UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL, ARQUITETURA E URBANISMO

DEPARTAMENTO DE ESTRUTURAS

Adaptatividade *hp* em paralelo

Autor: **Eng. Edimar Cesar Rylo**

Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL, ARQUITETURA E URBANISMO

DEPARTAMENTO DE ESTRUTURAS

Adaptatividade *hp* em paralelo

Autor: **Eng. Edimar Cesar Rylo**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

Curso: **Engenharia Civil**
Área de concentração: **Estruturas**

Tese de doutorado apresentada à comissão de Pós Graduação da Faculdade de Engenharia Civil da UNICAMP, como requisito para a obtenção do título de Doutor em Engenharia Civil.

Campinas, Agosto de 2007
SP - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

R985a

Rylo, Edimar Cesar
Adaptatividade hp em paralelo / Edimar Cesar Rylo. --Campinas, SP: [s.n.], 2007.

Orientador: Philippe Remy Bernard Devloo.
Tese de Doutorado - Universidade Estadual de Campinas, Faculdade de Engenharia Civil, Arquitetura e Urbanismo.

1. Programação paralela (Computação). 2. Método dos elementos finitos. 3. Programação orientada a objetos (Computação). 5. Parallel programming (Computer science). I. Devloo, Philippe Remy Bernard. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil, Arquitetura e Urbanismo. III. Título.

Título em Inglês: hp adaptive technique in parallel
Palavras-chave em Inglês: Finite element method, Object oriented programming (Computer science)
Área de concentração: Estruturas
Titulação: Doutor em Engenharia Civil
Banca examinadora: José Luiz Antunes de Oliveira e Sousa, Vinícius Fernando Arcaro, Marco Lúcio Bittencourt, Fernando Luiz Bastos Ribeiro
Data da defesa: 31/07/2007
Programa de Pós Graduação: Engenharia Civil

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL, ARQUITETURA E URBANISMO

Adaptatividade *hp* em Paralelo

Edimar Cesar Rylo

Tese de Doutorado aprovada pela Banca Examinadora, constituída por:

Prof. Dr. Philippe Remy Bernard Devloo
Presidente e Orientador / FEC-UNICAMP

Prof. Dr. José Luiz Antunes de Oliveira e Sousa
FEC-UNICAMP

Prof. Dr. Vinícius Fernando Arcaro
FEC-UNICAMP

Prof. Dr. Marco Lúcio Bittencourt
FEM - UNICAMP

Prof. Dr. Fernando Luiz Bastos Ribeiro
COPPE - UFRJ

Campinas, 31 de julho de 2007

Aos meus pais Alceu e Maria Elizia.

Agradecimentos

Ao Laboratório de Mecânica Computacional do Departamento de Estruturas da Faculdade de Engenharia Civil - UNICAMP pela infraestrutura.

À PETROBRAS e à FAPESP-Embraer pelo financiamento da infraestrura computacional e pela bolsa de estudos.

Ao Cenapad - SP por disponbilizar o ambiente de processamento paralelo.

Ao amigo e orientador Philippe pela amizade, orientação e compartilhamento de suas idéias.

Aos professores do DES pelo conhecimento transmitido.

Aos colegas do LabMeC pelo ambiente descontraído e estimulante.

À minha família em Campinas José Mauro, Sônia, Jonathan e Laís que me abrigaram em minha volta à Universidade.

À minha esposa Ane por toda sua paciência, compreensão e suporte durante essa jornada.

Resumo

Esse trabalho apresenta uma abordagem para a implementação de métodos auto-adaptativos *hp* em malhas de elementos finitos utilizando processamento paralelo para a seleção do padrão *hp* a ser utilizado em cada elemento da malha. Dois tópicos são destacados: análise da qualidade da aproximação e modo de melhoria do espaço de aproximação.

O trabalho apresentado propõe uma estrutura para a implementação de métodos *hp* auto-adaptativos no ambiente PZ. Essa estrutura é genérica e pode ser utilizada independentemente de: formulação fraca, tipo de elemento utilizado, método de resolução etc. A estrutura proposta define a interface requerida de um estimador de erros, bem como a interface para a seleção do padrão de refinamento. Tal interface contempla a possibilidade de análise de malhas com elementos contínuos ou descontínuos.

