos finitos. O modelo orientado a objetos proposto, parte inspirado no trabalho de McKenna [MCK97], é flexivel e extensível. O framework foi implementado a partir de alterações e extensões de um já existente denominado OSW [PAG98], destinado ao desenvolvimento de aplicações de modelagem em ciências e engenharia[1]. Além de aplicações de análise baseadas no framework, aplicações também foram desenvolvidas para geração e visualização dos casos de teste, mas o foco do texto está na descrição dos componentes do framework responsáveis apenas pela análise.

Os objetivos específicos são:

- Estudo das técnicas de paralelização do MEF com decomposição de domínio encontradas na literatura;
- Revisão de OSW e elaboração de um modelo orientado a objetos a partir do qual serão implementados os componentes do framework envolvidos na análise seqüencial e em paralelo do MEF;
- Implementação e descrição das classes do modelo proposto;
- Exemplificação e obtenção de medidas de desempenho de análises de sólidos elásticos realizadas com o framework.

Embora a tecnologia de elementos finitos permita a simulação dinâmica de corpos elastoplásticos tridimensionais, o modelo apresentado ao longo do texto não foi

[1]Uma aplicação de modelagem, no contexto de OSW, é um programa orientado a objetos de visualização e simulação baseada em física.

implementado com a intenção de desenvolver um framework que, na primeira versão, pudesse realizar todos os tipos de análise possíveis. No trabalho, apenas análise linear estática (com subestruturação ou não) foi implementada. No entanto, acredita-se que os tipos de análise abordados forneceram requisitos suficientes para elaboração do modelo orientado a objetos de tal sorte que o framework fosse flexível e pudesse ser mais facilmente estendido. Atingido esse objetivo, outros tipos de análise (não-linear, transiente, etc.), tanto seqüenciais quanto paralelas, podem ser incorporados ao framework através dos mecanismos de herança e polimorfismo.

Este é o primeiro trabalho utilizando computação paralela envolvendo o Grupo de Computação Gráfica e Mecânica Computacional e o Grupo de Algoritmos Paralelos e Distribuídos da UFMS. Pretende-se que o framework proposto possa servir de fundação para a implementação de outros modelos de análise, sejam baseados no MEF ou em outros métodos, tais como o MEC.

1.3 Resumo do Trabalho

Foram desenvolvidas duas aplicações baseadas no framework proposto, uma para análise seqüencial (com ou sem subestruturação) e outra para análise em paralelo de sólidos elásticos. Além disso, duas outras aplicações chamadas de FEG e FEV foram desenvolvidas para pré e pós-processamento, respectivamente, mas não são descritas no trabalho. FEG é responsável pela geração dos arquivos de dados necessários à criação dos modelos de análise; FEV é utilizada para visualização dos resultados de análise (campos de deslocamentos, tensões, etc.) no domínio e contorno de um sólido.

A interface gráfica de FEG possibilita a geração de sólidos simples tais como caixas, cilindros e domos (maciços ou vazados) e esferas (maciças ou ocas). A geometria dos sólidos é representada por malhas de tetraedros geradas por fontes baseados em triangulação de Delaunay [MÜL02][2]. FEG permite também a especificação de restrições nos graus de liberdade de pontos do contorno e a aplicação de carregamentos distribuídos e de pressão em trechos da superfície, bem como a definição das propriedades materiais do sólido gerado. Uma vez definido o modelo geométrico e seus atributos (material, restrições e carregamentos), estes dados são armazenados em um arquivo.

As aplicações de análise começam pela leitura dos dados de um arquivo gerado por FEG. A partir de tais dados é criada a malha de tetraedros que representa a geometria do sólido a ser analisado. Um filtro do framework, então, toma como entrada esta malha de tetraedros e as informações a respeito das restrições e carregamentos e gera como saída o modelo de análise do sólido. Para cada vértice da malha, um nó correspondente é criado e adicionado ao modelo de análise; para cada tetraedro da malha, um elemento finito correspondente é criado e adicionado ao modelo de análise. Por fim, o filtro adiciona também as restrições e os carregamentos ao modelo de análise. A análise é então efetuada e os resultados (deslocamentos, tensões, etc.) são armazenados em um ou mais arquivos que podem ser usados por FEV para o pós-processamento.

A análise seqüencial utiliza apenas um processador e considera o sólido como um todo. Alternativamente, pode-se utilizar a técnica de subestruturação. Nesta, o sólido

[2]Em um pipeline de simulação, um *fonte* é um objeto que representa um processo responsável pela criação de um modelo; um *filtro* é um fonte que transforma um modelo em outro e um *sumidouro* é um objeto que consome um modelo, por exemplo, transformando o modelo em imagens.

é subdividido em n partes e cada parte é analisada separadamente, considerando suas fronteiras fixas. Após essa primeira etapa da análise, as restrições dos pontos nodais das fronteiras são "relaxadas" para obtenção dos resultados reais nestes pontos. A partir destes, podem ser obtidos os resultados internos em cada subdomínio. Os componentes do framework envolvidos na análise são vários e descritos ao longo do texto.

A análise em paralelo é realizada por p processadores, cada qual responsável por executar uma parte da análise. Para tal, o sólido é subdividido em p subdomínios, sendo cada subdomínio enviado a um processador e analisado separadamente, como na subestruturação, mas agora em paralelo. Os resultados internos em cada subdomínio são também calculados em paralelo a partir dos resultados nas fronteiras.

A subdivisão, tanto no caso da análise seqüencial com subestruturação como no caso da análise em paralelo, é realizada por um filtro que toma como entrada o modelo de análise e gera como saída o modelo de análise particionado em p partes. Dentre as alternativas possíveis para decomposição de domínio, adotou-se neste trabalho uma baseada no particionamento de um grafo equivalente ao domínio. Dessa forma, o primeiro passo para realizar o particionamento é a obtenção de tal grafo, chamado, neste trabalho, de grafo dual. Neste, cada vértice equivale a um elemento finito e cada aresta conecta vértices equivalentes a elementos adjacentes na malha. Após a obtenção do grafo, o filtro realiza seu particionamento, o que equivale a particionar a malha de elementos finitos do modelo de análise. Para particionamento de grafos utilizou-se o pacote METIS [KAR98a], cujo código fonte foi encapsulado em uma classe C++ e anexado ao framework.

Para envio de objetos entre os processadores durante a análise em paralelo, implementou-se um esquema de *serialização* responsável por transformar objetos em um fluxo contíguo de bytes e, inversamente, transformar um fluxo nos objetos que lhe deram origem. Tais fluxos podem então ser enviados entre os processadores com auxílio de uma biblioteca de comunicação. A que foi adotada neste trabalho é a MPI (*message passing interface*).

Os resultados da análise baseada em decomposição do domínio em n partes (seqüencial ou paralela) são armazenados em n arquivos usados por FEV para visualização. Esta consiste na transformação de atributos de um objeto em imagens que representam informações sobre a estrutura e o comportamento do objeto. FEV possui componentes para visualização de escalares (mapa de cores, isolinhas e isosuperfícies) e de vetores (modelo deformado e *glyphs* orientados) [MÜL02] no contorno e domínio do sólido analisado.

1.4 Organização do Texto

O restante do texto é organizado em seis capítulos como descrito a seguir.

Capítulo 2
Modelo de Análise
No Capítulo 2 são descritas as principais funcionalidades das classes de objetos do framework envolvidas na representação de um domínio a ser analisado pelo MEF e seus componentes, tais como elementos finitos, nós, restrições e carregamentos.

Capítulo 3
Análise Seqüencial
No Capítulo 3 apresenta-se um resumo da formulação do método dos elementos finitos aplicado à análise seqüencial estática linear de sólidos elásticos, bem como o esquema computacional do MEF. São descritas as principais funcionalidades das classes de objetos do framework responsáveis pela análise e pelas passos da análise (manipulação das restrições, numeração dos graus de liberdade, e dimensionamento, montagem e solução do sistema linear, entre outras).

Capítulo 4
Decomposição do Domínio
O Capítulo 4 abrange o particionamento da malha de elementos finitos do modelo de análise. No capítulo são apresentados o pacote METIS e um resumo das técnicas de particionamento implementadas no pacote. São descritas as principais funcionalidades das classes de objetos do framework responsáveis pela decomposição de domínio.

Capítulo 5
Análise em Paralelo
O Capítulo 5 aborda a solução paralela do MEF adotada no trabalho. Para proporcionar uma visão mais ampla sobre o assunto, uma revisão bibliográfica sucinta sobre os trabalhos nessa área é apresentada. As principais funcionalidades das classes de objetos do framework relacionadas à serialização de objetos e comunicação via MPI são descritas.

Capítulo 6
Exemplos
O Capítulo 6 apresenta alguns exemplos de análises seqüencias e em paralelo executadas por aplicações desenvolvidas com o framework proposto. Os resultados obtidos são comparados e uma análise crítica dos mesmos é realizada.

Capítulo 7
Conclusão
No Capítulo 7 são apresentadas as conclusões obtidas com o trabalho e algumas possibilidades de trabalhos futuros.

CAPÍTULO 2

Modelo de Análise

2.1 Considerações Iniciais

Conforme visto no Capítulo 1, um modelo de análise é definido por uma malha de elementos (finitos, no contexto deste trabalho), os quais são interconectados em pontos chamados nós, e por condições de contorno dadas por restrições a graus de liberdade nodais e carregamentos aplicados a nós e contorno de elementos. Este capítulo introduz as classes de objetos que representam os componentes do modelo de análise implementado no framework.

A geometria de um modelo de análise é definida por uma malha de tetraedros a partir da qual o modelo é gerado. Genericamente, uma malha, ou modelo de decomposição por células, é uma coleção de células e de vértices. Um vértice é um ponto no espaço tridimensional que possui associadas coordenadas em um dado sistema global de coordenadas. A geometria da malha é definida pela coleção dos vértices que a compõem. Uma célula é caracterizada pela lista dos vértices sobre os quais ela incide, lista esta denominada lista de conectividade da célula. Tal coleção de células define a topologia do malha. Para cada célula da malha será criado um elemento finito correspondente; para cada vértice da malha será criado um nó correspondente no modelo de análise.

Um modelo de análise, portanto, é dado por duas malhas acopladas: de tetraedros e de elementos finitos. Justifica-se: as fases de pré e pós-processamento do pipeline de visualização manipulam apenas malhas de tetraedros (a primeira gera uma; a segunda se utiliza de uma para renderização de imagens com resultados da análise). A malha de elementos finitos é criada apenas na fase de processamento e descartada ao término da análise.

A Seção 2.2 introduz as classes das estruturas de dados utilizadas na implementação das coleções do modelo de análise e algumas classes acessórias. Na Seção 2.3 são descritas as classes associadas ao modelo de decomposição por células. Na Seção 2.4 são descritas as principais funcionalidades das classes que representam os demais componentes do modelo de análise.

2.2 Estruturas de Dados e Acessórios

Os dados do modelo de análise são encapsulados em estruturas de dados simples cujas classes de objetos são descritas nesta seção.

Classe `tDoubleListImp`

Uma lista duplamente encadeada de objetos de uma classe `T` é definida através da classe paramétrica `tDoubleListImp<T>`.

Os principais elementos de uma lista duplamente encadeada são os ponteiros `Head` e `Tail` que apontam, respectivamente, para o primeiro e para o último elemento da lista. A lista fornece métodos para:

- Adicionar elementos à lista. São fornecidos três métodos: `AddAtHead(e)` (adiciona o elemento `e` como primeiro elemento da lista), `AddAtTail(e)` (adiciona o elemento `e` ao final da lista) e `Add(e)` (invoca o método `AddAtTail(e)`);
- Remover elementos da lista. Essa funcionalidade é fornecida pelo método `Remove(e)`, que remove o elemento `e` da lista;
- Retornar o primeiro elemento da lista. O método `PeekHead()` retorna o primeiro elemento da lista;
- Retornar o último elemento da lista. O método `PeekTail()` retorna o último elemento da lista.

Uma lista duplamente encadeada é um contêiner, dessa forma, há uma classe que permite que os elementos do contêiner sejam percorridos. Isso é feito através de uma classe de iterador de listas duplamente encadeadas de objetos de uma classe `T`, implementada através de uma instância da classe paramétrica `tDoubleListIteratorImp<T>`.

Um iterador de listas duplamente encadeadas armazena como atributos um ponteiro para um objeto da classe `tDoubleListImp<T>` — que representa a lista que ele percorre — e um ponteiro para um objeto da classe `T`, que representa o objeto corrente. O iterador fornece métodos para:

- Reiniciar o iterador. O método `Restart()` faz com que o ponteiro `Current` aponte para o primeiro elemento da lista;
- Retornar o elemento corrente. O método `Current()` retorna o elemento atual do iterador, indicado pelo atributo `Current`;
- Iterar para o próximo elemento da lista. Dois métodos fazem com que o ponteiro `Current` aponte para o próximo elemento da lista: `operator++(int)` (retorna o elemento atual e faz com que `Current` aponte para o próximo elemento da lista) e `operator++()` (atualiza o ponteiro `Current` para o próximo elemento da fila e retorna o novo valor de `Current`);
- Iterar para o elemento anterior da lista. Dois métodos fazem com que o ponteiro `Current` aponte para o elemento anterior da lista: `operator--(int)` (retorna

o elemento atual e faz com que `Current` aponte para o elemento anterior a ele na lista) e `operator--()` (atualiza o ponteiro `Current` para o elemento anterior da fila e retorna o novo valor de `Current`).

Classe `tList`

Uma lista simplesmente encadeada de objetos de uma classe `T` é representada por um objeto da classe paramétrica `tList<T>`.

Similarmente às listas duplamente encadeadas, uma lista deve fornecer métodos para:

- Adicionar elementos à lista. São fornecidos três métodos: `AddAtHead(e)` (adiciona o elemento `e` como primeiro elemento da lista), `AddAtTail(e)` (adiciona o elemento `e` ao final da lista) e `Add(e)` (invoca o método `AddAtTail(e)`);
- Remover elementos da lista. O método `Remove(e)` retira o elemento `e` da lista; o método `RemoveAtHead()` remove o primeiro elemento da lista;
- Retornar o primeiro elemento da lista. O método `PeekHead()` retorna o primeiro elemento da lista;
- Inverter a lista. O método `Invert()` inverte a lista;
- Verificar se um elemento está presente na lista. O método `Contains(e)` verifica se o elemento `e` está na lista.

Para percorrer os elementos da lista, um iterador é fornecido e implementado pela classe paramétrica `tListIterator<T>`, cuja definição é semelhante ao iterador de listas duplamente encadeadas e, portanto, será omitida.

Classe `tArray`

A classe que implementa um *array* de objetos de uma classe `T` é representada pela classe paramétrica `tArray<T>`. Essa classe representa uma coleção de objetos da classe `T` e é derivada da classe `tFixedArray`.

A classe `tFixedArray<T>` possui como atributos um ponteiro `Data` para um objeto da classe `T`, que representa o início do *array* de objetos; e um inteiro sem sinal `Size`, que armazena o número de elementos armazenados. Seus principais métodos são as sobrecargas do operador colchete e a sobrecarga do operador igual.

Para iterar sobre os elementos de `tFixedArray<T>` existe a classe de iterador de elementos dessa classe, `tFixedArrayIterator<T>`, cujas funcionalidades e descrição são similares à classe `tDoubleListIteratorImp`, descrita anteriormente.

A classe `tArray` fornece métodos para:

- Adicionar elementos ao *array*. O método `Add(e)` adiciona o elemento na última posição do *array* e o método `AddAt(e, pos)` adiciona o elemento `e` na posição `pos` indicada pelo segundo argumento do método. Ambos os métodos retornam um valor lógico que indica se a inserção foi feita com sucesso;

- Remover elementos do *array*. O método `Remove(pos)` remove o elemento da posição indicada pelo argumento. O método `Remove(e)` remove o elemento passado como argumento. Ambos os métodos retornam um valor lógico que indica se a remoção foi feita com sucesso;
- Encontrar um elemento. O método `Find(e)` retorna o índice no *array* do elemento passado como argumento, ou -1, caso o elemento não seja encontrado.

Classe `tMatrix`

As matrizes são necessárias na simulação mecânica porque são utilizadas para armazenamento dos coeficientes das equações dos sistemas algébricos que representam o problema físico em questão. A classe `tMatrix` representa uma matriz de elementos numéricos — dados por números em ponto flutuante com precisão dupla (*double*) e possui métodos para:

- Retornar a matriz transposta;
- Retornar a matriz inversa;
- Calcular o determinante;
- Sobrecarregar os operadores que possibilitam soma, subtração, multiplicação, atribuição e comparação de matrizes.

Classe `t3DVector`

Um vetor tridimensional é representado por um objeto da classe `t3DVector` e, basicamente, armazena as três coordenadas que definem esse vetor no espaço, além de fornecer métodos para:

- Retornar seu versor;
- Negar o vetor;
- Normalizar o vetor;
- Calcular o produto interno com outro vetor;
- Calcular o produto vetorial com outro vetor;

Além desses métodos, são realizadas sobrecargas de operadores de forma a permitir a soma, subtração, multiplicação, atribuição e comparação de vetores.

2.3 Modelo de Decomposição por Células

Nesta seção são descritas as diferentes classes usadas na representação de malhas e seus componentes, ou modelo de decomposição por células.

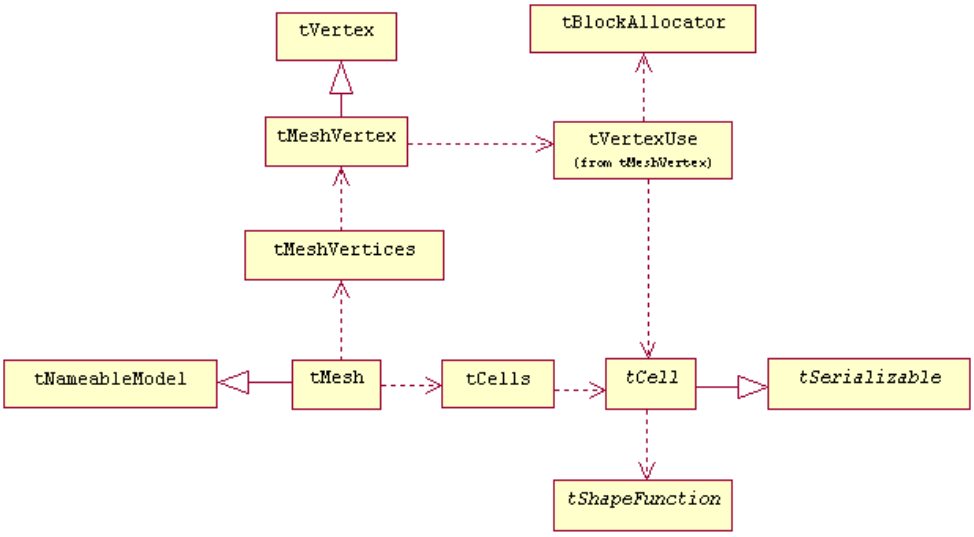


Figura 2.1: Diagrama de classes relacionadas à malha.

Classe `tMesh`

Uma malha é representada por um objeto da classe `tMesh` — derivada de `tNameableModel` —, que representa as informações relacionadas à geometria e à topologia do modelo de análise. Um objeto da classe `tMesh` possui um conjunto de vértices e outro de células e fornece funcionalidades para gerenciar os nós e as células que compõem o modelo, dentre as quais destacam-se:

- Fornecer número de células e de vértices;
- Fornecer vértices ou células específicos;
- Fornecer iteradores para vértices ou células;
- Adicionar e remover vértices ou células.

As coleções de vértices e células que compõem a malha são implementados em `tMesh` como listas duplamente encadeadas. O Programa 2.1 ilustra a definição parcial de `tMesh`.

O diagrama de classes que ilustra a classe `tMesh` e relacionadas é apresentado na Figura 2.1. A classe `tSerializable`, do diagrama de classes da Figura 2.1, indica que um objeto da classe que dela deriva pode ser "serializada", ou seja, pode-se criar um fluxo de bytes equivalente a tal objeto. O conceito de serialização é essencial na realização de análises paralelas e será explicado no Capítulo 5.

Classe `tVertex`

Um vértice representa um ponto no espaço tridimensional e é implementado pela classe `tVertex`. Um vértice é identificado por um número (representado pelo atributo da linha 36) e sua posição espacial é definida por um vetor 3D (atributo `Position` da linha 37). Os demais atributos são:

```
1   class tMesh:  public tNameableModel
2   {
3     public:
4       typedef tDoubleListImp<tMeshVertex> tVertices;
5       typedef tDoubleListIteratorImp<tMeshVertex> tVertexIterator;
6       typedef tDoubleListImp<tCell> tCells;
7       typedef tDoubleListIteratorImp<tCell> tCellIterator;
8       typedef tDoubleListImp<tMeshPartition> tPartitions;
9       typedef tDoubleListIteratorImp<tMeshPartition>
10        tPartitionIterator;
11
12      tMesh();
13      tMesh(const char*);
14      void Transform(const t3DTransfMatrix&);
15      void SetMaterial(const tMaterial&);
16      int GetNumberOfVertices() const;
17      tMeshVertex* GetVertex(int) const;
18      tVertexIterator GetVertexIterator() const;
19      void AddVertex(tMeshVertex*);
20      void DeleteVertex(tMeshVertex*);
21      void RenumerateVertices(bool = false);
22      int GetNumberOfCells() const;
23      tCell* GetCell(int) const;
24      tCellIterator GetCellIterator() const;
25      void AddCell(tCell*);
26      void DeleteCell(tCell*);
27      void RenumerateCells(bool = false);
28      int GetNumberOfPartitions() const;
29      tMeshPartition* GetPartition(int) const;
30      tPartitionIterator GetPartitionIterator() const;
31      void AddPartition(tMeshPartition*);
32      void DeletePartition(tMeshPartition*);
33      void RenumeratePartitions(bool = false);
34
35    protected:
36      tVertices Vertices;
37      int NextVertexId;
38      tCells Cells;
39      int NextCellId;
40      tPartitions Partitions;
41      int NextPartitionId;
42
43      DECLARE_SERIALIZABLE(tMesh);
44      ...
45  }; // tMesh
```

Programa 2.1: Classe `tMesh`.

- `Color` (linha 38). Objeto da classe `tColor` que representa a cor RGB do vértice;
- `Scalar` (linha 39). Armazena o valor de uma grandeza escalar qualquer atribuída ao vértice como, por exemplo, temperatura ou pressão;
- `Vector` (linha 40). Armazena o valor de uma grandeza vetorial qualquer atribuída ao vértice como, por exemplo, deslocamento ou força de superfície.

Classe `tMeshVertex`

Um objeto da classe `tMeshVertex` — derivada de `tVertex` — representa um vértice da malha e é definido no Programa 2.2. Cada vértice possui uma lista duplamente encadeada de seus usos, ou seja, de referências a todas as células que incidem sobre ele. O Programa 2.2 apresenta a definição dessa classe.

Classe `tMeshVertex::tUse`

Um objeto da classe `tMeshVertex::tUse` representa um "uso" de um vértice. Um vértice v possui uma lista de objetos dessa classe que, em conjunto, representam todas as células incidentes em v. Possui como atributo apenas um ponteiro para a célula que usa o vértice.

Classe `tCell`

A classe `tCell` define uma célula da malha. Uma célula deve fornecer, entre outras, as seguintes funcionalidades:

- Fornecer sua dimensão topológica;
- Verificar a si própria;
- Processar seu contornamento;
- Calcular suas funções de interpolação;
- Retornar um vértice, uma aresta ou uma face específicos;
- Retornar o material da célula;
- Retornar o número de vértices, arestas ou faces da célula.

Para representar uma célula, desenvolveu-se uma hierarquia de classes, cujo topo é representado pela classe abstrata `tCell`, definida parcialmente no Programa 2.3. Essa classe apenas define genericamente o comportamento esperado de uma célula sem, entretanto, implementar a maior parte de suas funcionalidades. Possui os seguintes atributos:

- **`int`** `Number`: número identificador da célula;
- `tMesh*` `Mesh`: ponteiro para a malha à qual pertence;
- **`int`** `NumberOfVertices`: número de vértices da célula;

```
1   class tMeshVertex:  public tVertex
2   {
3      friend class tMesh;
4      friend class tCell;
5      friend class tMeshVertexIterator;
6      friend class tDomainBuilder;
7      public:
8         class tUse;
9
10        tMeshVertex();
11        tMeshVertex(const tMeshVertex&);
12
13        ~tMeshVertex();
14
15        int CountUses() const;
16        tMeshVertex::tUse* GetUses() const;
17        tMesh* GetMesh() const;
18        tNode* GetNode() const;
19
20        bool Moveable;
21        bool BoundaryVertex;
22
23     protected:
24        tMeshVertex::tUse* Uses;
25        tMesh* Mesh;
26
27        void MakeUse(tCell*);
28        void KillUse(tCell*);
29     private:
30        tNode* Node;
31
32        DECLARE_DOUBLE_LIST_ELEMENT(tMeshVertex);
33        DECLARE_SERIALIZABLE(tMeshVertex);
34  }; // tMeshVertex
```

Programa 2.2: Classe `tMeshVertex`.

- `tVertex** Vertices`: lista de referências aos vértices da célula;
- `tMaterial Material`: material componente da célula;
- `tShapeFunction* ShapeFunction`: ponteiro para a função de forma.

A lista de incidências de uma célula é representada por um arranjo de ponteiros para objetos da classe `tVertex`. Diversos métodos fornecidos por `tCell` são virtuais puros, pois a implementação depende de informações relacionas à geometria da célula. Essas implementações só podem ser fornecidas em classes concretas derivadas de `tCell`.

A classe `t2DCell` deriva de `tCell` e implementa alguns dos métodos de sua classe base, mas ainda é uma classe abstrata, pois ainda não possui informações suficientes para implementar todas as funções virtuais puras definidas em `tCell`. Essa classe representa células bidimensionais e, como exemplo de classe concreta que dela deriva, pode-se citar a classe `t3N2DCell` que representa células com formato de triângulo.

A classe `t3DCell`, similarmente a `t2DCell`, também é uma classe abstrata, entretanto, representa células tridimensionais. Como exemplo de uma de suas classes concretas derivadas, pode-se citar a classe `t4N3DCell` que representa células com formato de tetraedros.

A Figura 2.2 apresenta o diagrama de classes relacionadas a `tCell`.

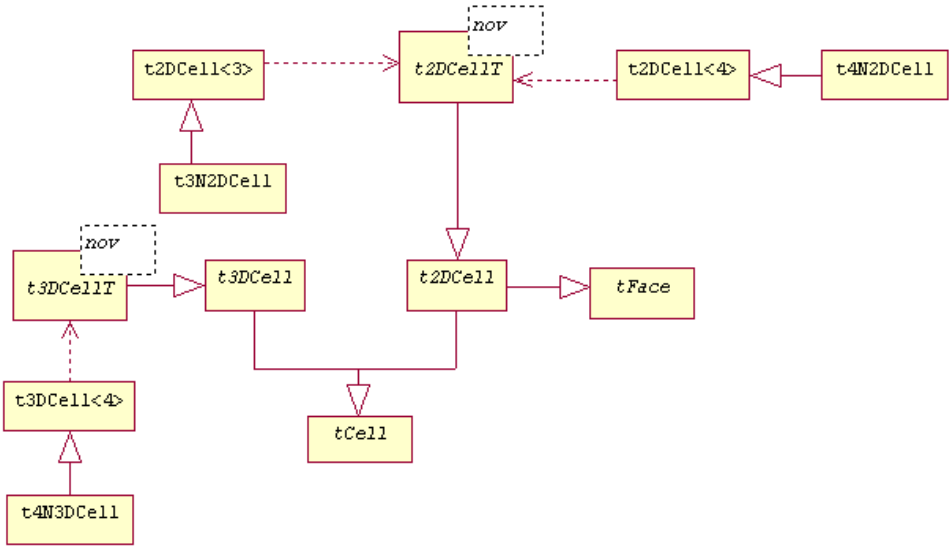


Figura 2.2: Hierarquia de células.

Classe `tFace`

Uma face é representada por um objeto de uma das classes derivadas da classe abstrata `tFace`, descrita no Programa 2.4. Uma face fornece iteradores para seus vértices e arestas através dos métodos `GetVertexIterator()`, linha 4, e `GetEdgeIterator()`, linha 8, respectivamente. O iterador interno que permite percorrer os vértices de uma face é obtido através do método virtual `MakeInternalVertexIterator()`, declarado

```
1   class tCell:  public tSerializable
2   {
3      public:
4         int Number;
5         tMaterial Material;
6
7         tCell();
8         tCell(int);
9
10        virtual tCell* MakeObject() const = 0;
11        virtual ~tCell();
12        virtual intGetDimension() const = 0;
13        virtual int GetNumberOfEdges() const= 0;
14        virtual int GetNumberOfFaces() const= 0;
15        virtual tEdge* GetEdge(int) const = 0;
16        virtual tFace* GetFace(int) const = 0;
17        int GetNumberOfVertices() const;
18        tMeshVertex* GetVertex(int) const;
19        tMesh* GetMesh() const;
20        IElement* GetElement() const;
21        tMeshPartition* GetPartition() const;
22        tMaterial GetMaterial() const;
23        void SetVertex(int, tMeshVertex*);
24        void ReverseOrder();
25        void SetMaterial(const tMaterial&);
26        void SetPartition(tMeshPartition*);
27        virtual void ComputeLocalSystem(tLocalSystem&) const;
28        virtual t3DVector ComputePositionAt(double []) const;
29        virtual tVector ComputeShapeFunctionsAt(double []) const = 0;
30        virtual tMatrix ComputeShapeFunctionDerivativesAt(double [])
31           const = 0;
32        virtual tJacobianMatrix ComputeJacobianMatrix() const
33
34     protected:
35        tMeshPartition* Partition;
36        tMesh* Mesh;
37        int NumberOfNodes;
38        tMeshVertex** Vertices;
39     private:
40        IElement* Element;
41
42        DECLARE_DOUBLE_LIST_ELEMENT(tCell);
43        DECLARE_ABSTRACT_SERIALIZABLE(tCell);
44        ...
45  }; // tCell
```

Programa 2.3: Classe abstrata `tCell`.

na linha 18. Similarmente, o iterador interno para arestas da face é obtido através do método `MakeInternalEdgeIterator()`, linha 19. Outros métodos implementados em `tFace` são:

- `Normal()` (linha 13). Retorna o vetor normal à face;
- `GetMaterial()` (linha 14). Retorna o material que constitui a face;
- `GetNumber()` (linha 15). Retorna o número que identifica a face.

```
1   class tFace
2   {
3      public:
4         tVertexIterator GetVertexIterator()
5         {
6            return tVertexIterator(MakeInternalVertexIterator());
7         }
8         tEdgeIterator GetEdgeIterator()
9         {
10           return tEdgeIterator(MakeInternalEdgeIterator());
11        }
12
13        virtual t3DVector Normal() const = 0;
14        virtual tMaterial GetMaterial() const = 0;
15        virtual int GetNumber() const = 0;
16
17     private:
18        virtual tInternalVertexIterator* MakeInternalVertexIterator();
19        virtual tInternalEdgeIterator* MakeInternalEdgeIterator();
20  }; // tFace
```

Programa 2.4: Classe `tFace`.

2.4 Modelo de Análise

Nesta seção são descritas as diferentes classes usadas na representação de malhas de elementos finitos, de restrições e de carregamentos, todos componentes do modelo de análise de um sólido elástico. Algumas das funcionalidades de tais classes, introduzidas nesta seção, apenas serão compreendidas à luz do material do Capítulo 3, que trata da análise seqüencial através do MEF.

Classe `tDomain`

Um domínio representa a malha acrescida de todas as informações necessárias à fase de análise e é implementado como um objeto da classe `tDomain`, definida no Programa 2.5. O domínio é composto por nós e por elementos — finitos, no caso do MEF —, além de restrições e casos de carregamento.

```
1   class tDomain:virtual public IElement, public tSharedObject
2   {
3      public:
4         typedef tPointerArray<IElement> tElementArray;
5         typedef tPointerArray<tNode> tNodeArray;
6         typedef tPointerArray<tSPConstraint> tSPConstraintArray;
7         typedef tPointerArray<tLoadPattern> tLoadPatternArray;
8
9         tDomain(tMesh&, int = 1, int = 1);
10        virtual ~tDomain();
11        int AddElement(IElement*);
12        int AddNode(tNode*);
13        int AddSPConstraint(tSPConstraint*);
14        int AddLoadPattern(tLoadPattern*);
15        int GetNumberOfElements() const;
16        int GetNumberOfNodes() const;
17        int GetNumberOfSPConstraints() const;
18        int GetNumberOfLoadPatterns() const;
19        IElement* RemoveElement(int);
20        tNode* RemoveNode(int);
21        tSPConstraint* RemoveSPConstraint(int);
22        tLoadPattern* RemoveLoadPattern(int);
23        void ClearAll();
24        void ApplyLoad(double);
25        void SetLastResponse(const tVector&);
26        void Update();
27        tGraph* GetElementGraph() const;
28        long GetModifiedTime() const;
29        tMesh* GetMesh() const;
30        long GetNumberOfEquations() const;
31
32        virtual void Modified();
33
34     protected:
35        tElementArray Elements;
36        tNodeArray Nodes;
37        tSPConstraintArray SPConstraints;
38        tLoadPatternArray LoadPatterns;
39     private:
40        long ModifiedTime;
41        tMesh* Mesh;
42        long NumberOfEquations;
43        ...
44  }; // tDomain
```

Programa 2.5: Classe `tDomain`.

O diagrama de classes da Figura 2.3 ilustra a classe `tDomain` e as principais classes relacionadas.

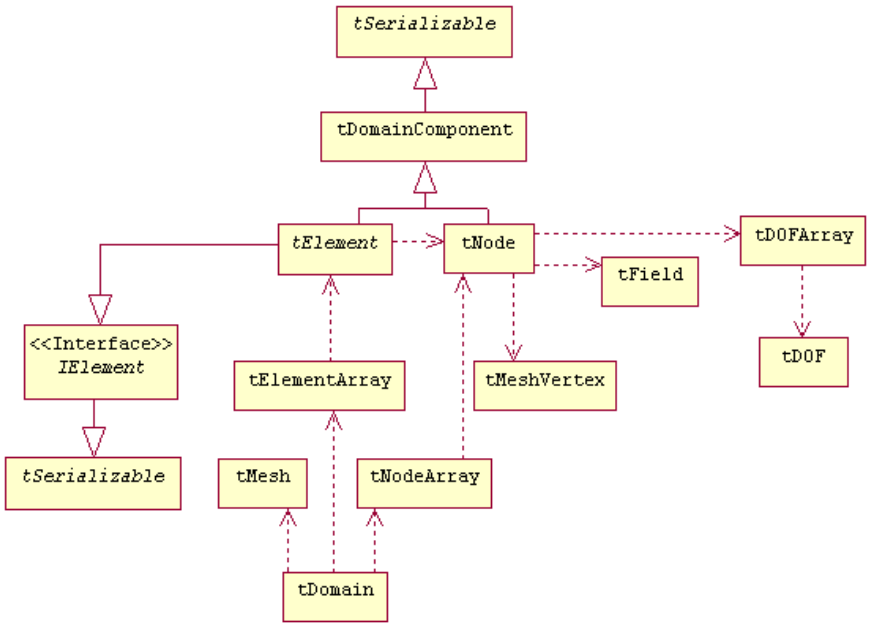


Figura 2.3: Diagrama de classes relacionadas ao domínio.

Classe `tDomainBuilder`

Um objeto da classe `tDomainBuilder` é responsável por construir um novo domínio — objeto de `tDomain` — a partir de uma malha.

Classe `tDomainComponent`

Para representar um componente de domínio genérico, criou-se a classe `tDomain-Component` — derivada da classe `tSerializable` — indicativa da possibilidade de serialização de um componente de domínio. Um objeto dessa classe armazena apenas informações comuns a qualquer componente: o seu número e um objeto da classe `tDomain` que representa o domínio ao qual o componente pertence; essa informação é de grande importância principalmente quando subdomínios são utilizados.

Classe `tNode`

Um nó é um objeto da classe `tNode`, derivada de `tDomainComponent`. Um nó de um domínio contém o vértice correspondente a ele na malha e, além disso, possui um vetor com seus graus de liberdade — representado por um ponteiro para um `tDOFArray`, onde cada DOF é representado por um objeto da classe `tDOF` — e um objeto `Field` — do tipo `tField`. O objeto `Field` serve para armazenar informações específicas de cada tipo de análise como, por exemplo, deslocamentos, forças, acelerações e velocidades. A definição parcial da classe `tNode` é apresentada no Programa 2.6.

Possui os seguintes métodos principais:

```
1   class tNode:public tDomainComponent
2   {
3      public:
4         tNode(int, tMeshVertex*);
5
6         virtual ~tNode();
7
8         tMeshVertex* GetVertex() const;
9         int GetNumberOfDOFs() const;
10        tDOFArray& GetDOFs() const;
11        tUse* GetUses() const;
12        tVector GetLoadVector() const;
13        bool AddLoad(const tVector&, double);
14
15     private:
16        tMeshVertex* Vertex;
17        tDOFArray* DOFs;
18        tVector LoadVector;
19
20        friend class tDomainBuilder;
21        ...
22  }; // tNode
```

Programa 2.6: Classe `tNode`.

- `GetVertex()`: retorna o vértice associado ao nó;
- `GetNumberOfDOFs()`: retorna o número de DOFs do nó;
- `GetDOFs()`: retorna os DOFs associados ao nó;
- `GetUses()`: retorna a lista de usos do nó;
- `GetLoadVector()`: retorna o vetor de carregamentos do nó;
- `AddLoad()`: adiciona um vetor de carregamentos ao nó.

Classe `tElement`

Um elemento é um objeto da classe `tElement`, derivada da classe `tDomainComponent`, e possui como atributos o número de nós que o compõem, um vetor de referências a esses nós e o material de que é composto. A definição da classe `tElement` é apresentada parcialmente no Programa 2.7.

Possui os seguintes métodos:

- `GetNumberOfExternalNodes()`: retorna o número de nós externos do elemento;
- `GetExternalNode()`: retorna o nó externo, da posição indicada pelo parâmetro, do elemento;

```
1    class tElement:virtual public IElement, public tDomainComponent
2    {
3       public:
4          int GetNumberOfExternalNodes() const;
5          tNode* GetExternalNode(int) const;
6          tMaterial GetMaterial() const;

7          void SetMaterial(const tMaterial&);
8
9       protected:
10         int NumberOfNodes;
11         tNode** Nodes;
12         tMaterial Material;
13
14         tElement(int, tNode**);
15
16         void SetExternalNode(int, tNode*);
17      ...
18   }; // tElement
```

Programa 2.7: Classe `tElement`.

- `GetMaterial()`: retorna o material do qual o elemento é composto;
- `SetMaterial()`: define o material do qual o elemento é composto;
- `SetExternalNode()`: adiciona um novo novo nó ao elemento.

Classe `tFiniteElement`

Um objeto da classe `tFiniteElement` representa um elemento finito. A classe é derivada de `tElement` e possui como atributos um objeto da classe `tMatrix`, que representa a matriz tangente, e um objeto de `tVector`, que representa o vetor residual. Possui os seguintes métodos principais:

- `FormTangent(tIntegrator* integrator)`: método que obtém a matriz tangente do elemento por intermédio de um integrador. Recebe como argumento um ponteiro para um integrador, o qual armazena as informações necessárias para auxiliar o elemento a calcular adequadamente a matriz, dependendo do tipo da análise em questão (no caso da análise linear, a matriz tangente é igual a matriz de rigidez do elemento finito, definida no Capítulo 3);
- `GetTangent()`: retorna a matriz tangente, obtida pelo método anterior;
- `FormResidual(tIntegrator* integrator)`: método que obtém o vetor residual do elemento. Assim como `FormaTangent()`, recebe um integrador como parâmetro (no caso da análise linear, o vetor residual é igual ao vetor de esforços nodais equivalentes do elemento finito, definido no Capítulo 3);
- `GetResidual()`: retorna o vetor residual calculado pelo método anterior;

- `GetLocationArray()`: retorna o *array* que define o número da equação de cada grau de liberdade nodal do elemento no sistema de equações da análise;
- `GetLastResponse()`: retorna o valor dos graus de liberdade incógnitos determinados pela análise.

A definição de `tFiniteElement` é apresentada no Programa 2.8.

```
1   class tFiniteElement:virtual public IFiniteElement, public tElement
2   {
3      public:
4         ~tFiniteElement();

5         void FormTangent(tIntegrator*);
6         void FormResidual(tIntegrator*);

7         tMatrix GetTangent() const;
8         tVector GetLoadVector() const;
9         tVector GetResidual() const;
10        tVector GetLastResponse() const;

11        const tLocationArray& GetLocationArray();

12        void Update();
13
14     protected:
15        tFiniteElement(int, tNode**);
16
17     private:
18        tMatrix Tangent;
19        tVector LoadVector;
20        tVector Residual;
21        tLocationArray* LocationArray;
22  }; // tFiniteElement
```

Programa 2.8: Classe `tFiniteElement`.

Classe `tFENode`

A classe `tFENode`, derivada de `tNode` define um nó para análise pelo MEF. Adicionalmente aos métodos de `tNode`, `tFENode` ainda possui métodos para retornar o vetor de carregamentos (`tVector GetLoadVector()`) e para adicionar carregamentos (`AddLoad()`).

O diagrama de classes da Figura 2.4 ilustra o relacionamento das classes `tElement`, `tNode` e suas derivadas.

Classe `tLoad`

A classe abstrata representa um carregamento genérico que pode ser aplicado a nós ou a elementos, representados, respectivamente, por `tNodalLoad` e `tElementalLoad`.

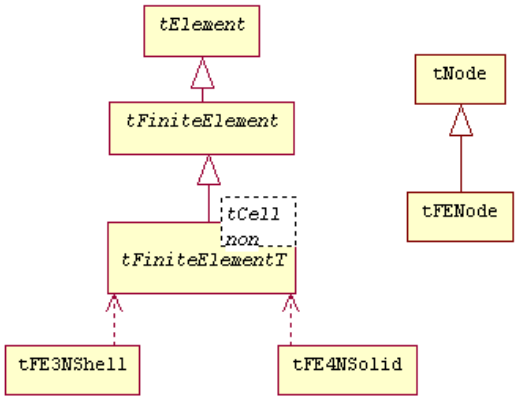


Figura 2.4: Diagrama de classes derivadas de `tElement` e `tNode`.

Classe `tLoadPattern`

Um objeto da classe `tLoadPattern` representa uma agregação de carregamentos aplicados a nós e a elementos de um domínio, no qual cada carregamento é representado por um objeto de uma classe derivada de `tLoad`. Além disso, essa agregação mantém uma função que determina o fator de escala que define a variação dos carregamentos ao longo do tempo.

Classe `tSPConstraint`

A classe `tSPConstraint` representa uma condição de contorno homogênea aplicada a um grau de liberdade de um nó, ou seja, é uma restrição a um DOF.

2.5 Considerações Finais

Neste capítulo foram introduzidas a conceituação fundamental e a formulação dos modelos de elementos finitos. O MEF foi introduzido como um dos métodos mais importantes na análise estrutural e foram salientadas algumas de suas diferenças com relação a outros métodos. A implementação do modelo desenvolvido no presente trabalho foi descrita juntamente com diagramas de classes e descrições das principais classes. Uma das principais características dessa implementação é a separação entre malha (`tMesh`) e domínio (`tDomain`); enquanto a primeira representa a geometria do sólido analisado, o segundo armazena informações mais específicas para a fase de análise.

Com base nos conceitos apresentados, torna-se imediato o interesse por alternativas para tornar o processo de análise através do MEF mais eficaz, tema dos dois capítulos subseqüentes. No Capítulo 3 são mostradas algumas implementações seqüenciais do MEF, com destaque à implementada neste trabalho. O Capítulo 5, por sua vez, aborda alternativas paralelas para implementação do mesmo, abrangendo uma revisão bibliográfica sobre alguns trabalhos relacionados e elencando algumas técnicas utilizadas, em especial, a decomposição de domínio, tema do Capítulo 4.

CAPÍTULO 3

Análise Seqüencial

3.1 Considerações Iniciais

De acordo com Oden [ODE87], provavelmente, nenhuma outra família de métodos de aproximação teve maior impacto, teórico e prático, do que os elementos finitos. Os métodos de elementos finitos têm sido utilizados em quase todas as áreas da ciência e engenharia que envolvem modelos caracterizados por equações diferenciais parciais. Ainda de acordo com este autor, tal popularidade deve-se, em parte, ao fato de que devido às características desses métodos é possível quebrar um domínio qualquer em um conjunto arbitrário de subdomínios disjuntos. Dessa forma, a análise pode ser feita localmente em cada subdomínio, o qual pode ser suficientemente pequeno para que se possa representar o seu comportamento local através de funções polinomiais.

Este capítulo aborda a análise seqüencial implementada no framework. Na Seção 3.2 são apresentados resumos do modelo matemático usado na análise de sólidos elásticos e da formulação do MEF para resolução seqüencial do modelo. A Seção 3.3 descreve o esquema computacional utilizado para o MEF seqüencial. Os diagramas de classes e as funcionalidades das principais classes dos componentes do framework associadas à análise são introduzidos na Seção 3.4.

3.2 Fundamentos

Esta seção mostra um resumo do modelo matemático da elasticidade tridimensional e da formulação do MEF aplicado à análise linear estática de sólidos elásticos, esta última derivada do método mais geral de resíduos ponderados. Um resumo mais abrangente desses tópicos pode ser encontrado em [PAG98].

3.2.1 Modelo Matemático da Elasticidade Linear

Considere um sólido definido por um domínio Ω e um contorno Γ, no qual são aplicados forças de volume e de superfície. Pode-se expressar o equilíbrio estático em cada ponto

do sólido, em termos de tensões e forças aplicadas ao volume, em notação indicial, através da seguinte equação diferencial [MAL69]:

$$\sigma_{ji,j} + b_i = 0, \tag{3.1}$$

onde σ_{ji} representa o *tensor de tensões* e b_i representa a *força de volume* por unidade de volume. Se o sólido apresentar pequenos deslocamentos e deformações, pode-se escrever

$$\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}), \tag{3.2}$$

onde ϵ_{ij} representa o *tensor de* (pequenas) *deformações* e u_i é o campo de *deslocamentos*. Se for assumido que o material do sólido é homogêneo e perfeitamente elástico, sua *relação constitutiva* é expressa pela *Lei de Hooke*

$$\sigma_{ij} = C_{ijlm}\epsilon_{lm}. \tag{3.3}$$

Para materiais isotrópicos, o *tensor de módulos elásticos* C_{ijlm} é dado por

$$C_{ijlm} = \lambda\delta_{ij}\delta_{lm} + \mu(\delta_{il}\delta_{jm} + \delta_{im}\delta_{jl}), \tag{3.4}$$

onde λ e μ são as *constantes de Lame* e δ_{ij} é o delta de Kronecker. Substituindo a Equação (3.4) na Equação (3.3), a versão isotrópica da Lei de Hooke fica

$$\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}. \tag{3.5}$$

Pode-se relacionar as constantes de Lame ao *módulo de elasticidade transversal* E, *coeficiente de Poisson* ν e *módulo de elasticidade transversal* G na forma

$$G = \mu = \frac{E}{2(1+\nu)} \qquad \text{e} \qquad \lambda = \frac{\nu\, E}{(1+\nu)(1-\nu)}. \tag{3.6}$$

Considerando a Equação (3.6) e substituindo a Equação (3.2) na Equação (3.5) e a equação resultante na Equação (3.1), obtém-se

$$G\ u_{i,jj} + \frac{G}{1-2\nu}\ u_{j,ij} + b_i = 0. \tag{3.7}$$

A Equação (3.7) é conhecida como *equação de Navier-Cauchy* da elasticidade. Tal equação expressa o equilíbrio estático de um sólido em função do campo de deslocamentos u_i e da força de volume b_i aplicadas ao sólido.

Para compreender totalmente a Equação (3.7) são necessários conhecimentos sobre cálculo tensorial e mecânica do contínuo que estão além do escopo deste texto. Portanto, sem um maior aprofundamento, enuncia-se que a unicidade da solução da Equação (3.7) deve satisfazer duas espécies de *condições de contorno* [BRE84]: deslocamentos prescritos (condições de contorno essenciais)

$$u_i = \overline{u}_i \qquad \text{em } \Gamma_1 \tag{3.8}$$

e forças de superfície prescritas (condições de contorno naturais)

$$p_i = \sigma_{ji}n_j = \overline{p}_i \qquad \text{em } \Gamma_2, \tag{3.9}$$

onde $\Gamma = \Gamma_1 + \Gamma_2$ é a superfície do contorno do sólido e n_j é a normal externa a Γ_2.

A Equação (3.1), derivada da teoria da mecânica do contínuo, admitidas as hipóteses simplificadoras listadas anteriormente, juntamente com as condições de contorno dadas pela Equação (3.8) e pela Equação (3.9), caracteriza o modelo matemático de um sólido elástico.

3.2.2 Resíduos Ponderados

Conforme dito no Capítulo 1, a solução do modelo matemático apresentado anteriormente só pode ser aproximadamente determinada, para os casos gerais de geometria e condições de contorno, através de métodos computacionais numéricos como o MEF e o MEC. As formulações para ambos os métodos podem ser derivadas de um método mais geral denominado *método dos resíduos ponderados*.

Em linhas gerais, o método consiste na adoção de uma *função aproximadora* satisfazendo ou não as condições de contorno do modelo matemático. A substituição da função aproximadora na equação diferencial do modelo, portanto, produz um *resíduo* que deve, de alguma forma, ser *ponderado* a fim de tornar-se próximo de zero se considerado em todo o domínio do problema. Isso é feito adotando-se uma segunda função, chamada *função ponderadora*. Usualmente, um resíduo $\mathbf{R}$ é distribuído em um domínio Ω tomando-se o *produto interno*

$$\int_{\Omega} \mathbf{R}\,\mathbf{w}\, d\Omega = 0, \tag{3.10}$$

onde $\mathbf{w}$ é a função ponderadora dessa distribuição. Observe que o produto interno transforma o resíduo de uma equação diferencial do modelo matemático em uma *equação integral* sobre o domínio Ω do problema em questão.

Para obtenção das expressões de resíduos ponderados para o problema da elasticidade linear tridimensional, será utilizada, ao invés da equação de Navier-Cauchy, a equação de equilíbrio estático em sua forma indicial,

$$\sigma_{jk,j} + b_k = 0 \qquad \text{em } \Omega, \tag{3.1-repetida}$$

juntamente com as condições de contorno

$$u_k = \overline{u}_k \qquad \text{em } \Gamma_1, \text{ e} \tag{3.11}$$

$$p_k = \overline{p}_k \qquad \text{em } \Gamma_2. \tag{3.12}$$

Seja o campo de deslocamentos $\mathbf{u}_0$ a solução exata do problema, a qual será aproximada no domínio Ω por uma função $\mathbf{u}$ (representada na forma indicial por u_k). Admite-se que a função aproximadora do campo de deslocamentos $\mathbf{u}$ satisfaz as condições de contorno essenciais e naturais do problema, não satisfazendo, entretanto, a Equação (3.1). Dessa forma, tem-se um resíduo

$$\sigma_{jk,j} + b_k \neq 0 \qquad \text{em } \Omega, \tag{3.13}$$

o qual pode ser distribuído pelo domínio com o produto interno

$$\int_{\Omega} (\sigma_{jk,j} + b_k)\, u_k^*\, d\Omega = 0, \tag{3.14}$$

onde a função ponderadora $\mathbf{w} = \mathbf{u}^*$ representa um campo de deslocamentos qualquer sobre um domínio Ω^* arbitrário. Assume-se que os gradientes do campo de deslocamentos ponderador $\mathbf{u}^*$ sejam pequenos em relação à unidade (ou seja, as relações deslocamento-deformação são válidas para $\mathbf{u}^*$) e que as equações constitutivas elásticas sejam satisfeitas em Ω^*.

Para a obtenção de uma expressão que relacione não somente os erros de domínio, mas também os resíduos no contorno, como usual em resíduos ponderados, considera-se a *forma transposta* da Equação (3.14) [BRE84]

$$\int_{\Omega} \sigma^*_{jk,j}\, u_k \, d\Omega + \int_{\Omega} b_k \, u^*_k \, d\Omega = -\int_{\Gamma} p_k \, u^*_k \, d\Gamma + \int_{\Gamma} u_k \, p^*_k \, d\Gamma, \tag{3.15}$$

onde σ^*_{jk} é o campo de tensões associado à função ponderadora u^*_k, a qual, agora, aproxima também as condições de contorno essencias e naturais. Após a substituição das condições de contorno (3.11) e (3.12) na Equação (3.15) e manipulações matemáticas adequadas, obtém-se [BRE84]

$$\int_{\Omega} (\sigma_{jk,j} + b_k)\, u^*_k \, d\Omega = \int_{\Gamma_2} (p_k - \overline{p}_k)\, u^*_k \, d\Gamma - \int_{\Gamma_1} (u_k - \overline{u}_k)\, p^*_k \, d\Gamma. \tag{3.16}$$

A Equação (3.16) é a *sentença original* de resíduos ponderados para o problema elastostático, supondo que a função $\mathbf{u}$ aproxima a equação diferencial em Ω e as condições de contorno em Λ.

3.2.3 Formulação do MEF

A formulação do MEF para a elasticidade linear pode ser obtida a partir da *forma fraca* [BRE84] da sentença original de resíduos ponderados, Equação (3.16), considerando-se uma função aproximadora u_k que satisfaz as condições de contorno essenciais, ou seja, $u_k \equiv \overline{u}_k$ em Γ_1. Com isso, pode-se reescrever a Equação (3.16), após dupla integração por partes, como

$$\int_{\Omega} \sigma_{jk}\, \epsilon^*_{jk} \, d\Omega - \int_{\Omega} b_k \, u^*_k \, d\Omega = \int_{\Gamma_2} \overline{p}_k \, u^*_k \, d\Gamma + \int_{\Gamma_1} p_k \, u^*_k \, d\Gamma. \tag{3.17}$$

Considera-se, agora, uma função ponderadora $u^*_k = \delta u_k$, na qual a variação δu_k é um campo de deslocamentos infinitesimais, definido a partir do mesmo conjunto de funções linearmente independentes utilizado na definição de u_k. Supondo que esse campo de deslocamentos satisfaz a versão homogênea das condições de contorno essenciais, ou seja $u^*_k = \delta u_k \equiv 0$ em Γ_1, pode-se reduzir a Equação (3.17) a

$$\int_{\Omega} \sigma_{jk}\, \delta\epsilon_{jk} \, d\Omega = \int_{\Omega} b_k \, \delta u_k \, d\Omega + \int_{\Gamma_2} \overline{p}_k \, \delta u_k \, d\Gamma. \tag{3.18}$$

Essa equação é a expressão do *princípio dos trabalhos virtuais* (PTV). Considerando-se os deslocamentos $\delta\mathbf{u}$ como deslocamentos virtuais aplicados a uma configuração de equilíbrio de um corpo, pode-se interpretar o lado direito da Equação (3.18) como o trabalho externo δW_{ext} realizado pelas forças de volume e de superfície atuantes no domínio e no contorno do corpo durante os deslocamentos imaginários $\delta\mathbf{u}$. O lado esquerdo da equação pode ser interpretado como a energia de deformação δU armazenada no corpo durante os deslocamentos imaginários. Na configuração de equilíbrio, $\delta U = \delta W_{\text{ext}}$, a função ponderadora $\delta\mathbf{u}$ deve satisfazer a versão homogênea das condições de contorno essenciais, dado que os deslocamentos virtuais devem ser cinematicamente

admissíveis, ou seja, devem satisfazer quaisquer deslocamentos prescritos. Desde que os deslocamentos virtuais são considerados como deslocamentos adicionais impostos a uma configuração de equilíbrio de um corpo, um componente de deslocamento virtual deve ser nulo sempre que o deslocamento atual for determinado por uma condição de contorno.

O equilíbrio de um sólido pode ser matematicamente expresso pelo princípio dos trabalhos virtuais, Equação (3.18), convenientemente escrito na forma matricial como

$$\int_{\Omega} [\delta\boldsymbol{\epsilon}]^T [\boldsymbol{\sigma}]\, d\Omega = \int_{\Omega} [\delta\mathbf{u}]^T \mathbf{b}\, d\Omega + \int_{\Gamma_2} [\delta\mathbf{u}]^T \mathbf{p}\, d\Gamma, \tag{3.19}$$

na qual $[\boldsymbol{\sigma}]$ e $[\boldsymbol{\epsilon}]$ são, respectivamente, vetores de componentes de tensão e de deformação.

O MEF baseia-se na subdivisão do domínio Ω em um conjunto de células, os elementos finitos, conectadas através de nós. Considera-se um sistema global de coordenadas utilizado para a obtenção das coordenadas cartesianas dos nós. Um elemento finito é uma subregião do domínio Ω, definido pela seqüência de nós sobre os quais o mesmo incide. A essa seqüência dá-se o nome de *lista de incidência do elemento*. Um elemento finito possui um sistema de coordenadas locais ou *normalizadas*.

Com a subdivisão do domínio do sólido em NE elementos finitos, a Equação (3.19) pode ser escrita como somas de integrais sobre o volume e a superfície de cada elemento finito do modelo:

$$\sum_{e=1}^{NE} \left\{ \int_{\Omega^{(e)}} \left[\delta\boldsymbol{\epsilon}^{(e)}\right]^T \left[\boldsymbol{\sigma}^{(e)}\right] d\Omega - \int_{\Omega^{(e)}} \left[\delta\mathbf{u}^{(e)}\right]^T \mathbf{b}^{(e)}\, d\Omega - \int_{\Gamma_2^{(e)}} \left[\delta\mathbf{u}^{(e)}\right]^T \mathbf{p}^{(e)}\, d\Gamma \right\} = 0, \tag{3.20}$$

onde $[\boldsymbol{\sigma}]$ é o vetor de componentes de tensão e $[\boldsymbol{\epsilon}]$ é o vetor de componentes de deformação.

Seja um elemento finito e definido por n nós. Seja q um ponto qualquer de e, cujas coordenadas $\boldsymbol{\xi}$ são tomadas com relação ao sistema de coordenadas normalizadas. O vetor de deslocamento $\mathbf{u}^{(e)}$ de q pode ser aproximado por

$$\mathbf{u}^{(e)} = \sum_{i=1}^{n} \mathbf{N}_i^{(e)} \mathbf{u}_i^{(e)}, \tag{3.21}$$

onde $\mathbf{u}_i^{(e)}$ é o vetor de deslocamentos do nó i, $\mathbf{N}_i^{(e)}$ é uma função da posição ξ associada ao nó i, e $\mathbf{N}^{(e)}$ é a matriz de funções de interpolação do elemento e.

Se todos os componentes do vetor de deslocamentos são interpolados de forma idêntica, tem-se, a partir da Equação (3.21),

$$\mathbf{u}^{(e)} = \sum_{i=1}^{n} N_i^{(e)} \mathbf{u}_i^{(e)}. \tag{3.22}$$

De maneira geral, pode-se utilizar expressões semelhantes à Equação (3.22) para interpolar grandezas mecânicas e geométricas sobre um elemento finito (ou de contorno), de posse dos valores nodais. Se forem utilizadas as mesmas funções para interpolar grandezas mecânicas e geométricas em um elemento, diz-se que o elemento

é isoparamétrico, sendo as funções de interpolação chamadas de *funções de forma do elemento*.

Com base na Equação (3.21), o vetor de componentes de pequenas deformações em um ponto qualquer q de um elemento finito e é dado por

$$\left[\boldsymbol{\epsilon}^{(e)}\right] = \mathbf{B}^{(e)}\,\mathbf{U}^{(e)} = \sum_{i=1}^{n} \mathbf{B}_i^{(e)}\mathbf{u}_i^{(e)}, \tag{3.23}$$

na qual $\mathbf{U}_{(e)}$ é o vetor de deslocamentos de todos os n nós do elemento e, $\mathbf{B}^{(e)}$ é a matriz de deformação-deslocamento de e e $\mathbf{B}_i^{(e)}$ é a matriz deformação-deslocamento associada ao nó i do elemento e, definida, no caso geral tridimensional, como

$$\mathbf{B}_i^{(e)} = \begin{bmatrix} \dfrac{\partial N_i^{(e)}}{\partial x} & 0 & 0 \\ 0 & \dfrac{\partial N_i^{(e)}}{\partial y} & 0 \\ 0 & 0 & \dfrac{\partial N_i^{(e)}}{\partial z} \\ \dfrac{\partial N_i^{(e)}}{\partial y} & \dfrac{\partial N_i^{(e)}}{\partial x} & 0 \\ 0 & \dfrac{\partial N_i^{(e)}}{\partial z} & \dfrac{\partial N_i^{(e)}}{\partial y} \\ \dfrac{\partial N_i^{(e)}}{\partial z} & 0 & \dfrac{\partial N_i^{(e)}}{\partial x} \end{bmatrix}. \tag{3.24}$$

O vetor de componentes de tensão em um ponto qualquer de um elemento finito e, considerando as relações constitutivas elásticas, é dado por

$$\left[\boldsymbol{\sigma}^{(e)}\right] = \mathbf{C}^{(e)}\,\mathbf{B}^{(e)} = \mathbf{C}^{(e)}\left(\sum_{j=1}^{n} \mathbf{B}_j^{(e)}\mathbf{u}_j^{(e)}\right), \tag{3.25}$$

onde $\mathbf{C}^{(e)}$ é a matriz constitutiva do elemento e.

Dessa forma, as versões virtuais do deslocamento e das deformações no ponto q do elemento e podem ser definidas como

$$\delta\mathbf{u}^{(e)} = \mathbf{N}^{(e)}\,\mathbf{U}^{(e)} = \sum_{i=1}^{n} \mathbf{N}_i^{(e)}\,\delta\mathbf{u}_i \quad \text{e} \quad \left[\delta\boldsymbol{\epsilon}^{(e)}\right] = \mathbf{B}^{(e)}\,\delta\mathbf{U}^{(e)} = \sum_{i=1}^{n} \mathbf{B}_i^{(e)}\,\delta\mathbf{u}_i^{(e)}, \tag{3.26}$$

onde $\delta\mathbf{U}^{(e)}$ é o vetor de deslocamentos nodais de todos os nós de e. Substituindo a Equação (3.26) e a Equação (3.25) na Expressão (3.19), e dado que os deslocamentos virtuais $\delta\mathbf{U}^{(e)}$ são arbitrários, obtém-se, para o elemento e,

$$\int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{C}^{(e)}\mathbf{B}^{(e)}\mathbf{U}^{(e)}d\Omega - \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}^{(e)}d\Omega - \int_{\Gamma_2^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{p}^{(e)}d\Gamma = 0. \tag{3.27}$$

A Equação (3.27) pode ser escrita da forma

$$\mathbf{K}^{(e)}\mathbf{U}^{(e)} = \mathbf{F}^{(e)}, \tag{3.28}$$

na qual $\mathbf{K}^{(e)}$ é a *matriz de rigidez* do elemento e e $\mathbf{F}^{(e)}$ é o *vetor de carregamentos nodais equivalentes* do elemento e, definidos, respectivamente, como

$$\mathbf{K}^{(e)} = \int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{C}^{(e)}\mathbf{B}^{(e)} d\Omega, \tag{3.29}$$

$$\mathbf{F}^{(e)} = \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}^{(e)} d\Omega + \int_{\Gamma_2^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{p}^{(e)} d\Gamma. \tag{3.30}$$

A partir dessas definições, tem-se

$$\sum_{e=1}^{\mathrm{NE}} \mathbf{K}^{(e)}\,\mathbf{U}^{(e)} = \sum_{e=1}^{\mathrm{NE}} \mathbf{F}^{(e)}, \tag{3.31}$$

ou

$$\mathbf{K}\,\mathbf{U} = \mathbf{F}, \tag{3.32}$$

na qual $\mathbf{K}$ é a matriz de rigidez global do modelo, $\mathbf{F}$ é o vetor de carregamentos nodais equivalentes do modelo e $\mathbf{U}$ é o vetor de incógnitas nodais de todos os nós do elemento. Os somatórios da Equação (3.31) são efetuados através da adição das contribuições de todos os elementos de forma que $\mathbf{K}$, $\mathbf{U}$ e $\mathbf{F}$ sejam organizados em partições nodais.

Com a solução do sistema linear expresso na Equação (3.32) e com a introdução das condições de contorno, tem-se os deslocamentos incógnitos do problema em questão. Assim, as deformações e tensões em todos os nós do domínio podem ser obtidas. Para obter esses valores em quaisquer outros pontos do domínio, utiliza-se a interpolação. A formulação do método dos elementos finitos aqui apresentada é denominada método dos deslocamentos, pois os deslocamentos nodais são desconhecidos.

3.3 Esquema Computacional

O esquema computacional utilizado para o MEF é composto pelos seguintes passos:

Passo 1 **Discretização do domínio**: o domínio Ω é subdividido em NE elementos finitos sobre os quais é aproximado o campo de deslocamentos com a utilização de funções de interpolação e de parâmetros nodais. Essa etapa é chamada *pré-processamento*;

Passo 2 **Computação das contribuições dos elementos**: para cada elemento finito e, determina-se a matriz de rigidez $\mathbf{K}^{(e)}$ e os carregamentos nodais $\mathbf{F}^{(e)}$;

Passo 3 **Montagem do sistema linear**: montam-se, com base nas contribuições de todos os NE elementos finitos do modelo, o lado esquerdo $\mathbf{K}$ e o lado direito $\mathbf{F}$ do Sistema (3.32);

Passo 4 **Introdução das condições de contorno**: a matriz $\mathbf{K}$, originalmente singular, é transformada em uma matriz regular, considerando-se um número apropriado de condições de contorno essenciais, as quais definem a *vinculação* do sólido;

Passo 5 **Solução do sistema linear**: com a solução do sistema, obtêm-se os deslocamentos incógnitos nos pontos e direções onde esses valores não são prescritos;

Passo 6 **Computação de valores no domínio**: conhecidos os deslocamentos nodais **U** do sólido, é possível determinar os deslocamentos, deformações e tensões em quaisquer pontos do domínio. Essa etapa é chamada de *pós-processamento*.

Uma alternativa para a análise seqüencial é a utilização de *subestruturação* — técnica que considera o domínio subdividido em diversas partes e considera apenas as interfaces para obtenção do sistema global. A subestruturação é utilizada na análise seqüencial, geralmente, quando a memória disponível para análise do domínio inteiro é insuficiente.

3.4 Implementação

Um amplo conjunto de classes foi criado para permitir a análise numérica pelo MEF, desde classes que definem métodos para solução de sistemas lineares até classes para representar as restrições que podem ser impostas ao domínio analisado. Nesta seção são descritas as classes relacionadas à análise seqüencial com ou sem subestruturação.

Classe `tAnalysis`

Todo processo de análise numérica é representado por um objeto cuja classe deriva, direta ou indiretamente, da classe `tAnalysis`, definida no Programa 3.1.

```
1   class tAnalysis
2   {
3     public:
4       class tAlgorithm;
5
6       virtual ~tAnalysis();
7
8     protected:
9       long ModifiedTime;
10
11      tAnalysis(tDomain&, tConstraintHandler&, tDOFNumberer&);
12      tDomain* GetDomain();
13      tConstraintHandler* GetConstraintHandler();
14      tDOFNumberer* GetDOFNumberer();
15
16    private:
17      tDomain* Domain;
18      tConstraintHandler* ConstraintHandler;
19      tDOFNumberer* DOFNumberer;
20  }; // tAnalysis
```

Programa 3.1: Classe `tAnalysis`.

Um objeto do tipo `tAnalysis` é uma agregação de outros objetos dos seguintes tipos:

- `tDomain`: domínio sobre o qual a análise será efetuada. O objeto é uma coleção de elementos, nós, DOFs, casos de carregamento e restrições;
- `tConstraintHandler`: um objeto cuja classe deriva da classe `tConstraintHandler` é responsável por manipular as restrições impostas ao domínio sendo analisado e, como conseqüência, criar os DOFs do domínio;
- `tDOFNumberer`: um objeto cuja classe deriva de `tDOFNumberer` é responsável por mapear os números de equações no sistema de equações da análise para os DOFs do domínio sendo analisado;
- `tAnalysis::tAlgorithm`: um objeto cuja classe deriva de `tAnalysis::tAlgorithm` é responsável por orquestrar os passos executados por um processo de análise.

O diagrama UML da Figura 3.1 mostra `tAnalysis` e seus principais relacionamentos. A classe declara como atributos ponteiros para o `tDomain` a ser analisado e para o `tConstraintHandler` e `tDOFNumberer` usados no processo de análise. Referências para os demais componentes da agregação (específicos para cada tipo de análise) são declaradas em classes derivadas de `tAnalysis`. As principais são:

- `tStaticAnalysis`: classe base de todo processo de análise estática;
- `tTransientAnalysis`: classe base de todo processo de análise dinâmica;
- `tDomainDecompositionAnalysis`: classe base de toda análise baseada em decomposição de domínio (por exemplo, `tSubstructuringAnalysis`).

Não há quaisquer métodos de análise declarados em `tAnalysis` (a computação numérica é executada pelos componentes da agregação). Mesmo o disparo do processo de análise é efetuado por métodos das classes derivadas de `tStaticAnalysis` e `tTransientAnalysis`.

Classe `tStaticAnalysis`

Um objeto da classe abstrata `tStaticAnalysis`, definida no Programa 3.2, é um contêiner de componentes responsáveis pela análise estática de um domínio. Além dos herdados de `tAnalysis`, a classe declara como atributos ponteiros para as seguintes classes de objetos:

- `tStaticEquilibriumAlgorithm`: derivada de `tAnalysis::tAlgorithm`, encapsula a funcionalidade comum dos algoritmos incrementais de análise numérica baseada em equilíbrio estático;
- `tStaticIntegrator`: derivada de `tIncrementalIntegrator` (a qual deriva, por sua vez, de `tAnalysis::tIntegrator`), implementa métodos para formação do sistema de equações lineares da análise estática, a partir das contribuições dos elementos e DOFs do domínio sendo analisado;

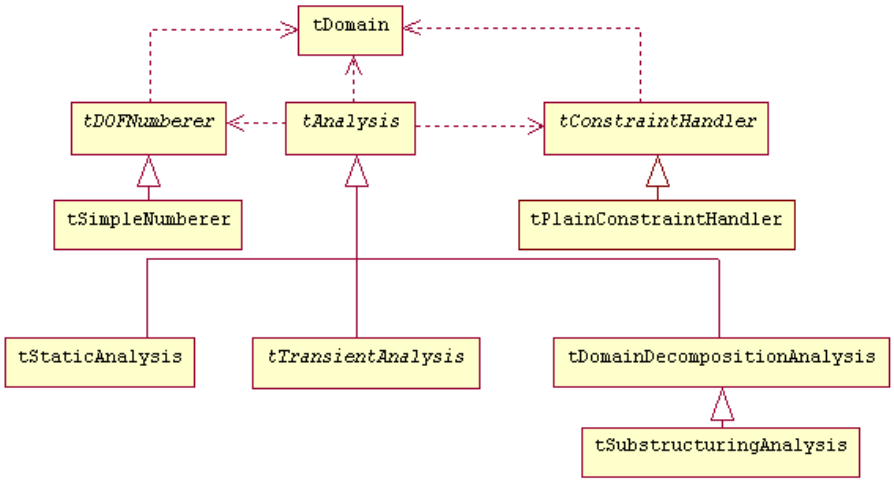


Figura 3.1: Classes de análise.

```
1   class tStaticAnalysis:public tAnalysis
2   {
3      public:
4         tStaticAnalysis(tDomain&, tConstraintHandler&,
5            tDOFNumberer&, tStaticEquilibriumAlgorithm&,
6            tLinearSOE&, tStaticIntegrator&);
7         ~tStaticAnalysis();
8
9         virtual bool Execute(int = 1);
10        virtual bool Update();
11
12     private:
13        tStaticEquilibriumAlgorithm* Algorithm;
14        tLinearSOE* LinearSOE;
15        tStaticIntegrator* Integrator;
16  }; // tStaticAnalysis
```

Programa 3.2: Classe `tStaticAnalysis`.

- `tLinearSOE`: derivada de `tLinearSystemOfEquations`, é classe base de todo sistema de equações lineares.

Os métodos públicos da classe são:

- `Execute(int numberOfSteps)`: responsável por disparar a análise. `numberOfSteps` é o número de passos da análise (o valor *default* é 1, o que significa análise linear). O código desse método é apresentado no Programa 3.3;
- `Update()`: responsável por invocar os métodos apropriados dos objetos da agregação sempre que o domínio sendo analisado for modificado.

A Figura 3.2 ilustra as classes relacionadas à análise estática.

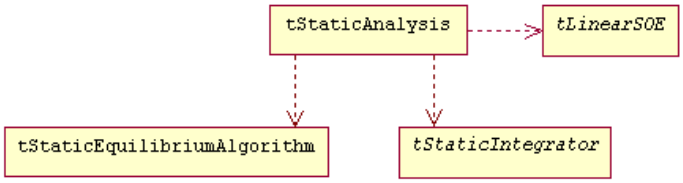


Figura 3.2: Classes de análise estática.

Classe `tAnalysis::tAlgorithm`

Um objeto de uma classe derivada da classe abstrata `tAnalysis::tAlgorithm` é responsável por orquestrar os passos executados por um processo específico de análise. Essa classe fornece dois métodos:

- **`bool`** `SolveCurrentStep()`: esse método é responsável por executar a análise equivalente ao passo de tempo atual;
- **`bool`** `Update()`: método virtual responsável por quaisquer alterações a serem efetuadas no algoritmo, caso haja alterações no domínio.

As ações executadas por cada um desses métodos depende do tipo de algoritmo utilizado, ou seja, depende da classe concreta que os implementa. A classe `tStaticEquilibriumAlgorithm` deriva de `tAlgorithm` e representa os algoritmos para solução de problemas estáticos sem decomposição de domínio e possui duas classes que dela derivam: `tLinearAlgorithm` e `tNewtonRaphsonAlgorithm`. A classe `tDomainDecompositionAlgorithm` também deriva de `tAlgorithm` e implementa o algoritmo usado para resolver problemas que utilizam a decomposição de domínio.

Classe `tLinearAlgorithm`

A classe `tLinearAlgorithm` é uma classe concreta derivada de `tStaticEquilibriumAlgorithm`. Um objeto dessa classe especifica os passos necessários para resolução de sistemas lineares, ele é responsável por invocar métodos de um objeto da classe `tLinearSOESolver` para resolver o sistema de equações lineares. O Programa 3.4 apresenta a definição da classe `tLinearAlgorithm`.

O único método definido nesta classe é `SolveCurrentStep()` herdado de suas classes bases e cujo código é apresentado no Programa 3.5.

Classe `tNewtonRaphsonAlgorithm`

Um objeto da classe `tNewtonRaphsonAlgorithm`, derivada de `tStaticEquilibriumAlgorithm`, é utilizado quando o problema é não-linear, o que gera um sistema de equações não-lineares, geralmente resolvidos por esquemas iterativos.

As classes relacionadas à análise seqüencial e seus relacionamentos são ilustrados no diagrama de classes da Figura 3.3.

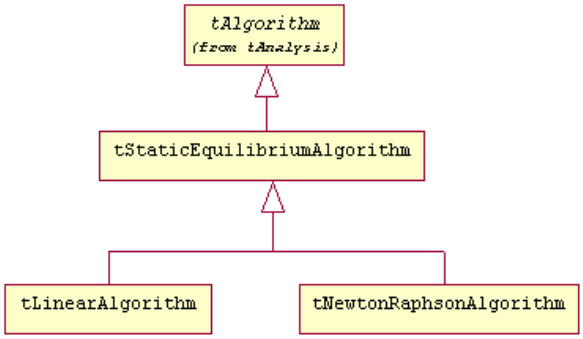


Figura 3.3: Classes de algoritmos para análise seqüencial.

Classe `tIntegrator`

Um objeto de uma classe derivada da classe abstrata `tIntegrator` é responsável por definir as contribuições dos elementos e DOFs do domínio sendo analisado para o sistema de equações. A classe abstrata `tIncrementalIntegrator`, definida no Programa 3.6, deriva de `tIntegrator` e especifica as ações realizadas para a solução incremental de problemas estáticos e dinâmicos. Essa classe possui dois atributos: um ponteiro para o domínio sobre o qual o integrador deve atuar (`tDomain* Domain`) e um ponteiro para o sistema de equações lineares equivalente ao problema (`tLinearSOE* LinearSOE`). Possui os seguintes métodos:

- `FormTangent()`: método que efetivamente calcula a matriz de rigidez solicitada pela chamada ao método `FormTangent()` do elemento finito;
- `FormUnbalance()`: método que forma o vetor de forças aplicadas ao domínio;
- `FormElementResidual()`: método que forma o vetor de forças aplicadas a um elemento;
- `FormNodalUnbalance()`: método que forma o vetor de forças aplicadas a um nó;
- `Commit()`: método que, ao final de um passo da análise, confirma as alterações efetuadas, no caso específico da análise linear ele não faz nada;
- `Update()`: método responsável pela realização de quaisquer alterações necessárias no integrador, caso o domínio tenha sido alterado.

As classes `tStaticIntegrator` e `tTransientIntegrator` são duas classes abstratas derivadas de `tIncrementalIntegrator` que definem integradores para análise estática e dinâmica, respectivamente. As diferentes classes de integradores são ilustradas na Figura 3.4

Classe `tLinearStaticIntegrator`

`tLinearStaticIntegrator` é uma classe concreta derivada de `tStaticIntegrator`. Essa classe possui os seguintes métodos:

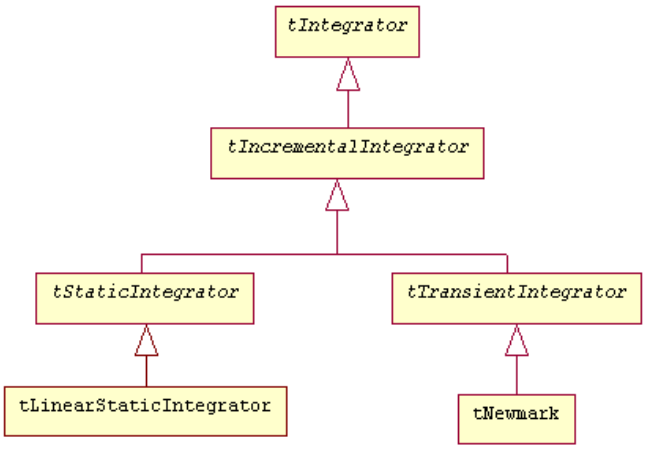


Figura 3.4: Classes de integradores.

- `NewStep()`: herdado de `tStaticIntegrator`. Calcula a solução do próximo passo da simulação. No caso da análise estática, esse passo é único;
- `UpdateResponse(const tVector& u)`: responsável por atualizar os valores armazenados nos graus de liberdade dos nós do domínio.

Classe `tNewmark`

A classe `tNewmark` é concreta e deriva de `tTransientIntegrator`. Essa classe fornece implementações para os métodos herdados de tal forma que possa resolver cada passo de tempo de um problema dinâmico.

Classe `tSystemOfEquations`

A classe abstrata `tSystemOfEquations` representa um sistema de equações genérico. Possui apenas dois métodos:

- `Solve()`: método virtual puro para a resolução do sistema de equações;
- `SetSize(domain)`: determina o tamanho do sistema de equações em termos do números de graus de liberdade de `domain`.

Classe `tLinearSOE`

A classe abstrata `tLinearSOE` — derivada de `tSystemOfEquations` — representa um sistema de equações lineares e possui como atributo uma referência a um objeto da classe `tLinearSOESolver`, que soluciona o sistema de equações lineares. Além dos métodos herdados, possui os seguintes:

- `GetX()`: retorna o vetor de incógnitas do sistema;
- `ZeroA()`: atribui zero a todos os elementos da matriz `A` de coeficientes;

- `ZeroB()`: atribui zero a todos os elementos do vetor `B` de termos independentes;
- `AddA(K, la)`: método responsável por adicionar a matriz de rigidez `K` à matriz de coeficientes nas linhas e colunas especificadas pelo `tLocationArray la` passado como parâmetro;
- `AddB(F, la)`: método responsável por adicionar o vetor de esforços `F` ao vetor de termos independentes nas linhas especificadas pelo parâmetro `tLocationArray la`.

Como essa classe é abstrata, é necessário que sejam implementadas classes concretas derivadas de `tLinearSOE` que implementem suas funcionalidades, é esse o caso de `tFullLinearSOE` e `tBandLinearSOE` que representam, respectivamente, sistemas lineares cheios e em banda e apenas redefinem o método `Solve()`.

As classes relacionadas à representação de sistemas de equações são ilustradas na Figura 3.5.

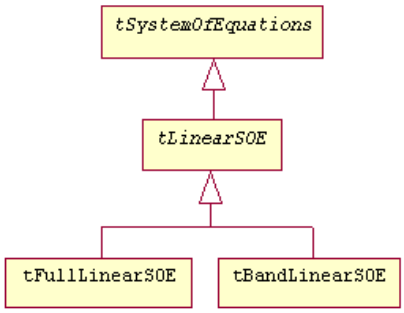


Figura 3.5: Classes de sistemas de equações.

Classe `tSOESolver`

`tSOESolver` é uma classe abstrata que define os métodos que devem ser implementados por uma classe qualquer para solução de sistemas de equações. Possui um único método `Solve()`, que resolve o sistema de equações.

Classe `tLinearSOESolver`

A classe `tLinearSOESolver` é abstrata e derivada de `tSOESolver` e define a interface básica de classes para solução de sistemas de equações lineares. Similarmente à classe `tLinearSOE`, existem duas classes concretas derivadas de `tLinearSOESolver`: `tFullLinearSOESolver` e `tBandLinearSOESolver` que definem, respectivamente, como são resolvidos sistemas cheios e em banda. As classes de solução de sistemas de equações são ilustradas na Figura 3.6.

Classe `tDOFNumberer`

`tDOFNumberer` é uma classe abstrata responsável por definir os métodos necessários para percorrer os nós do domínio e renumerar seus DOFs com a finalidade de diminuir

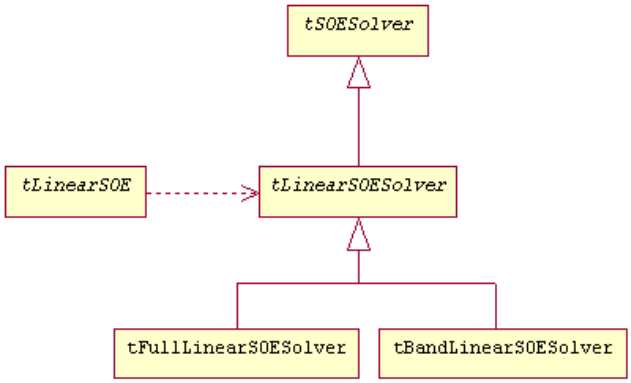


Figura 3.6: Classes para solução de sistemas de equações.

a largura de banda do sistema de equações. Possui um ponteiro para o domínio a ser renumerado como atributo. A renumeração é realizada por um objeto de uma classe concreta derivada de `tDOFNumberer` como, por exemplo, `tSimpleDOFNumbererer`.

Classe `tSimpleDOFNumberer`

Um objeto da classe `tSimpleDOFNumberer`, derivada de `tDOFNumberer`, percorre e renumera os DOFs de um domínio. Esse percurso é feito de forma a sempre fornecer números próximos a nós vizinhos, ou seja, conectados por uma aresta.

Classe `tConstraintHandler`

A classe abstrata `tConstraintHandler` representa um manipulador de restrições genérico. Um objeto de uma classe derivada como, por exemplo, `tPlainConstraint-Handler` é responsável por numerar inicialmente os graus de liberdade dos nós do domínio.

3.5 Execução

A classe `tFEMApp` encapsula as funcionalidades para executar uma aplicação de análise pelo MEF. O método `Run()` (Programa 3.7) é o responsável por iniciar a análise através da chamada ao método `PerformAnalysis()` (Programa 3.8) e, de acordo com os parâmetros recebidos, fazer o particionamento da entrada, se necessário, para o uso da subestruturação que, neste caso, é feito pelo método `PartitionDomain()`. É um objeto dessa classe que define o tipo de análise, o algoritmo de análise a ser executado, o método de resolução de sistemas de equações, o algoritmo de particionamento utilizado, o integrador, entre outras tarefas. As linhas 9-11 criam um objeto da classe `tStaticAnalysis`. A estrutura`tAnalysisDesc` possui um conjunto de *strings* que indicam o tipo de análise e os tipos de componentes a serem utilizados na análise: tipo do manipulador de restrições (padrão é `tPlainConstraintHandler`), o tipo de numerador de DOFs (padrão é `tSimpleDOFNumberer`), o tipo de algoritmo (padrão é `tLinearAlgorithm`), o tipo do sistema de equações (padrão é `tBandLinearSystem`) e o tipo do integrador (padrão é `tLinearStaticIntegrator`). Na linha 10, o método

`tAnalysisFactory::New()` cria a análise estática para `Domain` com os componentes especificados em `ad`. A função principal do programa é apresentada no Programa 3.9 e apenas retorna uma chamada ao método `Run` de um objeto de `tFEMApp`.

3.6 Considerações Finais

Neste capítulo descreveu-se como a análise seqüencial através do método dos elementos finitos é realizada. Apresentou-se uma pequena fundamentação teórica com algumas equações básicas para utilização do MEF. As classes implementadas foram descritas e os relacionamentos entre elas foi ilustrado através de diagramas de classes.

Ao longo do capítulo, expôs-se a possibilidade de realizar a análise seqüencial em duas abordagens. Na primeira, considera-se o domínio completo, os resultados são obtidos em cada nó do domínio; a segunda utiliza a subestruturação, técnica que considera o domínio do problema particionado em um determinado número de subdomínios e, dessa forma, a solução do problema fica restrita aos nós de interface. Essa segunda alternativa é abordada no Capítulo 4 e é bastante indicada quando há limitações quanto à memória disponível para armazenamento do domínio completo, visto que, em muitos casos, tal domínio pode conter milhões de elementos finitos.

A técnica da subestruturação, como dito, exige que o domínio seja particionado em subdomínios, logo, alguma técnica para particionamento do domínio deve ser utilizada. Esse particionamento não pode ser realizado de qualquer maneira, devendo satisfazer determinados critérios; dado que essa técnica tenciona diminuir o tamanho do problema, o ideal é que a extensão da interface entre os diferentes subdomínios seja a menor possível, restrição que inclui uma dificuldade adicional a essa tarefa.

Algumas características dessa segunda abordagem de análise seqüencial possuem características que sugerem a utilização de paralelismo na análise. De fato, para realizar análises paralelas pelo MEF, precisa-se dividir o domínio do problema entre os diferentes processadores, para tal podem ser aproveitadas as mesmas técnicas usadas na análise seqüencial. Além disso, a subestruturação também pode ser utilizada para obter a solução do problema, com a vantagem de que com o uso de múltiplos processadores, as análises em cada subdomínio podem ser feitas paralelamente, o que possibilita uma redução no tempo de processamento. A utilização da decomposição de domínio e da subestruturação na análise paralela de sólidos é o assunto do Capítulo 5.

```
1   bool tStaticAnalysis::Execute(int numberOfSteps)
2   {
3      tDomain* domain = GetDomain();
4
5      if (domain->GetNumberOfElements() == 0)
6      {
7         puts("Domain is empty");
8         return false;
9      }
10     BEGIN_TASK_GROUP("Executing analysis")
11        for (int i = 0; i < numberOfSteps; i++)
12        {
13           long modifiedTime = domain->GetModifiedTime();
14
15           if (modifiedTime != ModifiedTime)
16           {
17              ModifiedTime = modifiedTime;
18              BEGIN_TASK_GROUP("Updating analysis")
19                 if (!Update())
20                    return false;
21                 END_TASK_GROUP
22           }
23           BEGIN_TASK_GROUP("Creating a new current step")
24              if (!Integrator->NewStep())
25                 return false;
26           END_TASK_GROUP
27           BEGIN_TASK_GROUP("Solving the current step")
28              if (!Algorithm->SolveCurrentStep())
29                 return false;
30           END_TASK_GROUP
31           BEGIN_TASK("Commiting the current step")
32              if (!Integrator->Commit())
33              {
34                 Failed();
35                 return false;
36              }
37           END_TASK
38        }
39     END_TASK_GROUP
40     return true;
41  }
```

Programa 3.3: Método `tStaticAnalysis::Execute()`.

```
1   class tLinearAlgorithm:public tStaticEquilibriumAlgorithm
2   {
3     public:
4       bool SolveCurrentStep();
5   }; // tLinearAlgorithm
```

Programa 3.4: Classe `tLinearAlgorithm`.

```
1   bool tLinearAlgorithm::SolveCurrentStep()
2   {
3     tDomain* domain = GetDomain();
4     tIncrementalIntegrator* integrator = GetIntegrator();
5     tLinearSOE* linearSOE = GetLinearSOE();
6
7     if (domain == 0 || integrator == 0 || linearSOE == 0)
8     {
9       puts("**Algorithm:  undefined component(s)");
10      return false;
11    }
12    BEGIN_TASK_GROUP("Forming tangent")
13      if (!integrator->FormTangent())
14        return false;
15    END_TASK_GROUP
16    BEGIN_TASK_GROUP("Forming unbalance")
17      if (!integrator->FormUnbalance())
18        return false;
19    END_TASK_GROUP
20    BEGIN_TASK("Solving linear system")
21      if (!linearSOE->Solve())
22      {
23        Failed();
24        return false;
25      }
26    END_TASK
27    BEGIN_TASK_GROUP("Updating response")
28      if (!integrator->UpdateResponse(linearSOE->GetX()))
29        return false;
30    END_TASK_GROUP
31    return true;
32  }
```

Programa 3.5: Método `tLinearAlgorithm::SolveCurrentStep()`.

```
1   class tIncrementalIntegrator:public tIntegrator
2   {
3      public:
4         tIncrementalIntegrator();
5
6         ~tIncrementalIntegrator();
7         void SetComponents(tDomain&, tLinearSOE&);
8
9         virtual bool FormTangent();
10        virtual bool FormUnbalance();
11        virtual bool Commit();
12        virtual bool UpdateResponse(const tVector&) = 0;
13  }; // tIncrementalIntegrator
```

Programa 3.6: Classe `tIncrementalIntegrator`.

```
1   int tFEMApp::Run()
2   {
3      puts(``fem 1.0 Copyright(c) 2006 GCGMC-DCT/UFMS'');
4      if (argc == 1)
5      {
6         Usage();
7         exit(0);
8      }
9
10     char* domainFileName = 0;
11
12     // Processa argumentos da linha de comando
13     ...
14
15     Domain = ReadDomain(domainFileName);
16     if (Domain == 0)
17        Error(UNABLE_TO_READ_DOMAIN_FILE, domainFileName);
18     CheckDomain(Domain);
19     if (PerformAnalysis())
20        WriteAnalysisResults();
21     return 0;
22  }
```

Programa 3.7: Método `tFEMApp::Run()`.

```
1   bool tFEMApp::PerformAnalysis()
2   {
3      if (NumberOfSubdomains > 1)
4         BEGIN_TASK_GROUP(``Partitioning domain'')
5            if (!PartitionDomain())
6               return false;
7         END_TASK_GROUP
8
9      tAnalysisDesc ad;
10     tStaticAnalysis* analysis = tAnalysisFactory::New(*Domain, ad);
11     bool success = false;
12
13     try
14     {
15        success = analysis->Execute();
16     }
17     catch (Exception& e)
18     {
19        puts(e.GetMessage());
20     }
21     catch (...)
22     {
23        puts(``Unknown error'');
24     }
25
26     delete analysis;
27     return success;
28  }
```

Programa 3.8: Método `tFEMApp::PerformAnalysis()`.

```
1   int main(int argc, char* argv[])
2   {
3      return tFEMApp(argc, argv).Run();
4   }
```

Programa 3.9: Função principal da análise seqüencial.

CAPÍTULO 4

Decomposição do Domínio

4.1 Considerações Iniciais

O particionamento da malha de elementos finitos do modelo de análise é fundamental para o bom desempenho da simulação paralela, dado que através dele pode-se garantir um bom balanceamento da carga envolvida no processamento distribuído. Várias técnicas podem ser utilizadas para realizar o particionamento da malha, algumas das quais são apresentadas ao longo deste capítulo. Neste trabalho, escolheu-se uma abordagem na qual, primeiramente, obtém-se um grafo equivalente à malha e particiona-se tal grafo, o que equivale ao particionamento da malha. Das várias técnicas que podem ser empregadas para particionar um grafo, optou-se pela utilização de funcionalidades fornecidas pelo pacote de código aberto METIS. Este foi escolhido por ser o que melhor se adequou aos propósitos almejados, pois implementa algoritmos eficientes, de acordo com a literatura pesquisada, e possui documentação precisa, possibilitando a compreensão de seu funcionamento.

O principal objetivo deste capítulo é apresentar as técnicas encontradas na literatura para o particionamento de malhas de elementos finitos e, particularmente, as adotadas no framework desenvolvido. Na Seção 4.2 são apresentadas algumas alternativas para a decomposição de domínio, bem como alguns trabalhos relacionados. O problema do particionamento de grafos e algumas das técnicas usadas nessa tarefa são o tema da Seção 4.3. A Seção 4.4 apresenta as principais características dos algoritmos utilizados pelo METIS para o particionamento, o algoritmo para particionamento em k-vias multinível (*multilevel k-way partitioning algorithm* – MLkP) e a bissecção recursiva multinível (*multilevel recursive bisection* – MLRB). As características da técnica da subestruturação são descritas na Seção 4.5. As classes responsáveis pelo particionamento de malhas, implementadas com base nas funcionalidades fornecidas pelo pacote METIS, são apresentadas na Seção 4.6.

4.2 Revisão Bibliográfica

Há duas estratégias, encontradas na literatura, que podem ser adotadas para realizar a decomposição de domínio: baseada em nós ou baseada em elementos. Na primeira, os nós da malha são distribuídos entre os processadores, enquanto na segunda são os elementos que são distribuídos. Quando a primeira estratégia é utilizada, cada nó é atribuído a apenas um processador, não havendo replicação de dados entre processadores. Essa abordagem possui algumas dificuldades associadas, uma vez que os cálculos realizados no MEF envolvem elementos completos, com todos os seus nós de extremidade. Dessa forma, cuidado adicional deve ser tomado para garantir que haja comunicação entre os processadores de maneira que as contribuições de todos os nós de um elemento sejam contabilizadas no resultado. Exemplo da utilização dessa abordagem pode ser encontrado em [JON94]. Utilizando a segunda estratégia todos os nós de um elemento se encontram armazenados no mesmo processador, sendo, assim, uma abordagem mais intuitiva, dado que a análise de elementos finitos é fundamentalmente baseada em elementos [DIN96]. Entretanto, essa estratégia apresenta a desvantagem de ocasionar a replicação dos nós compartilhados por mais de um elemento finito.

Em [DIN96] é utilizada a estratégia de particionamento baseada em elementos. Cada elemento é associado ao seu centro de massa, o chamado *centróide*, e o particionamento da malha de elementos finitos é feito sobre o conjunto de centróides com a utilização de um algoritmo de *bissecção inercial recursiva* (*recursive inertial bisection* – RIB). Feito esse particionamento, os nós são transferidos para os processadores nos quais os seus elementos se encontram. Ainda de acordo com [DIN96], a vantagem dessa abordagem está no fato de que as malhas resultantes também são malhas conexas e todos os cálculos sobre os elementos das mesmas podem ser efetuados como no caso seqüencial, sem referências a informações não-locais.

A obtenção de uma decomposição ótima é difícil, mas tem sido bastante pesquisada e alguns algoritmos heurísticos têm sido desenvolvidos para tentar obter uma solução aproximadamente ótima. Muitos algoritmos têm sido propostos para obter bons particionamentos de domínios, com bom balanceamento de carga entre os processadores. Um algoritmo encontrado em muitos dos textos da literatura estudada é o da *bissecção espectral recursiva* (*recursive spectral bisection* – RSB). O RSB produz partições de boa qualidade mas, devido ao alto custo, seu uso fica restrito a grafos pequenos [CAS00]. Além disso, o RSB apresenta a desvantagem de produzir apenas partições em um número que seja potência de dois, o que consiste em mais um fator restritivo, dado que o número de processadores a serem utilizados é arbitrário. A referência [HSI95] apresenta dois algoritmos que generalizam o RSB básico para geração de um número arbitrário de subdomínios. Ambos os algoritmos utilizam o particionamento de grafos através do uso de técnicas espectrais e de representação em grafos de malhas de elementos finitos. Nesse mesmo trabalho, utiliza-se o *vetor de conectividade algébrica* como métrica para medir a qualidade da partição produzida.

Devido ao alto custo do RSB, utilizam-se métodos multiníveis para grafos de tamanho grande, como por exemplo, o *algoritmo de Kernighan-Lin Multiníveis (KL-Multiníveis)* [CAS00]. Grande número de pesquisas vêm sendo realizadas com a intenção de obter algoritmos multiníveis com complexidade computacional moderada para particionamento de grafos. Alguns esquemas estudados resultam em particionamentos muito próximos aos ótimos, algumas vezes, melhores do que aqueles resultantes da utilização

de métodos espectrais [BUI93, HEN93a, KAR95, KAR98b, KAR98c]. Os algoritmos multiníveis são mais rápidos do que os métodos espectrais. Em [KAR96a] é proposto um algoritmo multinível para o particionamento de grafos, cuja idéia principal reside no particionamento dos vértices do grafo em $\sqrt{p}$ partes (p é o número de processadores), enquanto a matriz de adjacências do grafo é dividida entre todos os p processadores. Essa estratégia conduziu a um algoritmo paralelo que alcançou *speedups* iguais a 56 com 128 processadores em problemas de tamanho moderado e produziu particionamentos de boa qualidade [KAR96a].

Em [BAR95a] é apresentada a *bissecção espectral recursiva multiníveis paralela* (*parallel multilevel recursive spectral bisection* – PMRSB), uma implementação paralela do método da bissecção espectral recursiva multiníveis (MRSB). O objetivo dessa paralelização é possibilitar o reparticionamento dinâmico de malhas adaptáveis, bem como o particionamento de malhas cujo tamanho exceda a capacidade de armazenamento das estações de trabalho. A ferramenta foi implementada sobre plataforma Cray T3D e é restrita a um número de processadores potência de dois. PMRSB se apóia na arquitetura de memória compartilhada distribuída[1] e apresenta como principais vantagens a possibilidade de resolver problemas muito maiores do que os passíveis de solução seqüencial, assim como execução mais rápida. Deve-se levar em consideração que, se ao invés da arquitetura de memória compartilhada distribuída fosse utilizada a arquitetura convencional de troca de mensagens, muitos dos algoritmos apresentados por Barnard [BAR95a] seriam ineficientes.

Outra abordagem encontrada na literatura pesquisada para particionamento de grafos é a utilização de *simulated annealing*. O conceito de *simulated annealing* foi primeiramente introduzido no ano de 1953 em [MET53] e se refere a uma generalização do *método de Monte Carlo*[2] para determinar o estado de equilíbrio de um conjunto de átomos a uma temperatura T qualquer [NAH86, SAN97]. No ano de 1983, em [KIR83], *simulated annealing* começou a ser utilizado como uma técnica heurística para encontrar soluções aproximadas para problemas difíceis de otimização combinatória. O método simula um processo estocástico[3] no espaço das soluções factíveis, no qual transições entre os estados são governadas por uma regra que favorece estados com menor custo ou "energia". O parâmetro "temperatura" compara o quão favorecidos são os estágios de menor energia: em um equilíbrio à baixa temperatura, estados de baixa energia ocorrem com probabilidade muito maior do que estados de alta energia, enquanto a temperaturas maiores essa probabilidade é menor [JER93]. Outras considerações acerca dessa técnica e algumas de suas aplicações podem ser encontradas em [JOH89, JOH91, ING93, FLE95, SWI00].

Em [NIK96] é apresentado um modelo analítico para estimar a eficiência paralela do

[1]Memória compartilhada distribuída é uma abstração utilizada para permitir o compartilhamento de dados entre computadores que não compartilham memória física. O acesso a esse tipo de memória é feito com se fosse local, havendo um sistema que permite a transparência do acesso, garantindo que os computadores possam "enxergar" as mudanças feitas pelos outros. Tem-se a impressão de uma memória compartilhada única, mas a memória está fisicamente distribuída [COU01].

[2]O método de Monte Carlo fornece soluções aproximadas para alguns problemas matemáticos através da realização de experimentos de amostragem estatística com a utilização de um computador. Conceitos e outras aplicações do método podem ser encontradas em [FIS96].

[3]Processo estocástico refere-se a situações em que são feitas observações quanto a um período de tempo, situações essas influenciadas por efeitos aleatórios ou de azar, não só em um único instante, mas por todo o intervalo de tempo ou seqüência de tempo que se está considerando [CLA79].

método de decomposição de domínio utilizado no trabalho por eles desenvolvido. De acordo com esse trabalho, métodos iterativos como, por exemplo, o *método do gradiente conjugado* (*conjugate gradient method* – CGM) [SHE94], são muito empregados para a solução de sistemas de equações, devido à sua simples paralelização. Entretanto, em alguns casos, certos sistemas de equações não podem ser resolvidos pelo CGM e, para evitar problemas de convergência, são utilizados métodos diretos para solucionar os sistemas de equações obtidos com a decomposição do domínio. O algoritmo para decomposição apresentado em [NIK96] utiliza *decomposição LDU* nos subdomínios e sistemas de equações de interface.

Grande parte dos algoritmos seqüenciais para particionamento de grafos não podem ser facilmente paralelizados. Em [BAR95b], Barnard e Simon apresentam uma implementação paralela de seu algoritmo espectral, mas o mesmo não possui boa escalabilidade. O algoritmo de Kernighan-Lin usado na fase de projeção também não é muito adequado à paralelização. Tentativas de obter bom desempenho podem ser vistas em [SAV91, KAR96a], resultando, porém, em processos com grande necessidade de comunicação [CAS00].

Outra alternativa é a utilização de métodos de particionamento geométrico de grafos, os quais se baseiam em informações a respeito das coordenadas dos mesmos, que, no caso de malhas de elementos finitos, estão facilmente disponíveis. As heurísticas geométricas são escaláveis, mas as partições produzidas não são tão boas quanto as dos métodos espectrais [CAS00]. As técnicas geométricas são abordadas na Seção 4.3, bem como outras técnicas de particionamento de grafos.

Em [MOR00] o programa adotado para os cálculos de elementos finitos é o *FEMOOP* (*finite element method – object oriented programming*), implementado utilizando conceitos de orientação a objetos. Para permitir a execução paralela, FEMOOP teve acrescido a seu código uma classe para permitir a comunicação entre os processadores. Essa classe encapsula a biblioteca responsável pelo gerenciamento da troca de mensagens. O encapsulamento torna possível uma eventual troca ou acréscimo de bibliotecas de comunicação. As bibliotecas atualmente suportadas são PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*), e o usuário pode optar pelo uso de uma delas em tempo de compilação. Muitos esforços têm sido aplicados no sentido de paralelizar a geração da malha de elementos finitos, que é seqüencial em FEMOOP. A idéia principal para a paralelização é a divisão do domínio inteiro em subdomínios distribuídos entre os processadores. Feito isso, cada processador geraria a sua malha local independentemente dos demais.

Em [SOT02] são apresentados dois tipos de métodos que podem ser utilizados para a paralelização: métodos baseados em subestruturação e métodos de "quebra" do domínio. O primeiro tipo corresponde aos métodos de decomposição do domínio. Nesses métodos, os graus de liberdade internos são condensados de forma a reduzir o sistema de equações ao número de graus de liberdade de interface. Essa condensação pode ser realizada em paralelo pelos processadores, de maneira independente. O sistema global de equações reduzido pode ser obtido também em paralelo com a utilização de solucionadores de equações diretos ou iterativos. Nesse caso, há a necessidade de comunicação entre os processadores, uma vez que pode haver necessidade de acesso a uma mesma posição da matriz e do vetor globais. Os métodos de "quebra" do domínio também são baseados na divisão do problema em subproblemas menores. A diferença fundamental com relação aos anteriores é a não utilização de subestruturação. Ao invés

disso, são utilizadas outras abordagens para restaurar a unicidade da solução nos graus de liberdade de interface [SOT02].

Em [FRA03] pode-se encontrar um estudo sobre alguns métodos de decomposição de domínio. Nesse trabalho é descrito um ambiente teórico para formulação de métodos de decomposição, o qual visa proporcionar uma plataforma matemática para tratamento uniforme de métodos de decomposição de domínio de alto desempenho em mecânica estrutural. Também é apresentada uma metodologia para desenvolvimento de novos métodos de decomposição a partir dos já existentes.

As preocupações apresentadas até o presente momento, divisão em super-elementos de tamanho aproximadamente igual e poucos nós de intersecção entre eles, são os principais aspectos a serem considerados quando são utilizadas máquinas idênticas na execução. O mesmo não ocorre quando da utilização de máquinas heterogêneas, situação na qual deve-se levar em consideração as características de cada uma das máquinas e da rede de interconexão para efetuar o particionamento mais apropriado. A referência [CHE02] apresenta um método para decomposição de domínio quando máquinas com diferentes configurações são utilizadas.

Métodos para realizar o reparticionamento de malhas também vêm sendo estudados nos últimos tempos. Com a utilização desses métodos, a malha é periodicamente redividida com a intenção de manter o balanceamento da carga entre os processadores. Dessa forma, os algoritmos utilizados nesse reparticionamento não podem ter custo elevado, uma vez que são chamados várias vezes durante a execução do método, e a transferência de elementos entre os processadores deve ser mínima. Um estudo mais detalhado dos métodos para reparticionamento de malhas pode ser encontrado nas referências [BIS97, SCH97, SCH98, WAL97].

Muitos métodos de decomposição de domínio são baseados em particionamento de grafos. Nessa abordagem é construído um grafo que representa a malha de elementos finitos. Tal grafo geralmente é de um dos seguintes tipos: grafo dual ou grafo de nós [HSI94]. No grafo dual, cada vértice corresponde a um elemento da malha de elementos finitos; se dois elementos possuem uma face (ou lado, no caso bidimensional) em comum, então deve existir uma aresta conectando os dois vértices equivalentes. O grafo de nós é o de concepção mais intuitiva, nele cada nó da malha corresponderá a um vértice do grafo e as arestas são as mesmas que conectam os nós da malha.

A construção do grafo dual relativo à malha é uma das técnicas mais utilizadas. Após a obtenção do grafo, o problema, agora, resume-se a efetuar o particionamento do grafo dual em p subgrafos de, aproximadamente, mesmo tamanho, minimizando o número de arestas de interconexão entre os vértices em diferentes subgrafos. Esse problema é conhecido como *problema do particionamento de grafos em p partes* (*p-way graph partitioning problem*) e é NP-difícil, mesmo no caso da divisão do grafo entre apenas dois processadores. Dessa forma, como tentativa de resolver o problema de maneira eficiente, várias heurísticas vêm sendo desenvolvidas. Dentre essas, destacam-se a bisecção espectral recursiva e o algoritmo de Kernighan-Lin. Com a obtenção desse particionamento, tem-se a carga dos processadores praticamente igual e comunicação mínima [CAS00, KAR96a].

Dentre todas as alternativas pesquisadas optou-se, neste trabalho, pela utilização de um pacote de programas que realize o particionamento do grafo equivalente à malha de elementos finitos, entrada do problema. O problema de particionamento de grafos é

apresentado na próxima seção e o pacote escolhido para realizar essa tarefa, o METIS, é apresentado na seção subseqüente.

4.3 Particionamento de Grafos

O particionamento de grafos possui grande importância na computação paralela e distribuída porque possibilita identificar a concorrência inerente a um dado problema, desde que o mesmo seja passível de modelagem através do uso de grafos. O particionamento leva a uma decomposição da carga de trabalho entre os processadores, através da atribuição aos processadores dos diferentes subgrafos obtidos. O tamanho dos subgrafos indica a quantidade de processamento que cada processador deve realizar e o número de arestas "cortadas" indica a quantidade de comunicação necessária [POT95]. Arestas cortadas são aquelas que, quando do particionamento de um grafo, possuem seus vértices extremos atribuídos a diferentes subgrafos.

Algoritmos para realizar bons particionamentos de grafos são de fundamental importância em uma grande variedade de aplicações científicas, as quais demandam alto desempenho em computadores paralelos. Nessas aplicações, as computações são realizadas em cada elemento de processamento, cada qual com sua parte do domínio do problema, e informações são trocadas entre elementos adjacentes. Para que tal computação seja realizada de maneira eficiente, é necessário que o domínio do problema, no caso a malha de elementos, seja dividida de forma que cada processador realize aproximadamente a mesma quantidade de processamento e que a troca de informações entre os processadores seja mínima.

O problema do particionamento de grafos é definido como segue [FJA98]:

Dado um grafo $G = (V, A)$, onde V é um conjunto de vértices com pesos e A é um conjunto de arestas também com pesos associados, e um inteiro positivo p, deve-se encontrar p subconjuntos $N_1, N_2, ..., N_p$ tais que:

1. $\cup_{i=1}^{p} N_i = N$ e $N_i \cap N_j = \emptyset$ para $i \neq j$;
2. $W(i) \approx \frac{W}{p}, i = 1, 2, ..., p$, onde $W(i)$ e W são as somas dos pesos dos nós em N_i e N, respectivamente;
3. o *tamanho do corte*, ou seja, a soma dos pesos das arestas com extremos em subconjuntos diferentes é mínima.

Qualquer conjunto $\{N_i \subseteq N : 1 \leq i \leq p\}$ é denominado uma partição em p partes de N se satisfaz a condição 1. Uma *bissecção* é, dessa forma, uma partição em duas partes. Uma partição que satisfaz a propriedade 2 é considerada balanceada. A Figura 4.1 ilustra o particionamento de um grafo em quatro partes.

O particionamento e posterior atribuição de cada subgrafo obtido a um processador diferente resulta em uma boa partição da malha de elementos finitos, ou seja, um bom balanceamento da carga inicial de trabalho. Além disso, como as técnicas de particionamento procuram realizar partições com o número mínimo de arestas de corte, a comunicação entre processadores também será pequena, uma vez que as arestas de corte representam a existência de nós de interface compartilhados pelos processadores.

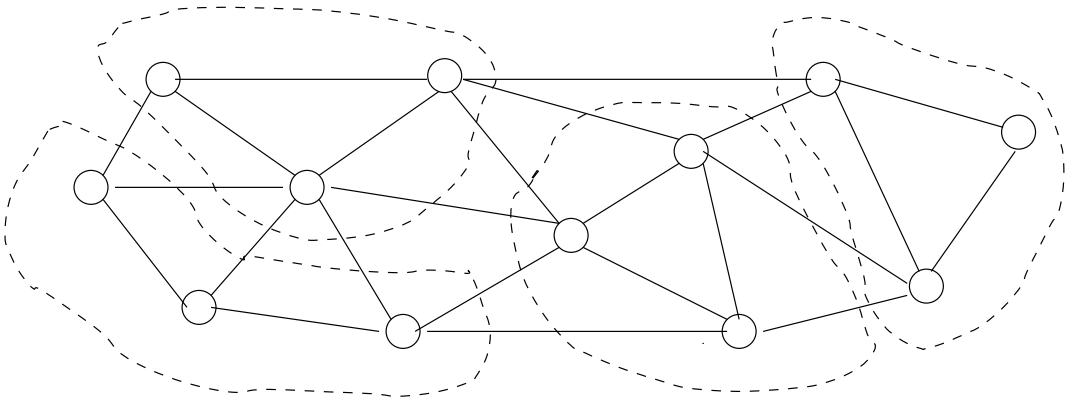


Figura 4.1: Grafo particionado em quatro partes.

O problema do particionamento de grafos é NP-completo, o que levou ao desenvolvimento de várias heurísticas para conseguir soluções aproximadas para o problema. Tais heurísticas podem ser classificadas como geométricas, combinatoriais, espectrais, técnicas de otimização combinatoriais ou métodos multiníveis. Na Subseção 4.3.1 são apresentadas as técnicas geométricas.

4.3.1 Técnicas Geométricas

As técnicas geométricas realizam o particionamento com base apenas nas coordenadas dos nós da malha, e não na conectividade da mesma. Dado que não são consideradas informações a respeito da conectividade da malha, o conceito de arestas cortadas não faz sentido nesse contexto. Dessa forma, para minimizar a comunicação entre os processadores essas técnicas devem utilizar uma outra métrica. Uma dessas métricas é a minimização do número de elementos do subdomínio adjacentes a elementos não-locais, ou seja, a fronteira do subdomínio é minimizada. Como essas técnicas, geralmente, particionam os elementos da malha diretamente e não os grafos equivalentes, elas são denominadas *esquemas de particionamento da malha*.

Técnicas geométricas somente podem ser utilizadas quando informações a respeito das coordenadas dos nós da malha estão disponíveis, sendo bastante velozes. Apesar disso, os particionamentos produzidos nem sempre são de boa qualidade. São apresentadas duas técnicas geométricas bastante utilizadas, a bissecção coordenada recursiva e a bissecção inercial recursiva.

Bissecção Coordenada Recursiva

A bissecção coordenada recursiva (*recursive coordinate bisection* - RCB), também conhecida como dissecção aninhada coordenada (*coordinate nested dissection* - CND), é um esquema que tenta minimizar as fronteiras entre os subdomínios e, conseqüentemente, a necessidade de comunicação.

No RCB a malha é dividida em duas através da normal ao seu eixo mais longo. O método inicialmente calcula os centros de massa, os centróides, dos elementos da malha, os quais são projetados no eixo correspondente à mais longa dimensão da malha, o que resulta em uma ordenação dos elementos. Essa lista ordenada é dividida ao meio,

produzindo uma bissecção da malha. Cada subdomínio obtido pode ser subdividido recursivamente com a utilização da mesma técnica.

O RCB executa rapidamente, consome pouca memória e é de fácil paralelização. Porém, os particionamentos podem ser de baixa qualidade. Quando aplicado a malhas com geometrias complicadas, pode produzir partições com subdomínios desconexos.

Bissecção Inercial Recursiva

Uma das restrições do RCB é que o mesmo produz apenas bissecções normais a um dos eixos de coordenadas, o que pode limitar a qualidade do particionamento. A Figura 4.2(a) ilustra um exemplo no qual o RCB não produz uma boa partição. A malha apresentada na figura é dividida de acordo com a normal à dimensão mais longa da mesma. A fronteira do subdomínio é longa porque não foi levado em consideração o fato de que a malha está inclinada com relação aos eixos de coordenadas.

O algoritmo da bissecção inercial recursiva (*recursive inertial bisection* - RIB) é uma tentativa de aperfeiçoar o RCB, levando em consideração as inclinações da malha com relação aos eixos. O RIB primeiramente computa o *eixo inercial* da malha e a projeção dos centróides é feita sobre esse eixo. O restante do procedimento é como no RCB. A Figura 4.2(b) ilustra a mesma malha de 4.2(a), dividida com a utilização do RIB. A seta indica o eixo de projeção utilizado pelo RIB, enquanto a linha tracejada indica a bissecção realizada.

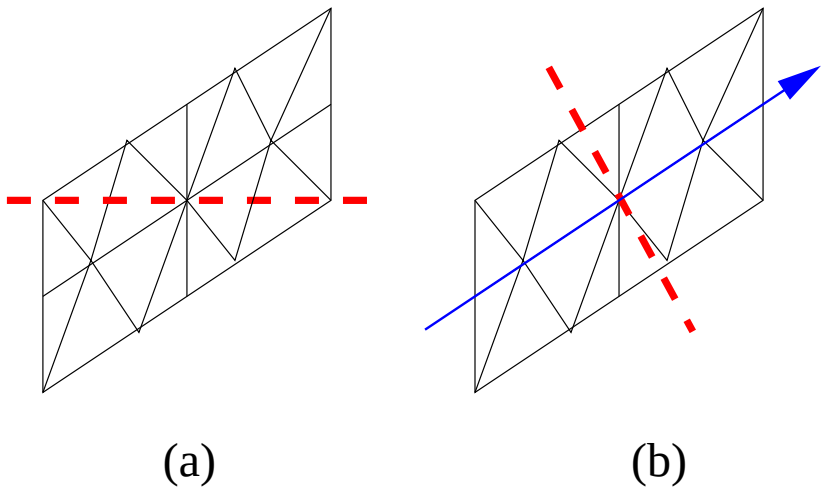


Figura 4.2: Divisão de uma malha utilizando: (a) o RCB e (b) o RIB.

4.4 Algoritmos de Particionamento Multinível

Os particionamentos multiníveis utilizam estratégias bem diferentes dos algoritmos tradicionais para particionamento de grafos, os quais agem diretamente sobre o grafo original fornecido como entrada (essa estratégia é ilustrada na Figura 4.3). A estratégia utilizada pelos particionamentos multiníveis consiste basicamente em três partes principais: a redução do grafo, o particionamento desse grafo reduzido e a expansão desse

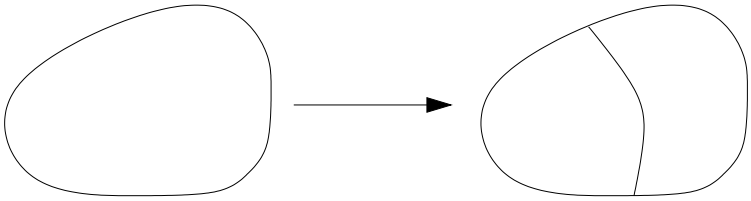


Figura 4.3: Particionamento tradicional.

particionamento para o grafo original. METIS utiliza novas abordagens para a redução sucessiva do tamanho do grafo, bem como técnicas para refinar os particionamentos obtidos a cada nível da fase de expansão. A Figura 4.4 ilustra o esquema utilizado pelo particionamento multinível de grafos.

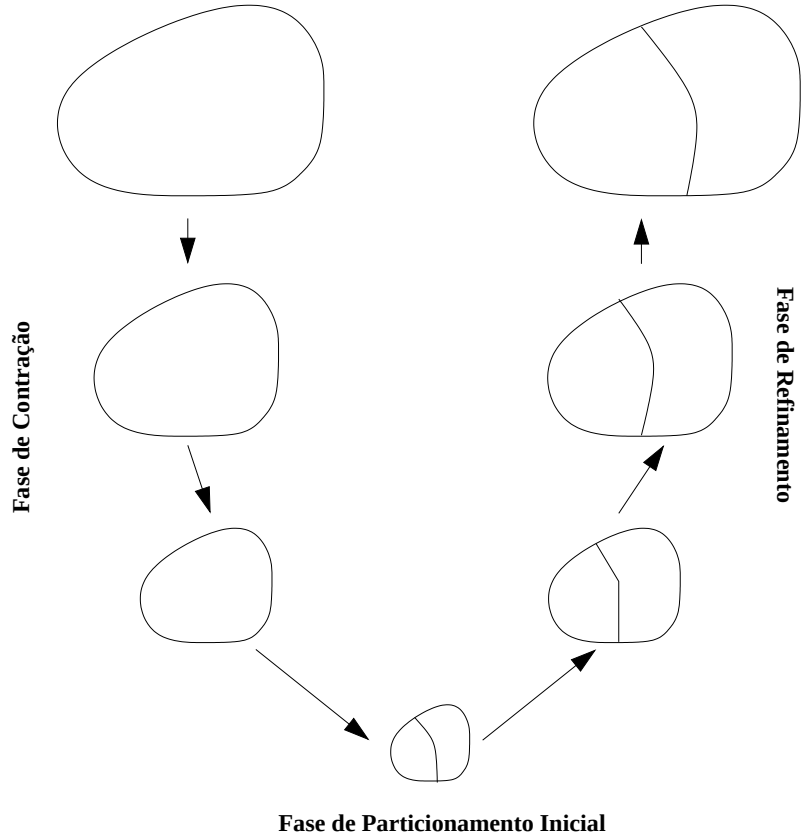


Figura 4.4: Particionamento multinível.

O particionamento no METIS é realizado com a utilização de dois algoritmos, o algoritmo para particionamento em k-vias multinível e a bissecção recursiva multinível. O primeiro algoritmo é aplicado quando se deseja particionar o grafo de entrada em mais de oito partes, o segundo é utilizado quando se deve particionar o grafo em oito partes ou menos.

Para utilizar o particionamento de grafos fornecidos por METIS, é necessário transformar a malha de elementos finitos de entrada em um grafo equivalente. O pacote fornece duas opções: transformação em um grafo nodal ou em um grafo dual. Apesar

disso, optou-se pelo desenvolvimento de uma nova função que realize essa transformação e obtenha o grafo dual equivalente à malha. Essa escolha se deve à necessidade de superar uma das limitações oferecidas pelos procedimentos fornecidos por METIS, os quais exigem que a malha seja constituída apenas por elementos de um mesmo tipo. Escolheu-se o grafo dual porque o método dos elementos finitos é baseado em elementos e, dessa forma, o particionamento do grafo dual é mais representativo. A transformação da malha em um grafo dual equivalente possibilita uma forma natural de minimizar a comunicação necessária entre partições adjacentes, a minimização das arestas de corte no particionamento do grafo.

Ambos os algoritmos utilizados utilizam a estratégia multinível de redução, particionamento inicial e expansão, acompanhada de refinamento do particionamento inicial. As próximas subseções apresentam, respectivamente, a bissecção recursiva multinível (MLRB) e o particionamento em k-vias multinível (MLkP).

4.4.1 Bissecção Recursiva Multinível

A bissecção recursiva é freqüentemente utilizada no particionamento de grafos em k subgrafos. Por essa técnica, primeiramente, obtém-se uma partição em duas partes do grafo inicial e, após isso, cada uma das partições resultantes também é particionada em duas partes e assim por diante. Ao final de $log\ k$ iterações, o número de partições desejadas é alcançado e o particionamento é encerrado. Essa abordagem transforma o problema de particionar em uma seqüência de bissecções.

A bissecção recursiva multinível (MLRB) surgiu como uma alternativa eficaz para particionar um grafo e sua estrutura básica é simples [KAR98c]. No MLRB o grafo G é "compactado", resultando em um grafo G' com número reduzido de vértices, o qual, por sua vez, é particionado em k partes através da bissecção recursiva. Devido ao seu tamanho relativamente pequeno, G' é particionado mais facilmente. O particionamento obtido é, dessa forma, projetado e refinado nos diversos grafos intermediários obtidos na fase da contração inicial, até atingir o grafo inicial G. Após o término dessa fase de projeção, tem-se o particionamento em k partes de G. A complexidade do MLRB para particionar um grafo $G = (V, E)$ é $O(|E|\ log\ k)$.

4.4.2 Particionamento em k-vias Multinível

O algoritmo de particionamento em k-vias multinível (MLkP), assim como a bissecção recursiva, também se baseia na compactação, particionamento inicial e posterior projeção e refinamento nos grafos intermediários obtidos na primeira fase. A diferença fundamental entre os dois métodos está na maneira como esse particionamento inicial é realizado; no MLkP o grafo reduzido é diretamente particionado em k partes. De acordo com Karypis e Kumar [KAR98c], há uma série de vantagens nessa abordagem; primeiramente, nela o grafo precisa ser compactado apenas uma vez, reduzindo a complexidade dessa fase para $O(|E|)$, ao invés de $O(|E|\ log\ k)$. Além disso, sabe-se que o particionamento diretamente em k partes pode produzir particionamentos arbitrariamente melhores do que a bissecção recursiva [SIM93]. Dessa forma, potencialmente, um método que obtenha uma k-partição de maneira direta, produz partições melhores.

Apesar de ambos os problemas serem NP-completos, é mais simples a obtenção de uma bissecção do que a obtenção direta de uma k-partição. Por esse motivo, essa

última é geralmente obtida por meio de bissecções sucessivas. Entretanto, no contexto dos particionamentos multiníveis, apenas um particionamento de qualidade razoável do grafo reduzido é necessário, dado que o mesmo será refinado na etapa de projeção nos grafos intermediários e no original. A desvantagem, contudo, na obtenção direta de uma k-partição, está na posterior necessidade de refinar a mesma, a qual é consideravelmente mais difícil que refinar uma bissecção [KAR98c]. A utilização de um refinamento KL mesmo para um grafo com oito partições é muito alto quando comparado com a bissecção, o que torna a utilização desse esquema de refinamento proibitivo.

Karypis e Kumar apresentam em [KAR98c] um algoritmo de particionamento em k partes cujo a complexidade é $O(|E|)$, ou seja, é linear ao número de arestas. Uma contribuição importante do trabalho é o esquema para refinamento da k-partição obtida. De acordo com os autores o esquema apresentado é substancialmente mais rápido do que o algoritmo KL de refinamento de uma bissecção [HEN93b] e igualmente eficaz, além de ser inerentemente paralelo [KAR96b]. As três principais fases do particionamento em k-vias multinível são descritas a seguir.

Fase de Compactação

A fase de compactação é responsável pela obtenção sucessiva de grafos reduzidos, ou seja, a partir de um grafo $G_0 = (V_0, E_0)$ deseja obter uma seqüência de grafos $G_i = (V_i, E_i)$ cujo número de vértices seja menor do que $G_{i-1} = (V_{i-1}, E_{i-1})$. Para realizar tal tarefa, muitos esquemas de compactação utilizam uma estratégia na qual um subconjunto de vértices de G_i são combinados para formar um único vértice em G_{i+1}. Seja V_i^v o subconjunto de vértices em G_i combinados para formar o vértice v em G_{i+1}. Para que se obtenha um bom particionamento de um grafo reduzido com relação ao grafo original é necessário que o peso do vértice v seja igual à soma dos pesos dos vértices em V_i^v. Além disso, para manter as informações de conectividade no grafo reduzido, as arestas de v devem corresponder à união das arestas dos incidentes nos vértices de V_i^v. Esse método de compactação garante as seguintes propriedades [HEN93a]:

- O corte de arestas de uma partição em um grafo reduzido é igual ao corte da mesma partição no grafo original;
- Uma partição balanceada de grafos reduzidos leva a uma partição também balanceada do grafo original.

A união de vértices adjacentes pode ser formalmente definida em termos de *matchings*. Um *matching* de um grafo é um conjunto de arestas no qual não existem duas arestas incidentes no mesmo vértice [KAR98c]. Com base nesse conceito, um grafo reduzido G_{i+1} é obtido a partir de G_i encontrando um *matching* de G_i e unindo os vértices do *matching* em "multinós", ou seja nós formados pela contração de outros. Os vértices não pertecentes ao *matching* são simplesmente copiados para G_{i+1}. Dado que a meta da união de vértices com a utilização de *matchings* é a diminuição do tamanho do grafo G_i, o tamanho de tal *matching* deve ser máximo. Um *matching* é considerado máximo se não é possível adicionar um outro vértice ao mesmo sem fazer com que duas arestas incidam sobre um mesmo vértice. Deve-se notar que, dependendo da maneira como os *matchings* forem calculados, o tamanho do *matching* máximo pode ser diferente.

A fase de compactação termina quando o grafo reduzido G_m tem um número suficientemente pequeno de vértices ou se a redução de tamanho entre grafos sucessivos for muito pequena. Nos experimentos de Karypis e Kumar [KAR98c], a fase de compactação era encerrada quando o número de vértices em um dos grafos reduzidos fosse menor do que ck, onde $c = 15$. A escolha desse valor para c, de acordo com eles, foi feita de forma a permitir que o particionamento inicial criasse partições com, aproximadamente, o mesmo tamanho. A redução nesses experimentos também era encerrada se a redução no tamanho de grafos sucessivos fosse menor que um fator de 0.8.

Há diversas formas de selecionar um *matching* máximo para a compactação, sendo três delas apresentadas no restante desta subseção, o *Matching* Aleatório (*Random Matching* - RM), *Heavy Edge Matching* - HEM e *Sorted Heavy-Edge Matching* - SHEM.

Matching Aleatório

Baseia-se na idéia de que um *matching* maximal pode ser obtido com a utilização de um algoritmo aleatório [KAR98c]. Nesse algoritmo os vértices são visitados em ordem aleatória, se o vértice u ainda não foi "casado"[4] tenta-se selecionar aleatoriamente um vértice v qualquer adjacente a ele. Se tal vértice existir, a aresta (u, v) é incluída no *matching* e ambos os vértices são marcados como casados. Caso contrário, o vértice u permanece como não casado. Esse algorimo possui complexidade $O(|E|)$.

Heavy-Edge Matching

O *matching* aleatório utiliza uma estratégia gulosa de diminuição dos níveis de compactação para obter um *matching* máximo. Entretanto, o objetivo em METIS é a obtenção de um *matching* maximal que minimize o número de arestas de corte [KAR98c]. Considere um grafo $G_i = (V_i, E_i)$, um *matching* M_i usado para compactar G_i e o grafo obtido com essa compactação $G_{i+1} = (V_{i+1}, E_{i+1})$. Define-se $W(A)$ como a soma dos pesos das arestas de A, onde A é um conjunto de arestas. Pode-se provar que [KAR98c]:

$$W(E_{i+1}) = W(E_i) - W(M_i). \tag{4.1}$$

Dessa forma, com a seleção de um *matching* maximal M_i cujas arestas tenham um peso alto, pode-se obter uma grande redução do valor de W_{i+1}. De acordo com a análise feita em [KAR95], dado que G_{i+1} tem um menor valor de W_{i+1}, esse mesmo grafo reduzido possui um menor corte de arestas.

O funcionamento do *heavy-edge matching* baseia-se nessa característica para escolher um *matching* maximal cujas arestas possuem o maior peso, como introduzido em [KAR98d]. O *matching* é obtido com a utilização de um algoritmo aleatório semelhante ao usado para obtenção de um *matching* aleatório. Assim como nesse algoritmo, os vértices são visitados em ordem aleatória, entretanto, a escolha do vértice adjacente com o qual u deve ser casado é feita de maneira diferente. Ao invés de escolher um vértice qualquer, escolhe-se o vértice v adjacente a u tal que o peso da aresta (u, v) seja máximo com relação às outras possíveis arestas. Deve-se ressaltar que esse algoritmo não garante que o *matching* obtido possui o maior peso possível, mas experimentos mostram que ele funciona bem na prática [KAR98c] e sua complexidade é igual a $O(|E|)$.

[4] Um vértice é denominado casado quando faz parte de algum *matching*.

Sorted Heavy-Edge Matching

Uma variação do algoritmo *heavy-edge matching*, descrito anteriormente, baseia-se na alteração da ordem de visitas aos vértices adjacentes a u, os quais serão visitados em ordem crescente do peso das arestas que os conectam a u. Essa variação é denominada *sorted heavy-edge matching* e torna possível evitar que o peso da aresta entre os vértices u e v exceda um determinado valor predefinido[KUC05].

Fase de Particionamento Inicial

A segunda fase do algoritmo consiste na obtenção de um particionamento inicial P_m do grafo reduzido $G_m = (V_m, E_m)$, tal que cada partição contém aproximadamente $1/k$ da soma dos pesos dos vértices de G_0; no caso do grafo G_0 possuir todos os vértices com o mesmo peso, isso equivale a dizer que cada partição possui aproximadamente $|V_0|/k$ vértices do grafo original.

Uma alternativa para obtenção de um particionamento inicial seria realizar a compactação do grafo original até alcançar um grafo com exatamente k vértices. O particionamento desse grafo pode ser obtido de maneira direta, com cada vértice pertencendo a uma partição diferente. Entretanto, há dois problemas com essa abordagem [KAR98c]:

- A redução de tamanho do grafo em relação ao grafo antecendente torna-se menor a cada passo da compactação, tornando o processo muito caro;
- Se a compactação até a obtenção de um grafo com k vértices for possível, o mesmo possuiria vértices com pesos muito diferentes, o que tornaria o particionamento inicial obtido desbalanceado.

No algoritmo apresentado po Karypis e Kumar[KAR98c] o particionamento inicial é realizado através do algoritmo de bissecção recursiva multinível, apresentado anteriormente.

Fase de Descompactação

A fase de descompactação é a responsável pela projeção do particionamento inicial do grafo G_m nos grafos intermediários $G_{m-1}, G_{m-2}, \cdots, G_1$ até o grafo de entrada G_0. Dado que cada vértice v de G_{i+1} equivale à junção de um subconjunto V_i^v de vértices de G_i, pode-se obter o particionamento P_i a partir de P_{i+1} através da atribuição dos vértices de V_i^v à mesma partição, em P_i, que contém v em P_{i+1}, ou seja, $P_i[u] = P_{i+1}[v], \forall u \in V_i^v$. Deve-se salientar que a partição obtida a cada passo da descompactação pode não ser tão boa quanto à do grafo anterior, sendo possível melhorá-la através da utilização de heurísticas locais [KAR98c].

Dentre os algoritmos de refinamento local, destacam-se os baseados no algoritmo de particionamento Kernighan-Lin (KL) [KER70] e suas variantes [FID82, HEN93a], dado que os mesmos tendem a obter bons resultados [KAR98c]. O algoritmo de KL troca vértices entre as partições de uma bissecção com a intenção de reduzir o corte de arestas do particionamento, até que se alcance um mínimo local[5]. Entretanto o

[5]Um particionamento está em mínimo local se a movimentação de qualquer vértice para uma outra partição não melhora o corte de arestas [KAR98c]

refinamento de particionamentos genéricos, denominado refinamento em k-vias, não é tão simples como em bissecções, dado que vértices podem ser movidos para mais de uma partição, o que aumenta a complexidade do problema. Uma extensão do algoritmo KL para refinamento em k-vias, cuja complexidade para um grafo de m arestas é igual a $O(k * m)$, é apresentado em [HEN92]. Devido à alta complexidade, esse algoritmo deve ser utilizado somente em particionamento em poucas partes, ou seja, com valores pequenos de k.

Karypis e Kumar [KAR98c] apresentam algoritmos de refinamento em k-vias que consistem em versões simplificadas do algoritmo de refinamento em k vias KL, cujas complexidades são independentes do número de partições. De acordo com os resultados obtidos pelos autores, esses algoritmos produzem particionamentos de boa qualidade em um tempo pequeno. A descrição desses algoritmos foge ao escopo deste trabalho, podendo ser encontrada em [KAR98c].

4.5 Subestruturação

A subestruturação ou particionamento estrutural consiste na divisão de um domínio em um número de subdomínios, ou subestruturas, cujas fronteiras podem ser especificadas arbitrariamente. Por conveniência, geralmente o particionamento estrutural é feito de acordo com o particionamento físico. Se cada subestrutura possui propriedades próprias de rigidez, cada uma delas pode ser tratada como um elemento estrutural complexo. Segue que os métodos de análise podem também ser aplicados à estrutura particionada. Encontrados os deslocamentos ou forças nas fronteiras das subestruturas, cada uma pode ser analisada separadamente, sob a ação de deslocamentos ou forças conhecidas em suas fronteiras.

No MEF, cada subestrutura é analisada separadamente, assumindo que suas fronteiras comuns com subestruturas adjacentes estão completamente fixas. Após isso, tais fronteiras são "relaxadas" simultaneamente e os deslocamentos reais das mesmas são determinados a partir das equações de equilíbrio de forças nas junções de fronteira. A solução dos deslocamentos das fronteiras envolve um número de incógnitas reduzido quando comparado à solução da estrutura não particionada. Cada estrutura pode ser analisada separadamente sob carga e deslocamentos de fronteira conhecidos, o que pode ser feito facilmente, uma vez que as matrizes envolvidas são de tamanho relativamente pequeno.

Conforme visto no Capítulo 3, o equilíbrio estático de um sólido é representado, no MEF, pela equação

$$\mathbf{K}\,\mathbf{U} = \mathbf{F}. \qquad \text{(3.32-repetida)}$$

Com a divisão do sólido em subestruturas são introduzidas fronteiras interiores no domínio a ser analisado. A matriz coluna dos deslocamentos comuns a mais de uma subestrutura será denotada por $\mathbf{U}_b$ e a matriz de deslocamentos dos pontos interiores denotada por $\mathbf{U}_i$. Representando as matrizes de forças externas de fronteira e interiores por, respectivamente, $\mathbf{F}_b$ e $\mathbf{F}_i$, pode-se reescrever a Equação (3.32) como:

$$\begin{bmatrix} \mathbf{K}_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{U}_b \\ \mathbf{U}_i \end{bmatrix} = \begin{bmatrix} \mathbf{F}_b \\ \mathbf{F}_i \end{bmatrix}. \qquad (4.2)$$

Assume-se, agora, que os deslocamentos totais da estrutura podem ser calculados

pela sobreposição de duas matrizes tais que

$$\mathbf{U} = \mathbf{U}^{(\alpha)} + \mathbf{U}^{(\beta)}, \tag{4.3}$$

onde $\mathbf{U}^{(\alpha)}$ representa a matriz dos deslocamentos devidos a $\mathbf{F}_i$ com $\mathbf{U}_b = 0$, e $\mathbf{U}^{(\beta)}$ representa as correções necessárias aos deslocamentos $\mathbf{U}^{(\alpha)}$ para permitir os deslocamentos de fronteira $\mathbf{U}^{(\beta)}$ com $\mathbf{F}_i = 0$. Dessa forma, a Equação (4.3) pode ser reescrita como:

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_b \\ \mathbf{U}_i \end{bmatrix} = \begin{bmatrix} \mathbf{U}_b^{(\alpha)} \\ \mathbf{U}_i^{(\alpha)} \end{bmatrix} + \begin{bmatrix} \mathbf{U}_b^{(\beta)} \\ \mathbf{U}_i^{(\beta)} \end{bmatrix}. \tag{4.4}$$

Na Equação (4.4) o último termo representa as correções devidas ao relaxamento das fronteiras, no qual, por definição:

$$\mathbf{U}_b^{(\alpha)} = 0. \tag{4.5}$$

De maneira similar, as forças externas representadas em $\mathbf{F}$ podem ser separadas em

$$\mathbf{F} = \mathbf{F}^{(\alpha)} + \mathbf{F}^{(\beta)}, \tag{4.6}$$

ou ainda,

$$\mathbf{F} = \begin{bmatrix} \mathbf{F}_b \\ \mathbf{F}_i \end{bmatrix} = \begin{bmatrix} \mathbf{F}_b^{(\alpha)} \\ \mathbf{F}_i^{(\alpha)} \end{bmatrix} + \begin{bmatrix} \mathbf{F}_b^{(\beta)} \\ \mathbf{F}_i^{(\beta)} \end{bmatrix}, \tag{4.7}$$

onde, por definição, tem-se que

$$\mathbf{F}_i^{(\alpha)} = \mathbf{F}_i \tag{4.8}$$

e

$$\mathbf{F}_i^{(\beta)} = 0. \tag{4.9}$$

Quando as fronteiras da subestrutura estão fixas, pode-se mostrar, utilizando a Equação (4.2), que

$$\mathbf{U}_i^{(\alpha)} = \mathbf{K}_{ii}^{-1}\mathbf{F}_i \tag{4.10}$$

e

$$\mathbf{F}_b^{(\alpha)} = \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{F}_i = \mathbf{R}_b. \tag{4.11}$$

Na Equação (4.11) $\mathbf{F}_b^{(\alpha)}$ representa as reações necessárias para garantir que $\mathbf{U}_b$ seja igual a $\mathbf{0}$, sob a ação das forças internas $\mathbf{F}_i$. Com o relaxamento das fronteiras, os deslocamentos $\mathbf{U}^{(\beta)}$ podem ser determinados pela mesma Equação (4.2) de tal forma que

$$\mathbf{U}_i^{(\beta)} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\mathbf{U}_b^{(\beta)}, \tag{4.12}$$

$$\mathbf{U}_b^{(\beta)} = [\mathbf{K}_{bb} - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}]^{-1}\mathbf{F}_b^{(\beta)}, \tag{4.13}$$

onde

$$\mathbf{K}_b = \mathbf{K}_{bb} - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} \tag{4.14}$$

representa a matriz de rigidez das fronteiras. Com o uso da Equação (4.7) e da Equação (4.11), pode-se determinar

$$\mathbf{F}_b^{(\beta)} = \mathbf{F}_b - \mathbf{F}_b^{(\alpha)} = \mathbf{F}_b - \mathbf{K}_{bi}\mathbf{K}_{ii}^{-1}\mathbf{F}_i = \mathbf{S}_b. \tag{4.15}$$

Com os deslocamentos de fronteira iguais a zero, tem-se as subestruturas isoladas umas das outras. Dessa maneira, a aplicação de uma força interna causa deslocamentos em uma única subestrutura. Os deslocamentos internos com as fronteiras fixas podem ser calculados separadamente para cada subestrutura, utilizando a Equação (4.10). Os deslocamentos de fronteira $\mathbf{U}_b^{(\beta)}$ são calculados com a Equação (4.13), que envolve a inversão de $\mathbf{K}_b$, a qual possui ordem muito menor do que a matriz de rigidez completa $\mathbf{K}$.

Os passos da análise com subestruturação são:

Passo 1 **Discretização do domínio e carregamento das condições de contorno**: etapa de pré-processamento na qual a malha de elementos é obtida e as condições de contorno (restrições e carregamentos) são aplicadas sobre o domínio;

Passo 2 **Particionamento do domínio em n subdomínios**: a malha é particionada em n partes com a utilização de um algoritmo para particionamento de grafos; o domínio agora pode ser visto como sendo composto por n "super-elementos", correspondentes aos subdomínios;

Passo 3 **Computação das contribuições dos elementos**: para que o sistema global possa ser montado, cada subdomínio deve formar sua matriz de rigidez e seu vetor de esforços, independentemente da contribuição dos subdomínios vizinhos;

Passo 4 **Montagem do sistema global equivalente às interfaces**: de posse das contribuições de cada subdomínio, o sistema global pode ser montado;

Passo 5 **Solução do sistema global**: o sistema global é resolvido;

Passo 6 **Atualização dos resultados nos subdomínios**: os subdomínios atualizam suas soluções internas com os resultados recebidos das interfaces.

4.6 Implementação

Nesta seção são apresentadas as características de implementação das classes desenvolvidas para possibilitar a decomposição de domínio e a subestruturação na análise estrutural.

4.6.1 Decomposição do Domínio

Três classes estão diretamente associadas ao particionamento da malha de elementos finitos em OSW: `tDomainPartitioner`, `tGraphPartitioner` e `tMetisPartitioner`. De forma sucinta, a classe `tDomainPartitioner` é o elemento do sistema de particionamento responsável por particionar o domínio do problema. Para tal, ela precisa de um particionador de grafos, representado genericamente pela classe `tGraphPartitioner`. O particionamento efetivo do grafo é realizado por uma classe qualquer derivada de `tGraphPartitioner`, responsável por implementar algum algoritmo de particionamento de grafos. Neste trabalho, uma dessas classes está disponível,

tMetisPartitioner, a qual encapsula as funcionalidades para particionamento de grafos do METIS. O diagrama de classes da Figura 4.5 ilustra as classes envolvidas no particionamento.

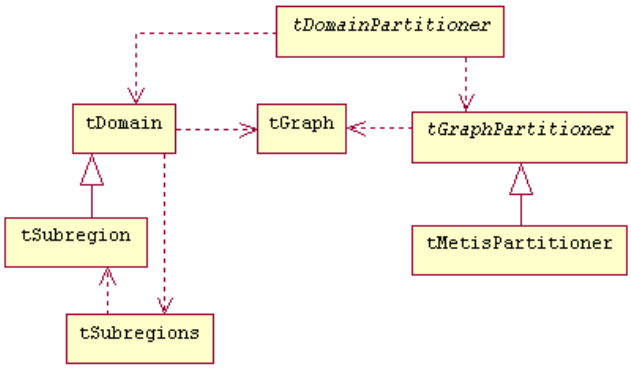


Figura 4.5: Classes relacionadas ao particionamento do domínio.

A classe tDomainPartitioner inicia o particionamento do domínio — um objeto da classe tDomain — com a criação de um grafo dual (tGraph) equivalente ao mesmo, através da chamada ao método CreateDualGraph; o grafo resultante da chamada a este método é representado pelo atributo Graph de tDomainPartitioner. Após a obtenção do grafo dual, cria-se uma instância da classe tGraphPartitioner passando como argumentos o grafo e o número n de partições em que o grafo deve ser dividido. A partir desse ponto, deve-se particionar o grafo através da chamada ao método Execute. Como dito anteriormente, o real particionamento não é realizado por tGraphPartitioner, mas sim por uma classe derivada.

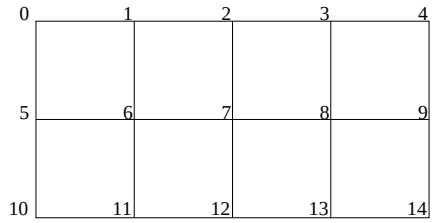
Após o particionamento, o domínio, que inicialmente não possuía partições, conterá uma lista de objetos de tSubregion (classe derivada de tDomain). Cada um desses objetos representa a parte geométrica de um novo subdomínio, sem conter informações específicas necessárias somente à fase de análise. Neste ponto, é importante ressaltar que a decomposição de domínio implementada é *totalmente independente* do método que será utilizado para realizar a análise. Para a fase de análise, precisa-se considerar as subregiões acrescidas de informações necessárias para aplicação do método escolhido; é com essa finalidade que a classe tSubdomain foi implementada. tSubdomain é derivada de tSubregion e de tFiniteElement e possui, dessa maneira, os componentes para realizar a análise através do MEF. A classe tSubdomain é descrita no Capítulo 5, juntamente com as demais classes relacionadas à análise.

Classe tGraph

A classe tGraph (Programa 4.1) representa a estrutura de um grafo, armazenado utilizando a estratégia CSR (*Compressed Storage Row*). No CSR, para armazenar um grafo com NumberOfVertices vértices e NumberOfEdges arestas, são utilizados dois vetores, FirstEdgeIndices e Adjacencies, cujos tamanhos são NumberOfVertices +1 e 2*NumberOfEdges, respectivamente.

Assumindo que a numeração dos vértices do grafo se inicia em zero, a lista de adjacências de um vértice i é armazenada no vetor Adjacencies a partir do índice

`FirstEdgeIndices[i]` até o índice `FirstEdgeIndices[i+1]-1`. Ou seja, para cada vértice i, sua lista de adjacências é armazenada em posições consecutivas do vetor `Adjacencies` e o vetor `FirstEdgeIndices` é utilizado para apontar o início e o final dessa lista. A Figura 4.6 ilustra um grafo simples de 15 vértices e como o mesmo é representado utilizando o esquema CSR. Se o grafo possuir pesos nos vértices, os



FirstEdgeIndices 0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44

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

Figura 4.6: Exemplo da estratégia CSR de armazenamento.

mesmos são armazenados em um vetor denominado `Vwgt`. O tamanho de `Vwgt` é igual a `NumberOfVertices*Ncon`, onde `Ncon` é o número de pesos associados a cada vértice. Os pesos do i-ésimo vértice estão nas `Ncon` entradas consecutivas a partir de `Vwgt[i*Ncon]` Se todos os vértices do grafo possuem o mesmo peso, `Vwgt` pode ser nulo. Os pesos das arestas, se houver, são armazenados no vetor `Adjwgt` cujo tamanho é igual a `2*NumberOfEdges`; o peso da aresta `i` é armazenado em `Adjwgt[i]`. Assim como no caso de pesos de vértices, se todas as arestas possuírem os mesmos pesos, `Adjwgt` pode ser nulo. Outro atributo importante é a variável lógica `ShouldDelete`, a qual indica se o destrutor da classe deve ou não desalocar a memória dos vetores `FirstEdgeIndices` e `Adjacencies`.

Dois construtores estão presentes em `tGraph`, um que recebe como argumentos o número de vértices e de arestas do grafo dual e outro que não recebe argumentos. O primeiro construtor é responsável por alocar a memória necessária para os vetores `FirstEdgeIndices` e `Adjacencies`, por atribuir o valor 1 a `Ncon` e o valor `true` a `ShouldDelete` e por tornar os vetores `Vwgt` e `Adjwgt` nulos, pois neste trabalho são considerados apenas grafos sem pesos nos vértices e arestas. O segundo cria um grafo vazio, não alocando memória para os vetores, pois em algum momento terá os mesmos atribuídos em algum passo dos algoritmos de particionamento; dessa forma, esse construtor apenas atribui o valor `false` para o atributo `ShouldDelete`. O destrutor de `tGraph` é responsável por liberar a memória alocada para os vetores `FirstEdgeIndices` e `Adjacencies`.

Classe `tSubregion`

A classe `tSubregion`, definida no Programa 4.2, representa cada uma das partições que compõem uma malha após o particionamento. `tSubregion` possui como atributos, além dos herdados de `tSubdomain`:

- `tMeshPartition* MeshPartition`: ponteiro para a partição;

```
1   class tGraph
2   {
3      public:
4         int NumberOfVertices;
5         int NumberOfEdges;
6         tIndex* FirstEdgeIndices;
7         tIndex* Adjacencies;
8         tIndex *Vwgt;
9         tIndex *Adjwgt;
10        int Ncon;
11
12        tDualGraph();
13        tDualGraph(int);
14        tDualGraph(int, int);
15     private:
16        bool ShouldDelete;
17  }; // tGraph
```

Programa 4.1: Classe `tGraph`.

- `tMaterial Material`: material de que é composta.

Possui os seguintes métodos principais:

- `GetNumberOfExternalNodes()`: retorna o número de nós externos da subregião, ou seja, o número de nós que pertencem simultaneamente a mais de uma subregião;
- `GetExternalNode(int i)`: retorna o i-ésimo nó externo da subregião;
- `GetMeshPartition()`: retorna um ponteiro para a partição equivalente à subregião;
- `GetNumberOfNodes()`: retorna o número de nós pertencentes à subregião;
- `GetNode(int i)`: retorna um ponteiro para o i-ésimo nó da subregião;
- `GetMaterial()`: retorna o material do qual a subregião é composta;
- `AddExternalNode(tNode*)`: adiciona um ponteiro para nó à lista de nós externos;
- `RemoveNode(int i)`: remove o i-ésimo nó da subregião;
- `AddLoad(tElementLoad* el, double i)`: adiciona o ponteiro para o carregamento `el` na i-ésima posição da lista de carregamentos;
- `Update()`: atualiza todos os elementos do domínio;
- `SetMaterial(const tMaterial& m)`: define `m` como o material componente da subregião;

- `GetExternalNodeIterator()`: retorna um iterador para a coleção de nós externos.

```
1   class tSubregion:  virtual public ISubregion,
2     public tDomainComponent, public tDomain

3   {
4     public:
5       tSubregion(tMeshPartition&, int, int, int, int, int = 1);
6       ~tSubregion();
7       intGetNumberOfExternalNodes() const;
8       tNode* GetExternalNode(int) const;
9       tMeshPartition* GetMeshPartition() const;
10      int GetNumberOfNodes() const;
11      tNode* GetNode(int) const;
12      tMaterial GetMaterial() const;
13      int AddExternalNode(tNode*);
14      tNode* RemoveNode(int);
15      void AddLoad(tElementLoad*, double);
16      void Update();
17      void SetMaterial(const tMaterial&);
18      void ClearAll();
19      tNodeIterator GetExternalNodeIterator() const;
20
21    protected:
22      tNodes ExternalNodes;
23
24    private:
25      tMeshPartition* MeshPartition;
26      tMaterial Material;
27  }; // tSubregion
```

Programa 4.2: Classe `tSubregion`.

Classe `tDomainPartitioner`

O particionamento do domínio do problema é feito por um objeto da classe `tDomainPartitioner`. Ela é responsável por obter o grafo dual correspondente ao domínio, iniciar o particionamento desse grafo e, posteriormente, criar as partições resultantes do particionamento do grafo. A classe utiliza um particionador de grafo, responsável por particionar o grafo equivalente ao domínio, obtido no método `Run()`.

A classe possui os seguintes métodos principais:

- `tDomainPartitioner(tGraphPartitioner& gp)`: construtor que recebe o particionador de grafos como argumento;
- `Run(tDomain& domain, int nop)`: método que particiona o domínio. Seus parâmetros são o domínio a particionar e o número de partições desejadas;

- `CreateSubdomain()`: responsável por criar as partições do domínio obtidas com o particionamento do grafo.

Classe `tGraphPartitioner`

A classe `tGraphPartitioner` é uma classe abstrata cuja finalidade principal é fornecer uma interface básica para as classes que realizam o particionamento de grafos através de diferentes algoritmos. Não é possível criar instâncias dessa classe, apenas de suas derivadas. Atualmente, a única classe derivada disponível é `tMetisPartitioner` (descrita a seguir), entretanto, a criação de uma classe base permite que tenhamos diferentes alternativas para particionar o grafo, o que fornece um ambiente ideal para permitir que o usuário escolha entre essas opções e possa fazer comparações entre as mesmas. A classe possui os seguintes atributos:

- `tGraph* Graph`: ponteiro para o grafo a ser particionado;
- **`int`** `NumberOfPartitions`: número de partições.

A definição parcial da classe `tGraphPartitioner` é apresentada no Programa 4.3.

```
1    class tGraphPartitioner
2    {
3       public:
4          tGraph* Graph;
5          int NumberOfPartitions;
6
7          tGraphPartitioner(tGraph*, int);
8          void Execute();
9          ...
10   }; // tGraphPartitioner
```

Programa 4.3: Classe abstrata `tGraphPartitioner`.

Classe `tMetisPartitioner`

A classe `tMetisPartitioner` é derivada da classe abstrata `tGraphPartitioner`, sendo responsável, portanto, por efetuar o real particionamento de um grafo. Para tal, utiliza os algoritmos de particionamento em k vias multiníveis, apresentados na Seção 4.4. Para utilizar o pacote METIS, foi necessário criar uma classe interna a `tMetisPartitioner`, denominada `tMetisGraph`, que possui como um dos atributos o grafo passado como argumento ao construtor e, ainda, outros específicos do algoritmo de particionamento, cujas descrições das funcionalidades podem ser encontradas em [KAR98a].

O grafo a ser particionado e o número de partições são passados como argumentos ao construtor da classe. Os principais atributos de `tMetisPartitioner` são:

- `tIndex* VertexPartitions`: vetor de tamanho igual ao número de vértices do grafo que armazena o resultado do particionamento. Cada posição do vetor

indica a qual partição determinado vértice foi atribuído, por exemplo, se o vértice i foi atribuído à partição j, tem-se `VertexPartitions[i]=j`;

- **int** `Options`: vetor de cinco posições utilizado para passar informações para diversas fases dos algoritmos. Se `Options[0]=0`, os valores padrão são utilizados. Caso `Options[0]=1`, as quatro demais opções são interpretadas da seguinte maneira:
  - `Options[1]`: Determina qual o algoritmo é utilizado para efetuar o "*matching*" de vértices na etapa de contração:
    1. *Matching* Aleatório (RM);
    2. *Heavy-Edge Matching* (HEM);
    3. *Sorted Heavy-Edge Matching* (SHEM): Opção padrão.
  - `Options[2]`: Determina o algoritmo utilizado no particionamento inicial. Os valores possíveis são:
    1. Crescimento de região: Opção padrão;
  - `Options[3]`: Determina o algoritmo utilizado para o refinamento. Os valores possíveis são:
    1. *Early-Exit Boundary FM Refinement*: Opção padrão;
  - `Options[4]`: Usado para depuração, sempre deve ser igual ao valor padrão 0.
- *EdgeCut*: armazena o número de arestas de corte, após o particionamento;
- *NumFlag*: indica qual esquema de numeração é utilizado no armazenamento do grafo. Assume um dos seguintes valores:
  1. Numeração dos vetores de acordo com a linguagem `C`, ou seja, a partir de 0;
  2. Numeração de acordo com a linguagem Fortran, a partir de 1.
- `WeightFlag`: indica se o grafo possui ou não pesos associados a ele. Seus possíveis valores são:
  1. Não há pesos;
  2. Pesos apenas nas arestas;
  3. Pesos apenas nos vértices;
  4. Pesos nos vértices e nas arestas.

O grafo a ser particionado e o número de partições a serem criadas são especificadas nos dois atributos da classe base.

O construtor da classe atribui o valor 0 a `Options[0]`, `WeightFlag` e `NumFlag`, para indicar, respectivamente, que os algoritmos padrão devem ser utilizados no particionamento, que o grafo não possui pesos e que a numeração dos vetores deve começar em 0. Além disso, como dito anteriormente, o construtor recebe como parâmetros o grafo e o número de partições e os atribui aos seus respectivos atributos. O destrutor da classe é responsável por liberar a memória alocada para o vetor `VertexPartitions`.

Alguns métodos pertencentes a `tGraphPartitioner` constituem apenas o encapsulamento de funções do METIS e suas explicações resultariam em um aprofundamento exagerado em detalhes dos algoritmos de particionamento, por esse motivo não serão aqui apresentados. Além desse métodos, a classe possui ainda mais três:

- `tMetisGraph* GetGraph()`: retorna o atributo `Graph`;
- **`void`** `SetGraph(tMetisGraph*)`: atribui seu parâmetro ao atributo `Graph`;
- **`void`** `Execute`: inicia o particionamento do grafo, chamando os métodos correspondentes a cada algoritmo implementado.

O Programa 4.4 apresenta parcialmente a definição da classe `tMetisPartitioner`, as funções específicas do METIS foram omitidas.

```
1   class tMetisPartitioner:  public tGraphPartitioner
2   {
3      public:
4         tIndex* VertexPartitions;
5         tIndex *Elmnts, *Epart, *Npart;
6         int Options[5];
7         int EdgeCut;
8         int NumFlag;
9         int WeightFlag;
10
11        tGraphPartitioner(tMetisGraph*, int);
12        tMetisGraph* GetGraph();
13        void SetGraph(tMetisGraph*);
14        void Execute();
15        ...
16  }; // tMetisPartitioner
```

Programa 4.4: Classe `tMetisPartitioner`.

Para exemplificar o funcionamento das classes de decomposição do domínio, a Figura 4.7 apresenta o resultado do particionamento em 100 partes do domínio representado por um dragão. A malha possui 54296 vértices, 108588 células e cada cor representa uma partição.

4.7 Análise com Decomposição de Domínio

Com base na subestruturação e na decomposição de domínio pode-se realizar a análise numérica considerando o domínio particionado em diversas partes, o que possibilita redução no tamanho do sistema de equações a serem resolvidas. Para abranger a análise com a utilização dessas técnicas, o modelo foi expandido e as classes adicionadas são descritas ao longo desta seção.

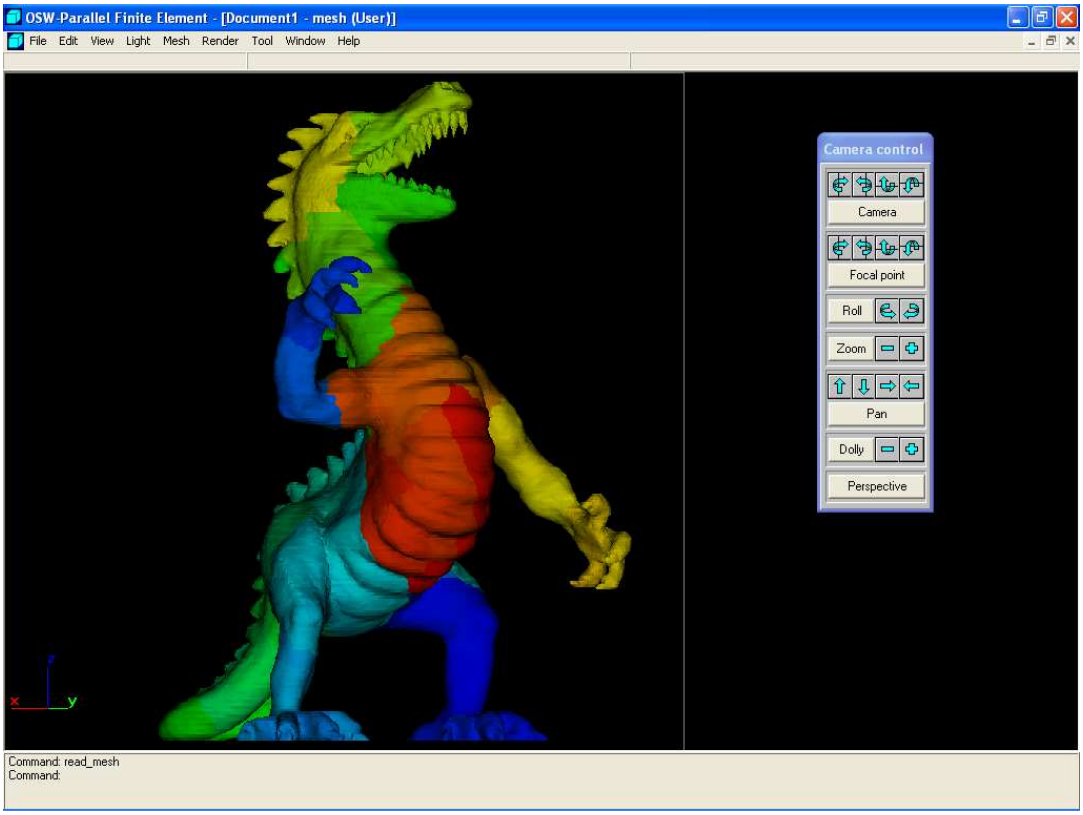


Figura 4.7: Exemplo de domínio particionado.

Classe `tSubdomain`

Um subdomínio, representado por um objeto da classe `tSubdomain` é uma subregião acrescida de atributos e métodos que possibilitam a análise via MEF. Para tal, a classe `tSubdomain` é derivada de `tSubregion` e de `tFiniteElement`.Além do construtor e dos métodos e atributos herdados das classes bases, um subdomínio fornece métodos para computar a matriz de rigidez, matriz de massa e vetor de carga. Na Figura 4.8 é apresentado o diagrama das classes relacionadas a `tSubdomain`.

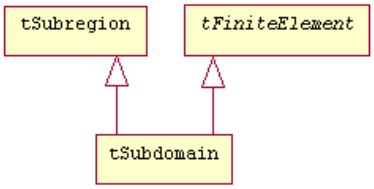


Figura 4.8: Classes relacionadas a `tSubdomain`.

Classe tDomainDecompositionAnalysis

No Capítulo 3 a classe `tAnalysis` foi descrita e mencionou-se que existem classes derivadas concretas responsáveis por fornecer implementações para os métodos nela

definidos, de acordo com o tipo de análise utilizado. Uma dessas classes é `tDomainDecompositionAnalysis`, que representa genericamente as classes que realizam a análise por decomposição de domínio. Essa classe possui como atributo um ponteiro para o subdomínio analisado, e os seguintes métodos:

- `FormTangent()`: responsável por calcular a matriz tangente do subdomínio, através de um objeto de uma das classes derivadas de `tIncrementalIntegrator`;
- `GetTangent()`: retorna a matriz tangente calculada pelo método anterior;
- `FormResidual()`: calcula o vetor de esforços com o auxílio de um objeto de uma classe derivada de `tIncrementalIntegrator`;
- `GetResidual()`: retorna o vetor calculado pelo método anterior;
- `ComputeInternalResponse()`: resolve o sistema para o passo atual da simulação, através de uma chamada ao método `SolveCurrentStep` do objeto `tAlgorithm`.

A definição de `tDomainDecompositionAnalysis` é apresentada no Programa 4.5.

As classes para análise por decomposição de domínio são ilustradas na Figura 4.9.

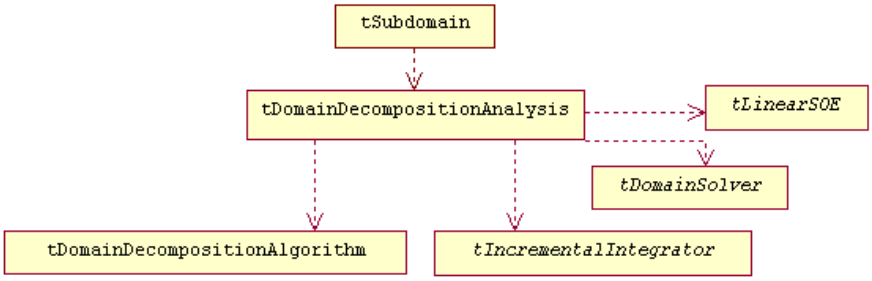


Figura 4.9: Classes de análise por decomposição de domínio.

Classe `tDomainDecompositionAlgorithm`

Um objeto da classe `tDomainDecompositionAlgorithm` é o responsável por coordenar as operações necessárias pelo algoritmo de análise por decomposição de domínio. Essa classe, como as outras classes de algoritmos é derivada de `tAnalysis::tAlgorithm` e é ilustrada na Figura 4.10.

Classe `tDomainSolver`

Classe abstrata que define os métodos responsáveis pela condensação da matriz de rigidez e do vetor de esforços nodais de um subdomínio, bem como os deslocamentos internos a partir dos valores das fronteiras. Possui os seguintes métodos:

- `CondenseA()`: calcula a matriz de rigidez condensada do subdomínio;

```
1   class tDomainDecompositionAnalysis:  public tAnalysis
2   {
3     public:
4       tDomainDecompositionAnalysis(tSubdomain&, tConstraintHandler&,
5         tDOFNumberer&, tDomainDecompositionAlgorithm&, tLinearSOE&,
6         tDomainSolver&, tIncrementalIntegrator&);
7
8       ~tDomainDecompositionAnalysis();
9
10      virtual bool Update();
11      virtual bool ComputeInternalResponse();
12      virtual bool FormTangent();
13      virtual bool FormResidual();
14
15      virtual tMatrix GetTangent();
16      virtual tVector GetResidual();
17
18      long GetNumberOfExternalEquations() const;
19      long GetNumberOfEquations() const;
20      long GetNumberOfInternalEquations() const;
21
22      protected:
23        tSubdomain* GetSubdomain() const;
24        tDomainDecompositionAlgorithm* GetAlgorithm() const;
25        tLinearSOE* GetLinearSOE() const;
26        tDomainSolver* GetDomainSolver() const;
27        tIncrementalIntegrator* GetIntegrator() const;
28
29      private:
30        tSubdomain* Subdomain;
31        tDomainDecompositionAlgorithm* Algorithm;
32        tLinearSOE* LinearSOE;
33        tDomainSolver* Solver;
34        tIncrementalIntegrator* Integrator;
35        long ModifiedTime;
36        long NumberOfExternalEquations;
37        tMatrix Tangent;
38        tVector Residual;
39  }; // tDomainDecompositionAnalysis
```

Programa 4.5: Classe `tDomainDecompositionAnalysis`.

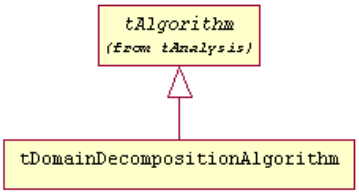


Figura 4.10: Classe `tDomainDecompositionAlgorithm`.

- `CondenseRHS()`: calcula o vetor de esforços nodais condensado do subdomínio;
- `SetComputedExternalX()`: atribui os valores calculados para os nós de interface;
- `SolveInternalX()`: calcula os resultados da análise nos nós internos do subdomínio;
- `GetCondensedA()`: retorna a matriz de rigidez condensada;
- `GetCondensedRHS()`: retorna o vetor de esforços nodais condensado.

4.8 Considerações Finais

O particionamento da malha de elementos finitos influencia de maneira significativa o desempenho de programas paralelos para análise estrutural, pois determina como será balanceada a carga de trabalho entre os processadores. Por esse motivo, a técnica empregada para realizar esse particionamento deve ser escolhida com cuidado. Nesse trabalho, optou-se por utilizar o particionamento de um grafo equivalente à malha através do pacote METIS.

O METIS é um pacote de programas que realiza o particionamento de grafos irregulares, malhas e utilizado também na computação da redução de preenchimento de matrizes esparsas [KAR98a]. Para o particionamento, METIS fornece o algoritmo da bissecção recursiva multinível e o algoritmo para particionamento em k vias multinível. Ambos os algoritmos baseiam-se em três fases: compactação, particionamento inicial e descompactação. Na fase de compactação, o grafo original é reduzido a um grafo com menor número de vértices, através da sucessiva união de alguns vértices adjacentes. Na segunda fase, realiza-se o particionamento do grafo reduzido, enquanto na fase de descompactação o particionamento do grafo reduzido é projetado e refinado progressivamente até atingir o grafo original. A diferença principal entre os dois algoritmos é que enquanto na bissecção o grafo obtido é particionado em apenas duas partes, no particionamento em k vias uma k-partição deve ser obtida de maneira direta.

Neste trabalho utiliza-se o particionamento de grafos implementado em METIS e não o particionamento de malhas. As funcionalidades para o particionamento foram encapsuladas na classe `tMetisPartitioner` e a classe `tDomainPartitioner` é responsável por obter o grafo dual e pela posterior obtenção do particionamento do domínio equivalente ao grafo particionado. Após realizado o particionamento, o domínio contém uma lista de partições, objetos da classe `tSubdomain`.

Além do particionamento da entrada do problema, é necessário utilizar alguma técnica que permita realizar a análise numérica em paralelo de maneira eficaz, como, por exemplo, a subestruturação. Essa técnica permite que o problema se restrinja à análise das fronteiras dos subdomínios.

CAPÍTULO 5

Análise em Paralelo

5.1 Considerações Iniciais

O objetivo principal da utilização de técnicas de paralelização é a obtenção de algoritmos capazes de resolver problemas de maneira mais eficiente, tanto em tempo de execução quanto em aproveitamento dos recursos disponíveis. Como em diversas outras áreas do conhecimento, há também crescente necessidade por alternativas que realizem simulações físicas de maneira eficiente, o que motiva a tentativa de utilizar a computação paralela em mecânica computacional.

A Seção 5.2 apresenta a estratégia implementada neste trabalho para a simulação paralela pelo MEF, com a descrição dos passos envolvidos nessa tarefa. A Seção 5.3 especifica detalhes da implementação das classes necessárias para utilização do paralelismo, com ênfase aos sistemas de serialização, comunicação e às interfaces que permitem que as alterações necessárias para alternar entre a simulação seqüencial e paralela sejam minimizadas.

5.2 Paralelização do MEF

Uma das principais estratégias para paralelização do MEF baseia-se na distribuição, entre os p processadores disponíveis, dos NE elementos finitos resultantes da discretização do domínio do sólido em questão. Com essa distribuição, cada processador conterá, idealmente, cerca de $\frac{\mathrm{NE}}{p}$ elementos finitos a ele atribuídos. Pode-se considerar o conjunto de elementos finitos atribuídos a um processador como um *super-elemento* para o qual o processador pode determinar a matriz de rigidez associada. Ao final dessa etapa, cada processador possuirá parte do sistema de equações global, ao qual faltam as contribuições totais dos nós de intersecção com elementos atribuídos a outros processadores.

Para obter o sistema de equações global as contribuições individuais de cada processador devem ser somadas. Uma alternativa para tal é a execução de uma rodada de comunicação, na qual cada processador enviará para seus vizinhos as matrizes lo-

cais dos elementos finitos nele armazenados, e receberá as matrizes armazenadas em seus vizinhos. Ao final dessa rodada de comunicação, cada processador conterá uma parte do sistema de equações real. Ao invés dos processadores trocarem as matrizes locais de seus super-elementos na rodada de comunicação, pode-se utilizar a chamada *subestruturação*. Com a utilização dessa técnica, o sistema de equações global fica restrito às colaborações dos nós de interface, ou seja, dos nós de intersecção entre os super-elementos. Dessa forma, o sistema global fica significativamente reduzido.

Almeida [ALM99] propõe uma técnica alternativa, denominada *simples particionamento dos graus de liberdade da estrutura*. Essa técnica, baseada em [REZ95], consiste em desenvolver um algoritmo que realize a montagem da matriz de rigidez global linha a linha. Para evitar os conflitos de acesso a uma mesma posição da matriz, propõe-se a sua montagem via abordagem nodal. Para tal, são determinados todos os elementos associados a um mesmo nó, contabilizando as contribuições dos elementos para o nó, montando, assim, a matriz linha a linha. Dessa forma divide-se o domínio dos graus de liberdade e não o domínio físico entre os processadores. Neste trabalho, entretanto, optou-se pela utilização de uma outra técnica, denominada *decomposição de domínio* — abordada no Capítulo 4. Sucintamente, pode-se dizer que, de acordo com essa técnica, a malha de elementos finitos é particionada em p submalhas, com aproximadamente $\frac{\mathrm{NE}}{p}$ elementos finitos cada. Cada uma dessas submalhas é enviada a um processador diferente. Após esse envio, a análise é feita de maneira semelhante à análise seqüencial, detalhada no Capítulo 3.

Os passos necessários para execução de uma simulação paralela — considerando n processadores $(p_0, \cdots, p_{n-1})$ — utilizando decomposição de domínio e subestruturação são:

Passo 1 **Discretização do domínio e carregamento das condições de contorno**: etapa de pré-processamento na qual a malha de elementos é obtida e as condições de contorno são aplicadas sobre o domínio;

Passo 2 **Particionamento do domínio em n subdomínios**: a malha é particionada em n partes com a utilização de um algoritmo para particionamento de grafos; o domínio agora pode ser visto como sendo composto por n "super-elementos", correspondentes aos subdomínios;

Passo 3 **Envio dos subdomínios para os processadores p_1, $\cdots$ e p_{n-1}**: o processador p_0 envia um subdomínio para cada processador p_i, $1 \leq i \leq n-1$. p_0, por sua vez, fica responsável pela computação do sistema global contendo as contribuições apenas dos nós de interface;

Passo 4 **Computação das contribuições dos elementos**: para que o sistema global possa ser montado por p_0, cada subdomínio deve formar sua matriz de rigidez e seu vetor de esforços, independentemente da contribuição dos subdomínios vizinhos, e enviar para p_0;

Passo 5 **Montagem do sistema global equivalente às interfaces**: de posse das contribuições de cada subdomínio, o sistema global pode ser montado;

Passo 6 **Solução do sistema global**: o sistema global é resolvido;

Passo 7 **Envio dos resultados do sistema global para os demais processadores**: os resultados obtidos nas interfaces pelo processador p_0 são enviados para os demais processadores;

Passo 8 **Atualização dos resultados nos subdomínios**: os processadores atualizam suas soluções com os resultados recebidos de p_0.

5.3 Implementação

A análise estrutural paralela via MEF utiliza-se das técnicas de decomposição do domínio e de subestruturação, apresentadas no Capítulo 4. A adaptação da subestruturação à computação paralela, bem como a utilização do paradigma da orientação a objetos, entretanto, não é uma tarefa simples, pois exige que informações sejam trocadas entre diferentes processadores. Para contornar essa dificuldade, foi necessário desenvolver uma estrutura complexa de classes para possibilitar a comunicação com a maior transparência possível, permitindo que a alteração do código de análise seqüencial para análise paralela seja mínima.

5.3.1 Interfaces

Uma das primeiras alterações necessárias à infra-estrutura de classes apresentada nos capítulos anteriores foi o acréscimo de *interfaces*. As interfaces são classes que definem apenas a assinatura dos métodos que devem obrigatoriamente ser implementados pelas classes concretas que delas descendem. Com a utilização dessas classes de interface é torna-se possível um "espelhamento" local de classes que estão realmente armazenadas remotamente, desde que o "espelho" local e o objeto real remoto sejam ambos de classes derivadas da mesma interface. Os objetos local e remoto fornecem implementações aos métodos definidos na interface, porém, no objeto local, tais implementações são apenas chamadas aos métodos implementados no objeto remoto; essas chamadas são feitas utilizando diretivas de comunicação. As classes de interface implementadas são todas derivadas, direta ou indiretamente, de `tSerializable`, o que indica que objetos de todas as classes derivadas dessas interfaces podem ser transformadas em um fluxo contíguo de bytes possibilitando seu envio através de alguma diretiva de comunicação. Essas classes são apresentadas ao longo desta subseção.

Classe `IDomain`

A classe `IDomain`, definida no Programa 5.1, define os métodos que devem ser implementados por qualquer domínio. Os principais são:

- `AddElement(IElement* e)`: adiciona o elemento `e` ao domínio;
- `AddNode(tNode* n)`: adiciona o nó `n` ao domínio;
- `AddSPConstraint(tSPConstraint* sp)`: adiciona a restrição `sp` ao domínio;
- `AddLoadPattern(tLoadPattern* lp)`: adiciona o caso de carregamento `lp` ao domínio;

- `GetNumberOfElements()`: retorna o número de elementos do domínio;
- `GetNumberOfNodes()`: retorna o número de nós do domínio;
- `GetNumberOfSPConstraints()`: retorna o número de restrições do domínio;
- `GetNumberOfLoadPatterns()`: retorna o número de carregamentos do domínio;
- `GetElement(int i)`: retorna o i-ésimo elemento do domínio;
- `GetNode(int i)`: retorna o i-ésimo nó do domínio;
- `virtual tSPConstraint* GetSPConstraint(int i)`: retorna a i-ésima restrição do domínio;
- `GetLoadPattern(int i)`: retorna o i-ésimo carregamento do domínio;
- `RemoveElement(int i)`: remove o i-ésimo elemento do domínio;
- `RemoveNode(int i)`: remove o i-ésimo nó do domínio;
- `RemoveSPConstraint(int i)`: remove a i-ésima restrição do domínio;
- `RemoveLoadPattern(int i)`: remove o i-ésimo carregamento do domínio;
- `ApplyLoad(double load)`: aplica a carga `load` ao domínio;
- `GetElementGraph()`: retorna o grafo equivalente aos elementos do domínio;
- `GetModifiedTime()`: retorna o tempo da última modificação do domínio;
- `GetMesh()`: retorna a malha equivalente ao domínio;
- `GetNumberOfEquations()`: retorna o número de equações do domínio;

Classe `IDomainComponet`

A classe `IDomainComponent`, apresentada no Programa 5.2, especifica a interface que deve ser implementada por todo componente de domínio. Define os seguintes métodos:

- `GetDomain()`: retorna o domínio ao qual o componente pertence;
- `GetNumber()`: retorna o número que identifica o componente;
- `SetDomain(tDomain* d)`: define o atributo `d` como o domínio ao qual o componente pertence;
- `SetNumber(int i)`; define `i` como o número do componente;

```
1   class IDomain:  virtual public tSerializable
2   {
3     public:
4       virtual int AddElement(IElement*) = 0;
5       virtual int AddNode(tNode*) = 0;
6       virtual int AddSPConstraint(tSPConstraint*) = 0;
7       virtual int AddLoadPattern(tLoadPattern*) = 0;
8
9       virtual int GetNumberOfElements() const = 0;
10      virtual int GetNumberOfNodes() const = 0;
11      virtual int GetNumberOfSPConstraints() const = 0;
12      virtual int GetNumberOfLoadPatterns() const = 0;
13
14      virtual IElement* GetElement(int) const = 0;
15      virtual tNode* GetNode(int) const = 0;
16      virtual tSPConstraint* GetSPConstraint(int) const = 0;
17      virtual tLoadPattern* GetLoadPattern(int) const = 0;
18
19      virtual IElement* RemoveElement(int) = 0;
20      virtual tNode* RemoveNode(int) = 0;
21      virtual tSPConstraint* RemoveSPConstraint(int) = 0;
22      virtual tLoadPattern* RemoveLoadPattern(int) = 0;
23
24      virtual void ClearAll() = 0;
25      virtual void ApplyLoad(double) = 0;
26      virtual void Update() = 0;
27
28      virtual tGraph* GetElementGraph() const = 0;
29      virtual long GetModifiedTime() const = 0;
30      virtual tMesh* GetMesh() const = 0;
31      virtual long GetNumberOfEquations() const = 0;
32  }; // IDomain
```

Programa 5.1: Classe `IDomain`.

```
1   class IDomainComponent:  virtual public tSerializable
2   {
3     public:
4       virtual ~IDomainComponent();
5
6       virtual tDomain* GetDomain() const = 0;
7       virtual int GetNumber() const = 0;
8
9       virtual void SetDomain(tDomain*) = 0;
10      virtual void SetNumber(int) = 0;
11      virtual void ClearId() = 0;
12  }; // IDomainComponent
```

Programa 5.2: Classe `IDomainComponent`.

Classe `IElement`

A classe `IElement` define a interface comum a todos os elementos e é definida no Programa 5.3. Possui os seguintes métodos principais:

- `GetNumberOfExternalNodes()`: retorna o número de nós externos do elemento;
- `GetExternalNode(int i)`: retorna o i-ésimo nó externo do elemento;
- `GetMaterial()`: retorna o material de que é composto o elemento;
- `AddLoad(tElementLoad* el, double d)`: adicionana um carregamento a um elemento;
- `SetMaterial(const tMaterial& m)`: define o material de que é composto o elemento.

```
1   class IElement:  virtual public IDomainComponent
2   {
3     public:
4       virtual int GetNumberOfExternalNodes() const = 0;
5       virtual tNode* GetExternalNode(int) const = 0;
6       virtual tMaterial GetMaterial() const = 0;
7
8       virtual void AddLoad(tElementLoad*, double) = 0;
9       virtual void Update() = 0;
10      virtual void SetMaterial(const tMaterial&) = 0;
11  }; // IElement
```

Programa 5.3: Classe `IElement`.

Classe `ISubregion`

Os métodos que precisam ser implementados por toda classe que representa uma subregião são definidos na classe `ISubregion`, apresentada no Programa 5.4. `ISubregion` é derivada da classe `IElement`, além dos métodos herdados de sua classe base, possui os seguintes:

- `GetMeshPartition()`: retorna a partição equivalente à subregião;
- `AddExternalNode(tNode*)`: adiciona um ponteiro para nó à coleção de nós externos da subregião.

```
1   class ISubregion:  virtual public IElement, virtual public IDomain
2   {
3     public:
4       virtual int GetNumberOfExternalNodes() const = 0;
5       virtual tNode* GetExternalNode(int) const = 0;
6       virtual tMeshPartition* GetMeshPartition() const = 0;
7
8       virtual int AddExternalNode(tNode*) = 0;
9   }; // ISubregion
```

Programa 5.4: Classe `ISubregion`.

Classe `IFiniteElement`

A classe `IFiniteElement`, derivada de `IElement`, define a interface que deve ser fornecida por todas as classes que implementam funcionalidades de elementos finitos. Adicionalmente aos fornecidos pela classe base, define os seguintes métodos:

- `FormTangent(tIntegrator*)`: calcula a matriz tangente do elemento finito;
- `FormResidual(tIntegrator*)`: calcula o vetor residual do elemento finito;
- `GetTangent()`: retorna a matriz tangente calculada por `FormTangent()`;
- `GetLoadVector()`: retorna o vetor de carregamentos do elemento finito;
- `GetResidual()`: retorna o vetor residual calculado por `FormResidual()`;
- `GetLastResponse()`: retorna o último resultado calculado;
- `GetLocationArray()`: retorna o vetor de localização do elemento finito;
- `ComputeStiffnessMatrix()`: calcula e retorna a matriz de rigidez do elemento finito;
- `ComputeMassMatrix()`: calcula e retorna a matriz de massa do elemento finito;
- **`virtual`** `tMatrix ComputeDampingMatrix()`: calcula e retorna a matriz de amortecimento do elemento finito.

```
1    class IFiniteElement:  virtual public IElement
2    {
3       public:
4          virtual bool FormTangent(tIntegrator*) = 0;
5          virtual bool FormResidual(tIntegrator*) = 0;
6
7          virtual tMatrix GetTangent() const = 0;
8          virtual tVector GetLoadVector() const = 0;
9          virtual tVector GetResidual() const = 0;
10         virtual tVector GetLastResponse() const = 0;
11
12         virtual const tLocationArray& GetLocationArray() = 0;
13         virtual tMatrix ComputeStiffnessMatrix() = 0;
14         virtual tMatrix ComputeMassMatrix() = 0;
15         virtual tMatrix ComputeDampingMatrix() = 0;
16   }; // IFiniteElement
```

Programa 5.5: Classe `IFiniteElement`.

5.3.2 Serialização de Objetos

O processamento paralelo ou distribuído freqüentemente implica na necessidade de compartilhamento de dados, entretanto, em grande parte das vezes, as máquinas envolvidas neste processamento não compartilham memória física. Nesse contexto, para que haja o compartilhamento de dados, é necessário que haja um sistema de comunicação que permita o envio e recebimento de dados. Existem diversas bibliotecas de comunicação que podem ser utilizadas com esse propósito, tais como MPI (*Message Passing Interface*) e PVM (*Parallel Virtual Machine*); essas bibliotecas fornecem diretivas simples para envio e recebimento de dados, entre outras funcionalidades. Quando o paradigma da programação orientada a objetos é utilizado, contudo, surge uma dificuldade adicional: dado que as bibliotecas citadas possibilitam apenas a manipulação de dados armazenados contiguamente na memória, objetos de uma classe nem sempre podem ser enviados de maneira imediata com a utilização das diretivas fornecidas por tais bibliotecas. Para contornar essa dificuldade, um sistema de "serialização de objetos" foi implementado, o qual é capaz de transformar objetos em um fluxo contíguo de bytes que, por sua vez, pode ser enviado.

Um fluxo de bytes é uma seqüência de bytes armazenados na memória de forma contígua, tal como um vetor. Esse fluxo representa, dessa maneira, um conjunto de dados que foram compactados de forma a poderem ser utilizados por diretivas de comunicação, por exemplo.

Em uma comunicação, a serialização é o processo responsável por criar um fluxo de bytes equivalente a um objeto no processo de origem e, ainda, por reconstruir tal objeto no processo de destino. Neste trabalho, esse sistema de serialização foi implementado de maneira a facilitar a escrita de aplicações paralelas, pois permite que o programador não tenha que se preocupar com detalhes da implementação desse sistema para poder utilizá-lo.

Todas as classes cujos objetos podem ser serializáveis derivam, direta ou indire-

tamente, de `tSerializable`. Essa classe não realiza nenhuma operação importante, podendo ser vista como uma "*tag*" sinalizadora da possibilidade de serialização de uma classe. Toda classe serializável deve possuir um "*streamer*" correspondente — um objeto derivado, direta ou indiretamente, de `tStreamer`. O *streamer* é o responsável por "ler" e "escrever" no fluxo e deve estar presente em todas as classes serializáveis; é o *streamer* que realmente torna a serialização possível.

Além de derivar de `tSerializable` toda classe serializável (`cls`) deve incluir a seguinte macro em sua definição:

```
DECLARE_SERIALIZABLE(cls);
```

Para exemplificar, considera-se a declaração da classe `tDomain` no Programa 5.6:

```
1    class tDomain:  public virtual tSerializable, public virtual IDomain
2    {
3       ...
4       DECLARE_SERIALIZABLE(tDomain);
5       ...
6    }; // tDomain
```

Programa 5.6: Classe `tDomain`.

Neste caso, a macro declara uma classe *streamer* (derivada de `tStreamer`) dentro de `tDomain` e um construtor utilizado pelo sistema de serialização para criar uma instância dessa classe para realizar a operação de leitura.

A implementação do *streamer* é realizada em dois passos:

Passo 1 Utilizar a macro `IMPLEMENT_SERIALIZABLE(cls)` para implementar os métodos declarados por `DECLARE_SERIALIZABLE`;

Passo 2 Definir os seguintes métodos:

- `tSerializable* cls::Streamer::Read(tObjectInputStream&)`: o argumento representa o fluxo de objetos lido a partir da entrada de dados e o valor de retorno é um ponteiro para o objeto construído a partir do fluxo de dados recebido como argumento;
- **void** `cls::Streamer::Write(tObjectOutputStream&)`: o argumento é o fluxo de saída que deve ser escrito pelo *streamer*.

Além das macros apresentadas, há ainda outras para declaração e implementação de serialização em classes abstratas ou com múltiplas classes bases virtuais, entretanto, tais classes não serão aqui descritas.

Um fluxo de saída de objetos — representado por um objeto da classe `tObjectOutputStream`, derivado de `tOutputStream` — é construído encadeado a outro fluxo de saída. Um objeto de `tObjectOutputStream` é responsável por construir o fluxo de bytes equivalente ao objeto a ser serializado. O outro fluxo de saída é responsável por efetivamente escrever o fluxo na saída. Essa característica de aninhar um fluxo de saída

em outro possibilita um "encadeamento de funcionalidades". Esse encadeamento é possível porque o construtor da classe `tObjectOutputStream` recebe como parâmetro um ponteiro para um objeto da classe `tOutputStream` que, por sua vez, é base de uma ampla hierarquia de classes, o que permite que ponteiros para objetos de qualquer uma das classes derivadas, direta ou indiretamente, possam ser passados como parâmetros para esse construtor. Os programas 5.7 e 5.8 ilustram, respectivamente, as definições das classes `tObjectOutputStream` e `tOutputStream`.

```
1   class tObjectOutputStream:  public tDataOutputStream
2   {
3     public:
4       tObjectOutputStream(tInputStream*);
5
6       void Flush();
7       void WriteObject(const tSerializable&);
8       void WriteObject(const tSerializable*);
9       void WriteVirtualBaseObject(const tStreamer&);
10      void WriteBaseObject(const tStreamer&);
11
12      tObjectOutputStream& operator <<(char);
13      tObjectOutputStream& operator <<(unsigned char);
14      tObjectOutputStream& operator <<(short);
15      tObjectOutputStream& operator <<(unsigned short);
16      tObjectOutputStream& operator <<(int);
17      tObjectOutputStream& operator <<(unsigned int);
18      tObjectOutputStream& operator <<(long);
19      tObjectOutputStream& operator <<(unsigned long);
20      tObjectOutputStream& operator <<(float);
21      tObjectOutputStream& operator <<(double);
22      tObjectOutputStream& operator <<(bool);
23      tObjectOutputStream& operator <<(const char*);
24      tObjectOutputStream& operator <<(const tSerializable&);
25      tObjectOutputStream& operator <<(const tSerializable*);
26      ...
27  }; // tObjectOutputStream
```

Programa 5.7: Classe `tObjectOutputStream`.

Para manter informações sobre quais objetos já foram escritos, um fluxo de saída de objetos possui uma tabela *hash* que armazena os endereços de todos os objetos escritos no fluxo de bytes, bem como seus respectivos deslocamentos com relação ao início do mesmo; de posse dessas informações, evita-se a replicação de dados dentro do fluxo. Quando uma referência a um objeto precisa ser escrita no fluxo, o objeto referenciado também precisa ser escrito, entretanto, antes de escrever esse objeto, primeiramente seu endereço é procurado na tabela *hash*, se não for encontrado, ele, então, é escrito; caso contrário, o objeto foi escrito anteriormente e uma referência ao seu deslocamento é inserida no fluxo.

De maneira similar, um fluxo de entrada de objetos — representado por um ob-

```
1    class tOutputStream
2    {
3       public:
4          virtual ~tOutputStream();
5
6          virtual void Close() = 0;
7          virtual void Flush() = 0;
8          virtual void Write(const void, long) = 0;
9    }; // tOutputStream
```

Programa 5.8: Classe `tOutputStream`.

jeto da classe `tObjectInputStream` — é construído através de um encadeamento a outro fluxo de entrada. Esse último efetivamente realiza a operação de leitura, enquanto o primeiro interpreta os bytes lidos e reconstrói o objeto original. Os programas 5.9 e 5.10 ilustram parcialmente as definições das classes `tObjectInputStream` e `tInputStream`.

Um fluxo de objetos de entrada também precisa ter algum meio para armazenar as informações relacionadas aos dados do fluxo lido para, assim, possibilitar a reconstrução correta dos objetos representados por esse fluxo. Com esse propósito, utilizou-se uma tabela *hash* para armazenar o endereço de cada novo objeto que precisar ser construído, bem como o seu deslocamento no fluxo de origem. Como descrito anteriormente nesta mesma seção, um fluxo de saída de objetos constrói o fluxo de bytes utilizando endereços de deslocamento para representar objetos previamente escritos. Com base nessa informação, no momento da leitura do fluxo, se uma referência a um objeto é encontrada, há duas possibilidades:

- A referência está acompanhada pelos atributos do objeto. Nesse caso, um novo objeto é instanciado e esses atributos são inseridos no mesmo. O endereço do objeto criado e seu deslocamento no fluxo de origem são inseridos na tabela *hash*;
- A referência indica um objeto previamente lido do fluxo e, conseqüentemente, já instanciado. Nesse caso, o deslocamento lido é buscado na tabela e a referência é atualizada no objeto que a possui com o endereço físico real do objeto.

A Figura 5.1 ilustra o diagrama das classes envolvidas nos fluxos de entrada e saída. As classes `tChannel` e `tMPIChannel` que aparecem no diagrama são utilizadas para comunicação e são abordadas na Seção 5.3.3.

A classe responsável por realizar a escrita do fluxo de bytes é `tOutputStream`; uma classe base abstrata que contém três métodos virtuais puros, que definem o comportamento padrão de todas suas classes derivadas:

- `Close() = 0`: método que fecha o fluxo, após o término da escrita;
- `Flush() = 0`: método responsável por esvaziar o fluxo;
- `Write(const void*, long) = 0`: escreve os dados passados como parâmetro na saída.

```
1   class tObjectInputStream:  public tDataInputStream
2   {
3     public:
4       tObjectInputStream(tInputStream*);
5
6       tSerializable* ReadObject(tSerializable&);
7       tSerializable* ReadObject();
8       tSerializable* ReadVirtualBaseObject(const tStreamer&);
9       tSerializable* ReadBaseObject(const tStreamer&);
10
11      tObjectInputStream& operator >>(char&);
12      tObjectInputStream& operator >>(unsigned char&);
13      tObjectInputStream& operator >>(short&);
14      tObjectInputStream& operator >>(unsigned short&);
15      tObjectInputStream& operator >>(int&);
16      tObjectInputStream& operator >>(unsigned int&);
17      tObjectInputStream& operator >>(long&);
18      tObjectInputStream& operator >>(unsigned long&);
19      tObjectInputStream& operator >>(float&);
20      tObjectInputStream& operator >>(double&);
21      tObjectInputStream& operator >>(bool&);
22      tObjectInputStream& operator >>(char*&);
23      ...
24  }; // tObjectInputStream
```

Programa 5.9: Classe `tObjectInputStream`.

```
1   class tInputStream
2   {
3     public:
4       virtual ~tInputStream();
5
6       virtual long Available() const = 0;
7       virtual void Close() = 0;
8       virtual long Read(void, long) = 0;
9       virtual long Skip(long) = 0;
10  }; // tInputStream
```

Programa 5.10: Classe `tInputStream`.

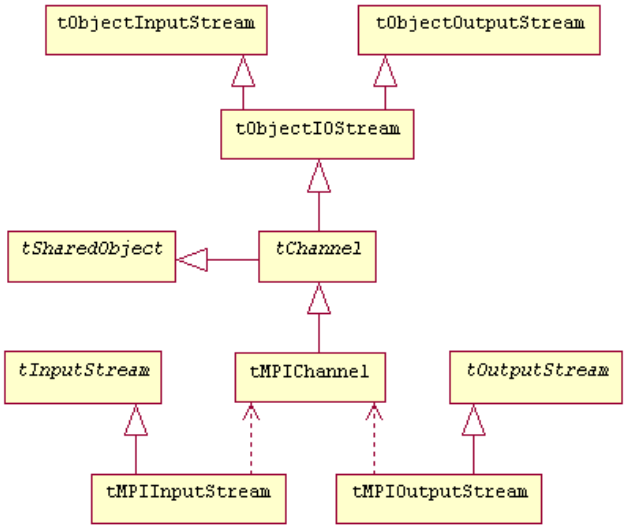


Figura 5.1: Fluxos de entrada e saída.

Como essa classe é abstrata, foi necessário que classes derivadas concretas fossem escritas para implementar os métodos definidos em `tOutputStream` e, dessa maneira, definir como essas operações devem ser executadas dependendo da saída para a qual o fluxo deve ser escrito — arquivo, rede (utilizando alguma biblioteca de comunicação), entre outros. Para exemplificar, a classe `tMPIOutputStream` (Programa 5.11) define como um fluxo de bytes é escrito, ou seja enviado, com a utilização de diretivas MPI.

O construtor de `tMPIOutputStream` recebe como argumento um inteiro que especifica o identificador único (*rank*) do processador de destino e apenas atribui esse valor ao atributo `Destination` e o valor zero a `Tag`. A implementação fornecida ao método `Write()` apenas invoca `MPI_Send()`, diretiva MPI para envio de um *buffer* de dados.

A classe `tInputStream` realiza o processo contrário de `tOutputStream`, ou seja, lê o fluxo de bytes da entrada e é definida como uma classe base abstrata com quatro métodos virtuais puros, cujos principais são:

- `Close() = 0`: fecha o fluxo, após o término da leitura;
- `Read(void*, long) = 0`: lê o fluxo de dados da entrada e armazena no *buffer* especificado como argumento. Retorna o número de bytes efetivamente lidos.

A implementação desses métodos, assim como em `tOutputStream`, é definida em classes derivadas como, por exemplo, a classe `tMPIInputStream` (ilustrada no Programa 5.12) que define como é realizada a leitura, ou seja, o recebimento de dados, com a utilização de diretivas MPI. `tMPIInputStream` atribui valores iniciais ao identificador do processo de origem e à *tag* em seu construtor; e a implementação dada ao método `Read` consiste apenas na chamada às diretivas `MPI_Recv` — para receber o fluxo — e `MPI_Get_count` — para obter o número de dados recebidos.

```
1   class tMPIOutputStream:  public tOutputStream
2   {
3      public:
4         tMPIOutputStream(int destination)
5         {
6            Destination = destination;
7            Tag = 0;
8         }
9
10        void Close(){}
11        void Flush(){}
12        void Write(const void* buf, long len)
13        {
14           MPI_Send(const_cast<void*>(buf), len, MPI_BYTE,
15              Destination, Tag, MPI_COMM_WORLD);
16        }
17
18     private:
19        int Destination;
20        int Tag;
21        tFile File;
22  }; // tMPIOutputStream
```

Programa 5.11: Classe `tMPIOutputStream`.

5.3.3 Comunicação

O esquema de comunicação implementado neste trabalho se baseia no modelo de atores de computação paralela [AGH85], também utilizado por McKenna [MCK97]. A característica fundamental nessa implementação é a ênfase à transparência da comunicação, permitida pelo esquema de serialização descrito na Seção 5.3.2.

A comunicação entre diferentes processos é realizada através de troca de mensagens entre *stubs*. Tais *stubs* são implementados pela classe `tStub` e suas derivadas. Duas dessas derivadas são `tLocalStub` e `tRemoteStub`. A classe `tLocalStub` define um *stub* local, responsável por se comunicar com o *stub* remoto, esse, por sua vez, efetivamente realiza em um processo remoto todas as operações solicitadas. As classes `tStub`, `tLocalStub` e `tRemoteStub` são definidas pelos Programas 5.13, 5.14 e 5.15, respectivamente.

O único atributo da classe `tStub` é um ponteiro para um objeto `tChannel`, que representa o canal de comunicação através do qual os *stubs* remotos e locais trocam mensagens. É esse canal que define para quem o *stub* local deve enviar suas mensagens e de quem o *stub* remoto deve receber mensagens. Tal canal é definido no momento da criação de um objeto de uma das classes, como pode ser visto claramente pelos construtores de `tStub` e `tRemoteStub`, que recebem uma referência a um objeto `tChannel` como parâmetro. O construtor de `tLocalStub`, por sua vez, recebe uma cadeia de caracteres como parâmetro, que indica o nome da classe do objeto remoto com o qual o *stub* precisa se comunicar. Com base nesse nome, o construtor invoca o método

```
1   class tMPIInputStream:  public tInputStream
2   {
3      public:
4         tMPIInputStream(int source)
5         {
6            Source = source;
7            Tag = 0;
8         }
9
10        long Available() const {return 0;}
11        void Close();
12        long Read(void* buf, long len)
13        {
14           MPI_Recv(buf, len, MPI_BYTE, Source, Tag,
15              MPI_COMM_WORLD, &Status);
16           MPI_Get_count(&Status, MPI_BYTE, &var);
17           return var;
18        }
19        long Skip(long) {return 0;}
20
21     private:
22        int Source;
23        MPI_Status Status;
24        int Tag;
25  }; // tMPIInputStream
```

Programa 5.12: Classe `tMPIInputStream`.

`CreateRemoteObject()` passando esse nome como parâmetro. Esse método cria o *stub* remoto e retorna um ponteiro para o canal de comunicação criado entre o *stub* local e o remoto recém criado. Para realizar essa tarefa, tal método utiliza um objeto de uma classe derivada de `tMachineBroker`. Tal objeto é capaz de encontrar um processo remoto disponível, criar um *stub* remoto nesse processo e retornar o canal de comunicação para esse *stub*. As definições das classes `tChannel` e `tMachineBroker` são apresentadas nos Programas 5.16 e 5.17.

O construtor da classe `tChannel`, por sua vez, recebe como argumentos apenas dois ponteiros para fluxos, um de entrada e outro de saída. Esses fluxos são os mecanismos utilizados para armazenar as mensagens a serem enviadas e recebidas. Como as informações acerca da criação de processos e de comunicação são dependentes da plataforma utilizada, para possibilitar o paralelismo é necessário implementar classes derivadas de `tChannel` e de `tMachineBroker`. Neste trabalho foram implementadas classes para permitir a utilização de MPI; a classe `tMPIChannel` (Programa 5.18) define um canal de comunicação para MPI e a classe `tMachineBroker` (Programa 5.19) permite a criação de *stubs* e de canais `tMPIChannel`.

O diagrama de classes da Figura 5.2 ilustra o relacionamento das classes envolvidas na comunicação e serialização.

Como visto anteriormente, quando a subestruturação é utilizada na análise seqüen-

```
1   class tStub
2   {
3      public:
4         enum
5         {
6            DESTROY_REMOTE_OBJECT,
7            LastMessage
8         }
9
10        tStub(tChannel&);
11
12        virtual~tStub();
13
14        tChannel* GetChannel() const
15        {
16           return Channel;
17        }
18     protected:
19        tChannel* Channel;
20  }; // tStub
```

Programa 5.13: Classe `tStub`.

```
1   class tLocalStub
2   {
3      public:
4         tLocalStub(const char* remoteObjectClassName):
5            tStub(*CreateRemoteObject(remoteObjectClassName))
6         {}
7
8         virtual ~tLocalStub();
9
10        void SendMessage(int msg)
11        {
12           *Channel << msg;
13        }
14     private:
15        static tMachineBroker* MB;
16        static tChannel* CreateRemoteObject(const char*);
17
18        friend class tMachineBroker;
19  }; // tLocalStub
```

Programa 5.14: Classe `tLocalStub`.

```
1    class tRemoteStub
2    {
3       public:
4          tRemoteStub(tChannel& channel):
5             tStub(channel), Terminated(false)
6          {}
7
8          void Terminate()
9          {
10            Terminated = true;
11         }
12
13         virtual void Run();
14      protected:
15         virtual void Dispatch(int) = 0;
16      private:
17         bool Terminated;
18   }; // tRemoteStub
```

Programa 5.15: Classe `tRemoteStub`.

```
1    class tChannel:  public tObjectIOStream, public tSharedObject
2    {
3       protected:
4          tChannel(tInputStream* is, tOutputStream* os):
5             tObjectIOStream(is, os)
6          {}
7    }; // tChannel
```

Programa 5.16: Classe `tChannel`.

cial, um subdomínio precisa de dados localizados em outro subdomínio. Para manter a transparência e minimizar as diferenças entre análise seqüencial e paralela, é necessário que as operações continuem considerando subdomínios, sem importar se todos esses subdomínios estão ou não armazenados localmente. Para permitir a utilização dos *stubs* sem ser necessário modificar todo o código da simulação com a introdução de chamadas a métodos de envio e recepção de mensagens, foram definidas classes que representam tanto domínios ou subdomínios quanto *stubs*, reunindo as funcionalidades das classes `tStub` — ou de uma de suas derivadas — e `IDomain` ou `ISubdomain`, respectivamente. Isso foi possível devido ao mecanismo de herança múltipla fornecido por C++.

Baseado nessa implementação, o código para realização da análise continua o mesmo, um subdomínio pode continuar solicitando dados de outros subdomínios mas, quando o paralelismo é utilizado, esses subdomínios são objetos de `tLocalStubSubdomain` ou de `tRemoteStubSubdomain`. A definição parcial de `tLocalStubSubdomain` é ilustrada no Programa 5.20.

Um objeto da classe `tLocalStubSubdomain` é responsável por instanciar um objeto

```
1   class tMachineBroker
2   {
3      public:
4         tMachineBroker();
5
6         virtual ~tMachineBroker();
7
8         virtual int GetNumberOfRemoteProcesses() const = 0;
9         virtual int GetPID() const = 0;
10        virtual tChannel* CreateRemoteObject(const char*);
11        virtual tRemoteStub* BuildRemoteStub(const char*, tChannel&);
12
13        virtual tChannel* GetRemoteProcess() = 0;
14        virtual void FreeRemoteProcess(tChannel*) = 0;
15     protected:
16        static void SetLocalStubMB(tMachineBroker*);
17  }; // tMachineBroker
```

Programa 5.17: Classe `tMachineBroker`.

```
1   class tMPIChannel:  public tChannel
2   {
3      protected:
4         tMPIChannel(int pid):
5            tChannel(new tMPIInputStream(pid),
6               new tMPIOutputStream(pid))
7         {}
8   }; // tMPIChannel
```

Programa 5.18: Classe `tMPIChannel`.

da classe `tSubdomain` em um `tRemoteStubSubdomain`, essa tarefa é realizada dentro de seu próprio construtor, linhas 5-11 do Programa 5.20, através do envio de uma mensagem ao *stub* remoto através do canal de comunicação que os conecta. Nessa mensagem o *stub* local especifica o método que deve ser executado (neste caso, `CONSTRUCTOR`) e os parâmetros necessários para a invocação remota do método. Após a criação do subdomínio remoto, elementos e nós podem ser adicionados a ele, o que também é realizado através de chamadas a métodos no *stub* local (linhas 13-21). A definição parcial de um *stub* remoto é feita através da macro apresentada no Programa 5.21.

Essa macro define o comportamento do *stub* remoto quando recebe uma mensagem que requer que determinada ação seja executada. A macro define qual método do subdomínio deve ser chamado de acordo com o identificador recebido na mensagem enviada por um objeto `tLocalStubSubdomain`. Dessa forma, deve ficar claro que quando uma solicitação é feita a um objeto da classe `tLocalStubSubdomain`, ela é realmente satisfeita em um subdomínio remoto. O `tLocalStubSubdomain` não armazena as informações necessárias para processar as solicitações localmente, mas o subdomínio que solicitou não tem conhecimento de todo o mecanismo envolvido no

```
1    class tMPIMachineBroker:  public tMachineBroker
2    {
3       public:
4          tMPIMachineBroker(int, char*);
5
6          virtual ~tMachineBroker();
7
8          virtual int GetNumberOfRemoteProcesses() const;
9          virtual int GetPID() const;
10         virtual tChannel* GetRemoteProcess();
11         virtual void FreeRemoteProcess(tChannel*);
12      private:
13         int Id;
14         tMPIChannel** Channels;
15         int NumberOfChannels;
16         int* FreeChannels;
17   }; // tMPIMachineBroker
```

Programa 5.19: Classe `tMPIMachineBroker`.

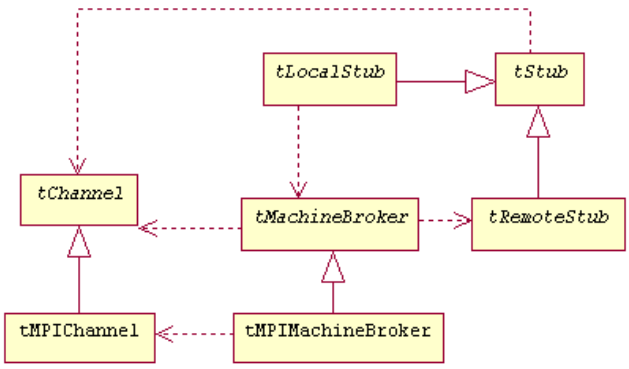


Figura 5.2: Classes de comunicação e serialização.

processamento da operação solicitada.

5.4 Execução

De forma similar à execução seqüencial, existe uma classe responsável pela execução da análise paralela utilizando a biblioteca MPI, denominada `tMPIFEMApp`. Essa classe deriva de `tFEMApp` que, conforme visto no Capítulo 3, é quem realiza essa atividade na análise seqüencial. A classe `tMPIFEMApp` define, de acordo com os parâmetros passados, o número de subdomínios que devem ser criados para a análise e orquestra a criação remota de tais subdomínios através da redefinição do método `CreateSubdomain()` da classe `tDomainPartitioner` (Figura 5.23) para retornar um ponteiro para um objeto `tLocalStubSubdomain` e não diretamente um `tSubdomain`. A função principal que inicia a execução seqüencial é apresentada no Programa 5.24.

```
1   class tLocalStubSubdomain:  public tLocalStubDomain,
2     virtual public ISubdomain

3   {
4     public:
5       tLocalStubSubdomain(tMeshPartition& mp, int noe, int non,
6         int noen, int nsp, int nlp):
7       {
8         *Channel << CONSTRUCTOR << mp << noe << non
9           << noen << nsp << nlp;
10      }
11
12      int AddElement(IElement*);
13      int AddNode(tNode*);
14      int AddExternalNode(tNode*);
15      int AddSPConstraint(tSPConstraint*);
16      int AddLoadPattern(tLoadPattern*);
17
18      bool AddNodalLoad(tNodalLoad*, int);
19      bool AddElementLoad(tElementLoad*, int);
20      bool ComputeNodalResponse();
21      bool FormTangent(tIntegrator*);
22      bool FormResidual(tIntegrator*);
23
24      tMatrix GetTangent();
25      tVector GetResidual();
26      tVector GetLastExternalResponse();
27      tVector GetLastResponse();
28
29      void SetAnalysis(tAnalysisDesc&);
30      void ClearAll();
31      void ApplyLoad(double);
32      void Update();
33      void SetLastResponse(const tVector&);
34      ...
35  }; // tLocalStubSubdomain
```

Programa 5.20: Classe `tLocalStubSubdomain`.

```
1    DEFINE_REMOTE_STUB(tSubdomain)
2       DEFINE_METHOD(tLocalStubSubdomain::CONSTRUCTOR)
3          int noe = Channel->ReadInt();
4          int non = Channel->ReadInt();
5          int nox = Channel->ReadInt();
6          int nsp = Channel->ReadInt();
7          int nlp = Channel->ReadInt();
8
9          Object = new tSubdomain(0, noe, non, nox, nsp, nlp);
10      END_METHOD
11      DEFINE_METHOD(tLocalStubSubdomain::SET_ANALYSIS)
12         tAnalysisDesc analysisDesc;
13         Object->SetAnalysis(analysisDesc);
14      END_METHOD
15      DEFINE_METHOD(tLocalStubSubdomain::ADD_ELEMENT)
16         IElement* element;
17         *Channel >> element;
18         *Channel << Object->AddElement(element);
19      END_METHOD
20      DEFINE_METHOD(tLocalStubSubdomain::ADD_NODE)
21         tNode* node;
22         *Channel >> node;
23         *Channel << Object->AddNode(node);
24      END_METHOD
25      DEFINE_METHOD(tLocalStubSubdomain::ADD_SP_CONSTRAINT)
26         tSPConstraint* sp;
27         *Channel >> sp;
28         *Channel << Object->AddSPConstraint(sp);
29      END_METHOD
30      DEFINE_METHOD(tLocalStubSubdomain::ADD_LOAD_PATTERN)
31         tLoadPattern* lp;
32         *Channel >> lp;
33         *Channel << Object->AddLoadPattern(lp);
34      END_METHOD
35      DEFINE_METHOD(tLocalStubSubdomain::ADD_EXTERNAL_NODE)
36         tNode* n;
37         *Channel >> node;
38         *Channel << Object->AddExternalNode(node);
39      END_METHOD
40      DEFINE_METHOD(tLocalStubSubdomain::ADD_NODAL_LOAD)
41         tNodalLoad* load;
42         int lpId;
43         *Channel >> load >> lpId;
44         *Channel << Object->AddNodalLoad(load, lpId);
45      DEFINE_METHOD(tLocalStubSubdomain::ADD_ELEMENT_LOAD)
46         tElementLoad* load;
47         int lpId;
48         *Channel >> load >> lpId;
```

Programa 5.21: Macro para definição de um *stub* remoto de subdomínio — Parte 1.

```
1         *Channel << Object->AddElementLoad(load, lpId);
2       END_METHOD
3       DEFINE_METHOD(tLocalStubSubdomain::CLEAR_ALL)
4         Object->ClearAll();
5       END_METHOD
6       DEFINE_METHOD(tLocalStubSubdomain:APPLY_LOAD)
7         double pseudoTime;
8         int lpId;
9         *Channel >> pseudoTime;
10        Object->ApplyLoad(pseudoTime);
11      END_METHOD
12      DEFINE_METHOD(tLocalStubSubdomain::UPDATE)
13        Object->Update();
14      END_METHOD
15      DEFINE_METHOD(tLocalStubSubdomain::FORM_TANGENT)
16        *Channel << Object->FormTangent(0);
17      END_METHOD
18      DEFINE_METHOD(tLocalStubSubdomain::FORM_RESIDUAL)
19        *Channel << Object->FormResidual(0);
20      END_METHOD
21      DEFINE_METHOD(tLocalStubSubdomain::GET_TANGENT)
22        *Channel << Object->GetTangent();
23      END_METHOD
24      DEFINE_METHOD(tLocalStubSubdomain::GET_RESIDUAL)
25        *Channel << Object->GetResidual();
26      END_METHOD
27      DEFINE_METHOD(tLocalStubSubdomain::COMPUTE_NODAL_RESPONSE)
28        *Channel << Object->ComputeNodalResponse();
29      END_METHOD
30      DEFINE_METHOD(tLocalStubSubdomain::GET_LAST_RESPONSE)
31        *Channel << Object->GetLastResponse();
32      END_METHOD
33      DEFINE_METHOD(tLocalStubSubdomain::SET_LAST_RESPONSE)
34        tVector X;
35        *Channel >> X;
36        Object->SetLastResponse(X);
37      END_METHOD
38      DEFINE_METHOD(tLocalStubSubdomain::SET_LAST_EXTERNAL_RESPONSE)
39        tVector X;
40        *Channel >> X;
41        Object->SetLastExternalResponse(X);
42      END_METHOD
43    END_REMOTE_STUB
```

Programa 5.22: Macro para definição de um *stub* remoto de subdomínio — Parte 2.

```
1    tSubdomain*
2    tMPIFEMApp::tPartitioner::CreateSubdomain(tMeshPartition* mp) const
3    {
4       int noe = mp->GetNumberOfCells();
5       return new tLocalStubSubdomain(mp, noe, 100, 20, 20);
6    }
```

Programa 5.23: Método `tMPIFEMApp::CreateSubdomain()`.

5.5 Considerações Finais

Com o surgimento de um crescente número de aplicações que demandam alta capacidade de processamento e necessitam de cada vez mais espaço de armazenamento, a utilização de técnicas de processamento paralelo e distribuído está se tornando uma alternativa bastante interessante.

Ao longo dos últimos anos, um grande número de trabalhos demonstraram a viabilidade da utilização de paralelismo no MEF, o que motivou o desenvolvimento deste trabalho. Uma das principais etapas de uma aplicação paralela consiste na divisão da entrada do problema entre os processadores envolvidos no processamento; no caso do MEF, o que precisa ser dividido é a malha de elementos finitos ou, ainda, o domínio do problema. Dentre as diversas técnicas pesquisadas, escolheu-se utilizar uma ferramenta pronta para particionamento de grafos.

Para realizar o envio de objetos de um processo para outro (geralmente localizados em processadores diferentes), é necessária a utilização de alguma forma de comunicação. Uma das alternativas para realizar essa comunicação é a utilização de bibliotecas de troca de mensagens como, por exemplo, o MPI, usado neste trabalho. Essas bibliotecas possuem a limitação de enviar apenas dados armazenados contiguamente na memória, o que implica na necessidade de desenvolvimento de alguma forma de serialização de objetos. Essa serialização é responsável por criar fluxos de bytes equivalentes a objetos, bem como por criar objetos a partir de um fluxo de bytes.

```
1   int main(int argc, char* argv[])
2   {
3      try
4      {
5         tMPIMachineBroker mb(argc, argv);
6
7         if (mb.GetPID() == 0)
8         {
9            tMPIFEMApp(argc, argv).Run();
10           return 0;
11        }
12
13        tMPIChannel parent(0);
14        char* serverType = parent.ReadString();
15        tRemoteStub* server = mb.BuildRemoteStub(serverType, parent);
16
17        delete serverType;
18        serverStub->Run();
19        delete server;
20        return 0;
21     }
22     catch (Exception e)
23     {
24        puts(e.GetMessage());
25     }
26     return -1;
27  }
```

Programa 5.24: Função principal da análise paralela.

CAPÍTULO 6

Exemplos

6.1 Considerações Iniciais

Visando testar as funcionalidades e comprovar a viabilidade de uso do sistema desenvolvido foi escolhida uma estrutura simples: uma viga retangular. Foram geradas malhas com diversos números de elementos, que permitiram a comparação entre os diferentes tipos de análise implementados: seqüenciais sem subestruturação, seqüenciais com subestruturação e paralelas com subestruturação. Nos dois últimos casos, foram considerados diferentes números de partições. Os tempos de execução — por fase e totais — obtidos com a análise seqüencial e paralela são comparados em todos os casos. A Seção 6.2 descreve a viga utilizada como exemplo e apresenta os resultados das execuções e uma análise crítica de tais resultados é o tema da Seção 6.3.

6.2 Viga Retangular

O exemplo executado consiste em uma viga retangular engastada em uma de suas extremidades e com um carregamento vertical aplicado na extremidade livre. A viga utilizada é definida por centro (0,0,0) e orientação (0,0,0). O módulo de elasticidade é de $210,000\ uf/d^2$ e coeficiente de Poisson igual a $0,3$. Foi aplicado um carregamento distribuído com vetor de carga $(0,\ -20,\ 0)$ na extremidade livre. Foram consideradas malhas com cinco diferentes discretizações:

1. (1/2, 1/3, 1/10);
2. (1/8, 1/12, 1/25);
3. (1/10, 1/15, 1/30);
4. (1/8, 1/27, 1/30);
5. (1/15, 1/18, 1/35).

Para gerar as cinco malhas das vigas, utilizou-se um gerador de malhas desenvolvido em um módulo separado do programa de análise (*Finite Element Generator*), cuja interface é mostrada na Figura 6.1. A Tabela 6.1 ilustra o número de nós e de elementos

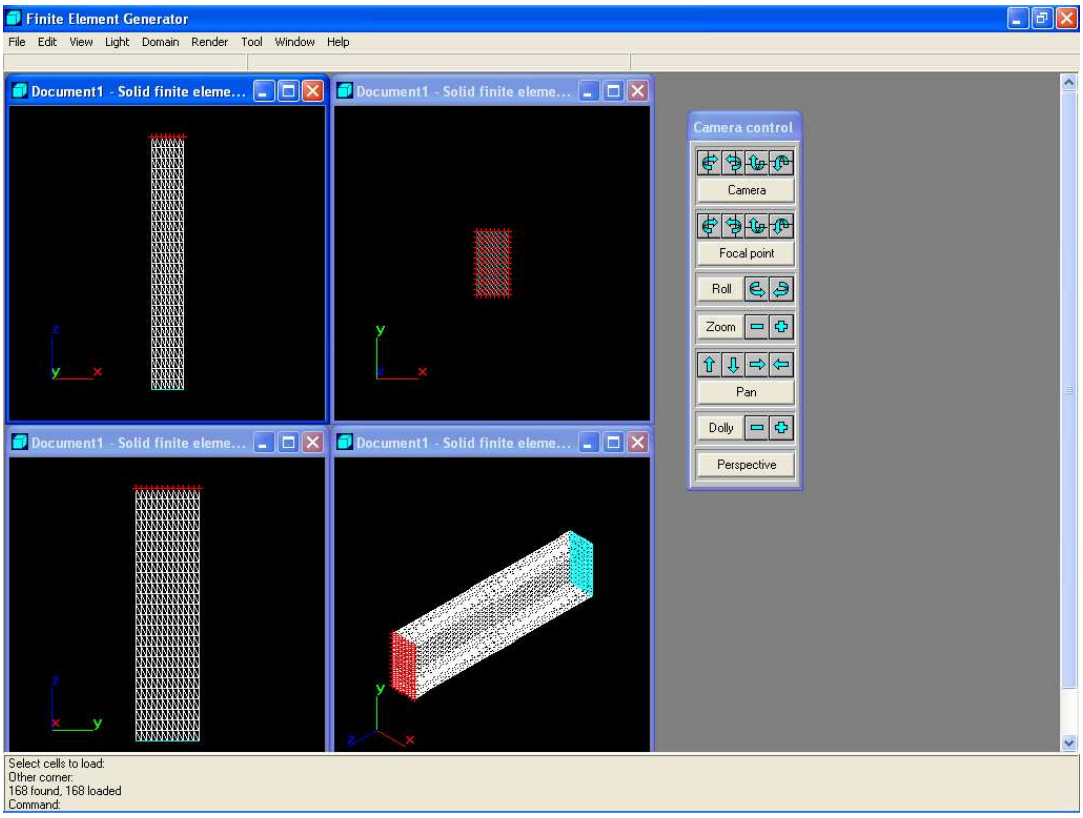


Figura 6.1: Interface do gerador de malhas.

gerados pelo FEG para as discretizações.

Discretização	Nós	Elementos
1	1638	1592
2	3042	2384
3	5456	3600
4	7812	5064
5	10944	5700

Tabela 6.1: Números de nós e elementos das discretizações.

Para cada uma das discretizações o programa foi executado seqüencialmente sem subestruturação ou com duas e quatro subestruturas e em paralelo com a utilização de três ou cinco máquinas. Todas as máquinas possuem processador Intel Celeron de 2.53GHz e 512 MB de memória RAM e 80 GB de disco rígido. Os testes executados visaram a obtenção dos tempos totais de processamento e dos tempos parciais de cada uma das cinco fases da execução:

1. Particionamento do domínio;

2. Inicialização da análise. Nesta fase são feitas a manipulação das restrições aplicadas ao domínio, a numeração dos DOFs, a definição do tamanho do sistema linear e a atualização do integrador do algoritmo;

3. Montagem do sistema de equações lineares;

4. Resolução do sistema de equações;

5. Finalização. Fase na qual são feitas atualizações no domínio, de acordo com o resultado da análise.

A Tabela 6.2 apresenta os tempos em segundos obtidos pela execução da análise seqüencial das diferentes discretizações da viga.

Discret.	Inic.	Mont. SE	Resol. SE	Finalização	Total
1	1,07	0,74	1,23	0,00	3,04
2	6,08	2,82	8,27	0,00	17,17
3	27,11	10,54	32,37	0,00	70,02
4	66,52	24,45	82,89	0,01	173,87
5	143,51	51,16	183,60	0,01	378,54

Tabela 6.2: Viga - execução seqüencial sem subestruturação.

Os tempos obtidos em segundos na execução seqüencial com a utilização de duas e quatro subestruturas são apresentados, respectivamente, nas Tabelas 6.3 e 6.4.

Discret.	Partic.	Inic.	Mont. SE	Resol. SE	Finalização	Total
1	0,39	0,00	8,14	0,02	0,03	8,19
2	0,92	0,00	46,84	0,13	0,90	47,87
3	8,48	0,00	185,63	0,36	0,24	186,23
4	16,49	0,02	495,83	1,37	0,53	497,75
5	6,03	0,01	1112,97	1,80	1,14	1115,93

Tabela 6.3: Viga - execução seqüencial com duas subestruturas.

Discret.	Partic.	Inic.	Mont. SE	Resol. SE	Finalização	Total
1	0,56	0,00	9,45	0,36	0,03	9,84
2	1,53	0,20	49,52	2,69	0,12	52,35
3	8,96	0,04	197,56	9,66	0,34	207,60
4	20,27	0,07	404,31	19,47	0,66	424,51
5	23,73	0,14	1256,22	47,87	2,15	1306,38

Tabela 6.4: Viga - execução seqüencial com quatro subestruturas.

A execução paralela foi realizada com máquinas com a mesma configuração da execução seqüencial, interconectadas por uma rede de 10Mbps. Os resultados obtidos em segundos com as execuções em três e cinco máquinas são apresentados nas Tabelas 6.5 e 6.6.

Discret.	Partic.	Inic.	Mont. SE	Resol. SE	Finalização	Total
1	4,12	0,00	4,20	0,02	0,02	4,24
2	8,19	0,00	24,43	0,12	0,05	24,60
3	23,54	0,00	99,0300	0,36	0,13	99,52
4	38,28	0,00	281,18	, 1,38	0,29	282,85
5	40,87	0,02	598,86	1,80	0,45	601,13

Tabela 6.5: Viga - execução paralela com duas máquinas.

Discret.	Partic.	Inic.	Mont. SE	Resol. SE	Finalização	Total
1	4,39	0,0	2,57	0,36	0,01	2,94
2	8,95	0,01	13,42	2,69	0,05	16,17
3	23,65	0,05	54,75	9,65	0,11	64,56
4	42,09	0,08	101,04	19,50	0,17	120,79
5	56,34	0,14	356,58	47,97	0,42	405,04

Tabela 6.6: Viga - execução paralela com quatro máquinas.

6.3 Análise dos Resultados

Com base nos resultados obtidos, algumas considerações podem ser feitas. Quando são considerados os tempos totais obtidos na execução seqüencial com e sem subestruturação, excetuando-se o particionamento do domínio, há uma relativa redução no desempenho da análise. Esse resultado pode ser explicado pelo processo executado pela subestruturação pois, apesar de o sistema global ter sido reduzido, as contribuições de cada subestrutura ainda precisam ser calculadas e, após o cálculo do sistema global, as subestruturas precisam ser relaxadas, ou seja, receber as contribuições reais dos nós de interface. Dessa forma, com a utilização de apenas um processador para a subestruturação aumenta o processamento que deve ser realizado por um único processador. A influência exercida pela subestruturação na análise seqüencial depende do tamanho da malha do domínio global de acordo com os testes executados. Pode-se notar que, para as malhas menores (discretizações 1, 2, 3 e 4), o aumento do número de partições — de 2 para 4 —implica no aumento do tempo de execução, enquanto que, para as demais malhas, com quatro subestruturas foi obtido um tempo menor de execução com relação ao uso de duas subestruturas.

O processamento paralelo mostrou-se significativamente mais eficiente do que a execução seqüencial em quase todos os exemplos executados, excetuando-se a discretização 5. Esse resultado era esperado visto que com o uso de mais processadores, algumas tarefas podem ser realizadas em paralelo. A abordagem paralela implementada neste trabalho utiliza o paradigma mestre/escravo de programação, no qual um processador solicita que operações sejam realizadas por outros processadores; mais especificamente, o processador 0 é o mestre, sendo responsável por resolver o sistema dos nós de interface mas, para tal, precisa pedir as colaborações de cada subdomínio e fica esperando por isso. Essas colaborações, que eram calculadas em seqüência com um único processador, podem ser feitas em paralelo nos processadores diferentes de 0. Entretanto, com o aumento do número de nós dos subdomínios, aumenta também a necessidade de comunicação, pois o número de dados a serem trocados entre diferentes processadores e comunicação torna-se o gargalo da aplicação. Além disso, o

algoritmo utilizado para condensação em cada subestrutura foi implementado baseado nas fórmulas da subestruturação, apresentada no Capítulo 4, sem nenhuma otimização, o que resultou em um algoritmo mais ineficiente do que o algoritmo utilizado na análise seqüencial. Dessa forma, a resolução em paralelo equivale a resolver vários sistemas menores ao mesmo tempo, mas utilizando um algoritmo ineficiente; enquanto que a resolução seqüencial equivale a resolver um sistema maior com um algoritmo ótimo. Conclui-se, assim, que, para que a paralelização seja uma alternativa eficiente para malhas grandes, é necessário implementar melhorias na condensação da matriz de rigidez e no vetor de esforços de cada subdomínio.

Outro ponto que merece atenção é que não existe uma quantidade ideal de processadores para realizar a análise no menor tempo, pois isso depende de vários fatores como, por exemplo, a proporção entre o número total de nós e o número de nós que pertencem à interface, a velocidade da rede de interconexão e capacidade de processamento das máquinas envolvidas.

6.4 Considerações Finais

Para testar o sistema desenvolvido considerou-se uma viga retangular engastada em uma extremidade e com um carregamento distribuído na extremidade oposta. Foram realizados testes — seqüenciais com e sem subestruturação e paralelos — considerando cinco diferentes discretizações do domínio e os resultados foram comparados para verificar a eficiência de cada uma das quatro fases principais da análise: inicialização, montagem do sistema de equações lineares, resolução do sistema e finalização.

Os resultados obtidos nos testes mostraram que a utilização da subestruturação seqüencialmente é mais ineficiente devido à fase de obtenção das contribuições de cada subdomínio. A subestruturação paralela, por sua vez, conseguiu tempos de execução menores em quatro dos cinco testes realizados, entretanto, na maior das malhas o tempo de execução da simulação, com dois ou com quatro processadores, foi maior. Esses resultados justificaram-se, principalmente, pelo aumento das fronteiras entre os subdomínios, o que ocasiona um aumento na comunicação necessária entre os processadores, limitando a velocidade de processamento.

CAPÍTULO 7

Conclusão

7.1 Resultados Obtidos

O objetivo geral deste trabalho foi o *desenvolvimento de um framework em C++ para análise seqüencial e em paralelo de sólidos elásticos pelo método dos elementos finitos* (MEF), a partir de um modelo orientado a objetos flexível e extensível.

No Capítulo 1 foram apresentados os tempos obtidos com a execução do método dos elementos de contorno (MEC) de uma esfera oca, cujos pólos estavam engastados, submetida a uma pressão interna e constatou-se que, mesmo nesse exemplo simplificado, a demanda por capacidade computacional foi elevada. Com base nessa constatação, justificou-se a necessidade de buscar alternativas para realizar a análise de forma mais eficiente, dentre elas o paralelismo. O MEF foi escolhido após a comparação de sua formulação com a do MEC, na qual verificou-se que este último, apesar de resultar em sistemas de equações com um número menor de incógnitas, depende de informações de todo o contorno do sólido sendo analisado, o que dificulta a divisão do domínio do problema e, conseqüentemente, a paralelização.

Os objetivos específicos foram:

- *Estudo das técnicas de paralelização do MEF com decomposição de domínio encontradas na literatura.* A revisão bibliográfica mostrou que existem diferentes abordagens para paralelizar o MEF e que uma característica é comum a todas: a necessidade de dividir o domínio do problema entre os processadores. Dentre as alternativas pesquisadas, destacou-se a técnica da decomposição de domínio, na qual a malha de elementos finitos é particionada em submalhas de tamanhos aproximadamente iguais e cada uma dessas submalhas pode ser analisada separadamente. Dado que essas submalhas não são de fato estruturas independentes, é necessária a utilização de alguma técnica que possibilite a interação entre elas; nesse contexto, a subestruturação mostrou-se adequada. Com a subestruturação, cada uma das submalhas — chamadas de subestruturas — pode ser tratada como uma estrutura isolada, em paralelo. Após a obtenção dos resultados em cada subestrutura, as suas contribuições para os nós de interface — pertencentes a mais de uma subestrutura — são combinadas e a solução real para os nós de interface

é obtida e retornada às subestruturas que, paralelamente, obtêm a solução real no restante do domínio. Resumo dos estudos efetuados sobre o método dos elementos finitos, decomposição de domínio baseada em particionamento de grafos e paralelização do MEF são apresentados nos Capítulos 3, 4 e 5.

- *Revisão de OSW e elaboração de um modelo orientado a objetos a partir do qual foram implementados os componentes do framework envolvidos na análise seqüencial e em paralelo do MEF.* Com a orientação a objetos, tornou-se mais clara a divisão de funcionalidades entre componentes do sistema, através da implementação de diversos conjuntos de classes responsáveis pelas diversas fases da análise. As classes foram modeladas de forma a garantir a máxima flexibilidade do sistema; para alterar uma característica da análise como, por exemplo, o algoritmo de solução dos sistemas de equações ou o tipo de análise (estática ou dinâmica), é suficiente a alteração de apenas algumas linhas do programa principal. Isso garante que o sistema seja extensível e estimula o reaproveitamento de código. Diagramas de classes UML com partes do modelo são mostrados nos Capítulos 2, 3, 4, e 5.

- *Implementação e descrição das classes do modelo proposto.* Foram implementadas a análise seqüencial, com e sem subestruturação, e paralela com subestruturação. A implementação paralela utilizou as diretivas de comunicação da biblioteca MPI encapsulada em classes para escrita e leitura de fluxos de bytes. Para criação desses fluxos, foi criado um sistema de serialização que possibilitou que qualquer objeto que precisasse ser enviado para um processador remoto pudesse ser transformado em um fluxo de bytes, bem como o processo contrário, de obtenção do objeto original no processador destino da comunicação. A descrição das principais funcionalidades das classes mais importantes do framework é apresentada nos Capítulos 2, 3, 4 e 5.

- *Exemplificação e obtenção de medidas de desempenho de análises de sólidos elásticos realizadas com o framework.* Exemplos da execução das aplicações de análise desenvolvidas com o framework são apresentados no Capítulo 6. Os resultados obtidos com as diferentes abordagens de análise demonstraram que a fase de resolução do sistema de equações lineares, conforme esperado, é mais eficiente com a utilização da subestruturação, entretanto, esse desempenho superior é degradado pela estratégia adotada para a fase de montagem do sistema, embora, um pouco menor na abordagem paralela. Esse baixo desempenho também já era esperado pois a subestruturação seqüencial precisa analisar cada subestrutura seqüencialmente, não aproveitando a característica de independência entre elas. Na subestruturação paralela, diferentemente, percebe-se que a sobrecarga acarretada pela comunicação também é significativa, mesmo não sendo tão prejudicial quanto a seqüencialização das subestruturas.

Do que posto acima pode-se concluir, portanto, que todos os objetivos do trabalho, geral e específicos, foram alcançados.

7.2 Propostas para Trabalhos Futuros

O trabalho desenvolvido foi um primeiro passo no sentido de utilizar os conceitos de paralelismo na simulação estrutural dada pelo Grupo de Computação Gráfica e Mecânica Computacional em conjunto com Grupo de Algoritmos Paralelos e Distribuídos do DCT-UFMS. Os resultados obtidos indicam a necessidade do estudo de alternativas para realizar essa tarefa visando um melhor desempenho como, por exemplo, diferentes estratégias para a fase de montagem do sistema de equações, a fase mais custosa do processo.

Além das melhorias, pode-se elencar uma série de características que podem ser adicionadas ao sistema. Uma funcionalidade de particular interesse seria a incorporação de classes que realizem a análise dinâmica de estruturas, ou seja, que permitam acompanhar o comportamento de um sólido ao longo de um período de tempo. Essa adição pode ser feita de maneira a não precisar alterar as classes já existentes, pois tal hierarquia foi proposta considerando a análise composta por diversos passos — que, no caso da análise seqüencial, é único —, permitindo a integração da análise dinâmica de maneira imediata. A adição de novos métodos de resolução de sistemas de equações, incluindo paralelos, é outra alternativa que deve ser considerada para tentar diminuir ainda mais o tempo gasto por essa tarefa.

Pela análise dos resultados dos testes, percebe-se que os esforços devem se concentrar em implementar uma nova estratégia para realizar a montagem do sistema de equações, concentrando esforços na diminuição da comunicação, que se mostrou o gargalo no desempenho. Além disso, seria interessante a integração do sistema de análise desenvolvido neste trabalho com módulos que realizem as fases de pré-processamento — geração da malha de elementos finitos e aplicação gráfica que permita sua visualização e aplicação de cargas e restrições — e de pós-processamento — visualização dos resultados obtidos —, que neste trabalho foram feitas por duas aplicações diferentes, FEG e FEV.

Referências Bibliográficas

[AGH85] AGHA, G.A. *"Actors: A Model of Concurrent Computation in Distributed Systems"*, Technical Report 844 (N00014-80-C-0505), MIT Artificial Intelligence Laboratory, Cambridge, Massachussets, 1985.

[ALM99] ALMEIDA, V.S. *Uma Adaptação do MEF para Análise em Multicomputadores: Aplicações em alguns Modelos Estruturais*. São Carlos, 1999. Dissertação (Mestrado) – Escola de Engenharia de São Carlos – Universidade de São Paulo.

[ASS03] ASSAN, A.E. *Método dos Elementos Finitos: Primeiros Passos*. 2 ed. Campinas, SP. Editora da Unicamp, 2003.

[BAR95a] BARNARD, S.T. PMRSB: Parallel Multilevel Recursive Spectral Bisection. *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing* . p.1-23, 1995.

[BAR95b] BARNARD, S.T.; SIMON, H. A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes. *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*. p.627-632, 1995.

[BIS97] BISWAS, R.; OLIKER, L. Load Balancing Unstructured Adaptive Grids for CFD. *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. 1997.

[BRE78] BREBBIA, C.A. *The Boundary Element Method for Engineers*. London, Pentech Press, 1978.

[BRE84] BREBBIA, C.A.; TELLES, J.C.F.; WROBEL, L.C. *Boundary Element Techniques: Theory and Applications in Engineering*. Springer-Verlag, 1984.

[BUI93] BUI, T.; JONES, C. T. A Heuristic for Reducing Fill in Sparse Matrix Factorization. *6th SIAM Conference on Parallel Processing for Scientific Computing*. p.445-452, 1993.

[CAS00] CASTAÑOS, J.G.; SAVAGE, J.E Repartitioning Unstructured Adaptive Meshes *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*. p.823-832, 2000.

[CHE02] CHEN, J.; TAYLOR, V.E Mesh Partitionig for Efficient Use of Distributed Systems *IEEE Transactions on Parallel and Distributed Systems*. v.13, n.1, p.67-79, 2002.

[CLA79] CLARKE, A.B.; DISNEY, R.L. *Probabilidade e Processos Estocásticos*. 3a ed, Livros Técnicos e Científicos, 1979.

[COU01] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems - Concepts and Design*. 3a ed, Addison-Wesley, 2001.

[DIN96] DING, H.Q.; FERRARO, R.D. An Element-Based Concurrent Partitioner for Unstructured Finite Element Meshes. *10th International Parallel Processing Symposium (IPPS '96)*. p.601-605, 1996.

[DOR95] DORMANNS, M.; HEISS, H.U. Partitioning and Mapping of Large FEM-Graphs by Self-Organization. *3rd Euromicro Workshop on Parallel and Distributed Processing*. p.227-235, 1995.

[FID82] FIDUCCIA, C.M.; MATTHEYSES, R.M A Linear Time Heuristic for Improving Network Partitions. *Proceedings of the 19th IEEE Design Automation Conference*. p.175-181, 1982.

[FIS96] FISHMAN, G.S. *Monte Carlo: Concepts, Algorithms, and Applications*. New York, Springer-Verlag, 1996.

[FJA98] FJÄLSTRÖM, P.O. Algorithms for Graph Partitionig: A Survey. *Linköping Electronic Articles in Computer and Information Science*. v.3, n.10, 1998.

[FLE95] FLEISCHER, M. Simulated Annealing: Past, Present, and Future. *Proceedings of the 27th Conference on Winter Simulation*. ACM Press, p.155-161, 1995.

[FRA03] FRAGAKIS, Y.; PAPADRAKAKIS, M. The Mosaic of High Performance Domain Decomposition Methods for Structural Mechanics: Formulation, Interrelation and Numerical Efficiency of Primal and Dual Methods. *Computer Methods in Applied Mechanics and Engineering*. p.3799-3830, 2003.

[GOP96] GOPPOLD, M.; HAASE, G., HEISE, B., KUHN, M. *Preprocessing in BE/FE Domain Decomposition Methods*. Technical Report 96-2, Institute of Mathematics – University Linz, 1996.

[GRA97] GRABOWSKY, L.; REHM, W. Effiency Issues of a Parallel FEM Implementation on Shared Memory Computers. *Proceedings of 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*. p.156-161, 1997.

[HEN92] HENDRICKSON, B.; LELAND, R. *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

[HEN93a] HENDRICKSON, B.; LELAND, R. *A Multilevel Algorithm for Partitioning Graphs*. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[HEN93b] HENDRICKSON, B.; LELAND, R. *The Chaco User´s Guide, version 1.0*. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[HSI94] HSIEH, S.H. A Mesh Partitioning Tool and its Applications to Parallel Processing. *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*. p.168-173, 1994.

[HSI95] HSIEH, S.H.; PAULINO, G.H.; ABEL, J.F. Recursive Spectral Algorithms for Automatic Domain Partitioning in Parallel Finite Element Analysis. *Computer Methods in Applied Mechanics and Engineering*. v.121, n.1-4, p.137-162, 1995.

[ING93] INGBER, L. Simulated Annealing: Practice versus Theory. *Mathematical Computer Modelling*. v.18, n.11, p.29-57, 1993.

[JER93] JERRUM, M.; SORKIN, G.B. Simulated Annealing for Graph Bisection. *IEEE Symposium on Foundations of Computer Science*. p.94-103, 1993.

[JOH89] JOHNSON, D.S.; ARAGON, C.R.; MCGEOCH, L.A.; SCHEVON, C. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitionig. *Operations Research*. Operations Research Society of America, v.37, n.6, p.865-892, 1989.

[JOH91] JOHNSON, D.S.; ARAGON, C.R.; MCGEOCH, L.A.; SCHEVON, C. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitionig. *Operations Research*. Operations Research Society of America, v.39, n.3, p.378-406, 1991.

[JON94] JONES, M.; PLASSMANN, P. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. *Proceedings of 1994 Scalable High-Performance Computing Conference*. IEEE Computer Society, p.478-485, 1994.

[KAN94] KANE, J.H. *Boundary Element Analysis in Engineering Continuum Mechanics*. Prentice-Hall International, 1994.

[KAR95] KARYPIS, G.; KUMAR, V. *Analysis of Multilevel Graph Partitioning*. Technical Report TR 95-037, Department of Computer Science – University of Minnesota, 1995.

[KAR96a] KARYPIS, G.; KUMAR, V. Parallel Multilevel Graph Partitioning. *10th International Parallel Processing Symposium (IPPS '96)*. p.314-319, 1996.

[KAR96b] KARYPIS, G.; KUMAR, V. *Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs*. Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996.

[KAR98a] KARYPIS, G.; KUMAR, V. *METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. Department of Computer Science – University of Minnesota / Army HPC Research Center, Minneapolis, MN 55455, 1998.

[KAR98b] KARYPIS, G.; KUMAR, V. *Multilevel Algorithms for Multi-constraint Graph Partitioning*. Technical Report TR 98-019, Department of Computer Science – University of Minnesota, 1998.

[KAR98c] KARYPIS, G.; KUMAR, V. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*. v.48, n.1, p.96-129, 1998.

[KAR98d] KARYPIS, G.; KUMAR, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*. v.20, n.1, p.359-392, 1998.

[KER70] KERNIGHAN, B.W.; LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*. v.49, n.2, p.291-307, 1970.

[KIR83] KIRKPATRICK, S.; GELLAT, C.D.; VECCHI, M.P. Optimization by Simulated Annealing. *Science*. v.220, n.4598, p.671-680, 1983.

Engineering Analysis with Boundary Elements. v.19, n.1, p.33-39, 1997.

[KUC05] KÜÇÜKPETEK, S.; POLAT, F.; OGUZTÜZÜN, H. Multilevel Graph Partitioning: An Evolutionary Approach. *Journal of the Operational Research Society*. v.56, n.5, p.549-562, 2005.

[MCK97] MCKENNA, F.T. *Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing*. Berkeley, 1997. Tese (Doutorado) – University of California, Berkeley.

International Journal of Supercomputing. 2000.

[MAL69] MALVERN, L.E. *Introduction to the Mechanics of a Continuous Medium*. New Jersey, Prentice-Hall International, 1969.

[MET53] METROPOLIS, N.; ROSENBLUTH, A.W.; ROSENBLUTH, M.N.; TELLER, A.H.; TELLER,E. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*. v.21, n.6, p.1087-1092, 1953.

[MOR00] MORETTI, C.O.; CAVALCANTE NETO, J.B; BITTENCOURT, T.N.; MARTHA, L.F. A Parallel Environment for Three-Dimensional Finite Element Analysis. *Developments in Engineering Computational Technology*, Topping, B.H.V., ed, Civil-Comp Press, Edinburgh, Scotland, p.283-287, 2000.

[MÜL02] MÜLLER, R.M. *Visualização de Modelos de Sólidos Analisados pelo Método dos Elementos de Contorno*. Dissertação (Mestrado) – Universidade Federal de Mato Grosso do Sul. Campo Grande, 2002.

[MUR95] MURTHY, K.N.B.; MURTHY, C.S.R. A New Parallel Algorithm for Solving Sparse Linear Systems. *1995 IEEE International Symposium on Circuits and Systems (ISCAS '95)*. v.2, p.1416-1419, 1995.

[MUR96] MURTHY, K.N.B.; MURTHY, C.S.R. Parallel Algorithm for Solving Sparse Linear Systems. *Electronics Letters*. v.32, n.19, p.1766-1768, 1996.

[NAH86] NAHAR, S.; SAHNI, S.; SHRAGOWITZ, E. Simulated Annealing and Combinatorial Optimization. *Proceedings of the 23rd ACM/IEEE conference on Design automation*. ACM Press, p.293-299, 1986.

[NIK96] NIKISHKOV, G.P.; MAKINOUCHI, A. Efficiency of the Domain Decomposition Method for the Parallelization of Implicit Finite Element Code. *1996 International Conference on Parallel and Distributed Systems (ICPADS '96)* . p.49-56, 1996.

[ODE87] ODEN, J.T. Some Historic Comments on Finite Elements. *Proceedings of the ACM Conference on History of Scientific and Numeric Vomputation*. p.125-130, 1987.

[PAG98] PAGLIOSA, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos*. Tese (Doutorado) – Escola de Engenharia de São Carlos – Universidade de São Paulo. São Carlos, 1998.

[PIO83] PION, G.; BECKER, M.; CALLEGHER, E.; FOURNIER, R. Is Parallelism Useful in Order to Improve Response Time of a Finite Element Method? *IEEE Transactions on Magnetics*. v.19, n.4, p.2520-2522, 1983.

[POT95] POTHEN, A. Graph Partitioning Algorithms with Applications to Scientific Computing. *Parallel Numerical Algorithms*, Keyes, D.E., Sameh, A.H., Venkatakrishnan, V., eds, Kluwer Academic Press, 1995.

[REZ95] REZENDE, M.N. *Processamento Paralelo em Análise Estrutural*. Tese (Doutorado) – Escola de Engenharia de São Carlos – Universidade de São Paulo. São Carlos, 1995.

[SAN97] SANDIA NATIONAL LABORATORIES. Global Optimization Survey - Main Page. *http://www.cs.sandia.gov/opt/survey/sa.html*. 1997. Data de Acesso: 20 de janeiro de 2004.

[SAV91] SAVAGE, J.E.; WLOKA, M. Parallelism in Graph Partitioning. *Journal of Parallel and Distributed Computing*. v.13, p.257-272, 1991.

[SCH97] SCHLOEGEL, K.; KARYPIS, G.; KUMAR, V. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *Journal of Parallel and Distributed Computing* . v.47, n.2, p.109-124, 1997.

[SCH98] SCHLOEGEL, K.; KARYPIS, G.; KUMAR, V. Dynamic Repartitioning of Adaptively Refined Meshes. *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. p.1-8, 1998.

[SCH00] SCHIAVONE, G.A.; TRACY, J.; PALANIAPPAN, R. Preliminary Investigations into Distributed Computing Applications on a Beowulf Cluster. *Proceedings of the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications*. p.13-17, 2000.

[SCH96] SCHROEDER, W.J; MARTIN, K.; LORENSEN, B. *The Visualization Tollkit - An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1996.

[SCO03] SCOTT, J.A. Parallel Frontal Solvers for Large Sparse Linear Systems. *ACM Trans. Math. Softw.*. v.29, n.4, p.395-417, 2003.

[SHE94] SHEWCHUK, J.R. *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*. Technical Report – School of Computer Science – Carnegie Mellon University, 1994.

[SIM93] SIMON, H.D.; TENG, S. *How Good Is Recursive Bisection*. Technical Report RNR-93-012, NAS Systems Division, NASA, Moffet Fiels, CA, 1993.

[SMI90] SMITH, S.H.,JR.; KRAD, H. A Parallel, Iterative Method for Solving Large Linear Systems. *Proceedings of Southeastcon '90*. IEEE. v.2, p.533-537, 1990.

[SOT02] SOTELINO, E.D.; DERE, Y. Parallel and Distributed Finite Element Analysis of Structures. *Engineering Computational Technology*, Topping, B.H.V., Bittnar, Z., eds, Saxe Coburg Publications, Stirling, Scotland, p.221-250, 2002.

[SWI00] SWISHER, J.R.; HYDEN, P.D.; JACOBSON, S.H.; SCHRUBEN, L.W. Simulation Optimization: A Survey of Simulation Optimization Techniques and Procedures. *Proceedings of the 32nd Conference on Winter Simulation*. Society for Computer Simulation International, p.119-128, 2000.

[TSE86] TSENG, P.S.; HWANG, K. Parallel Preprocessing and Postprocessing in Finite-element Analysis on a Multiprocessor Computer. *Proceedings of 1986 Fall Joint Computer Conference on Fall Joint Computer Conference*. IEEE Computer Society Press. p.307-314, 1986.

[WAL97] WALSHAW, C.; EVERETT, M.G.; CROSS, M. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*. v.47, n.2, p.102-108, 1997.

[WIL96] WILBURN, V.C.; HAK-LIM KO; ALEXANDER, W.E. An Algorithm and Architecture for the Parallel Solution of Systems of Linear Equations . *Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications*. p.392-398, 1996.

[ZIE94a] ZIENKIEWICZ, O.C.; TAYLOR, R. *The Finite Element Method*. v.1, McGraw-Hill, 1994.

[ZIE94b] ZIENKIEWICZ, O.C.; TAYLOR, R. *The Finite Element Method*. v.2, McGraw-Hill, 1994.