A implementação apresentada contempla o processamento em máquinas paralelas, de modo que o tempo de obtenção de uma malha adaptada seja aceitável em aplicações práticas. O cálculo do erro bem como a definição dos padrões de refinamento pode ser feito utilizando processamento paralelo, em ambientes com memória compartilhada ou distribuída.

Uma metodologia de refinamento *h* baseado em padrões de refinamento foi desenvolvida, implementada e validada. Essa metodologia facilita a implementação de padrões de refinamento. Em contrapartida, a geração de malhas com espaços de aproximação contínuos impõe restrições para a seleção do padrão de refinamento de um elemento. Assim, para a seleção de um padrão de refinamento de um elemento foi desenvolvida uma metodologia de análise de padrões adimissíveis. A seleção do padrão de refinamento tendo por base uma análise de padrões admissíveis é um ponto que requer novas pesquisas, sendo considerado um dos desafios da auto-adaptatividade (ver Zienkiewicz [55]).

Abstract

This work presents a study of *hp* adaptive methods applied to finite element approximations. Two topics are emphasized: analysis of the quality of the approximation and methodology of refinement of the approximation space.

The main objective of the work is to conceive a framework for developing *hp*-adaptive algorithms within the PZ environment. The framework is independent of the weak statement, type of element or resolution method. The framework uses separate interfaces to define the error estimation method and selection of refinement pattern.

Secondly, the framework was ported to parallel processing using the object oriented framework OOPar. The intent of parallelizing the adaptive process is to reduce the time spent in error estimation and choice of the optimal refinement pattern and thus bring adaptivity to a level where it can be used as a routine analysis method. Both error estimation and choice of refinement pattern are implemented on a shared and/or distributed machine.

Finally, a methodology was developed to extend the h-adaptive refinement process based on refinement patterns. Together with the implementation of refinement patterns, a procedure was developed to check on the compatibility of refinement patterns of two neighboring elements. The choice of the "best" refinement patterns is one of the main challenges of adaptive methods (Zienkiewicz [55]). The availability of different ways of refining elements increases the flexibility of the code, but also introduces the challenge of deciding which pattern is the "best" pattern. It is possible that the combination of optimized h-refinement together with choice of h and/or p refinement may lead to very efficient approximation spaces for a given problem.

Sumário

Lista de Figuras

Lista de Algoritmos

Nomenclatura

Siglas

CDC: condição de contorno

C++: linguagem de programação

CPU: unidade de processamento central

CVS: sistema de versões concorrentes (concurrent version system)

ddd: interface gráfica para o *gdb*

dof: graus de liberdade (degrees of freedom)

gdb: depurador GNU (debugger)

I/O: entrada – saída (input – output)

Linux: sistema operacional baseado em software livre

MEF: Método dos elementos finitos

MPI: padrão de comunicação de dados entre computadores utilizado em computação científica

OO: orientação a objetos

PZ: Ambiente de programação de elementos finitos orientado a objetos utilizado para a implementação computacional desse trabalho

UML: linguagem unificada de modelagem (*unified modelling language*)

UNIX: sistema operacional multitarefa desenvolvido inicialmente pela Bell Labs

Operadores

$Dim(.)$: dimensão do espaço de funções

$\triangle$:Operador laplaciano

∇: Operador gradiente

$||.||_{\infty}$: Norma infinita

$||.||_{E}$: Norma energia

Letras latinas Minúsculas

u: função correspondente a solução da equação diferencial de interesse

u^D: valor da fução u no contorno submetido a condição de contorno Dirichlet

$\hat{u_D}$: função u restrita a u^D

$\hat{u_0}$: função pertencente a $H^1(\Omega)$ que atende condições de contorno não homgêneas

u_h: solução aproximada de u

$f, f(x, y, z)$: função definida no espaço tridimensional
$g, g(x, y, z)$: idem a f
v: função teste utilizada na definição da formulação fraca
$e(x)$função erro de aproximção
i, j, k: índices
h_k: maior distância euclidiana entre nós do elemento k
$r(x)$: função resíduo
e_{hi}: base de funções para a definição do espaço V_h

Letras Latinas Maiúsculas

$\mathbb{R}^3$: Espaço Euclidiano tridimensional
$\mathbf{H}$: Espaço de funções de Hilbert
$B(u, v)$: Operador bilinear em u e v
$L(v)$: Operador Liner
V: Espaço de funções teste (dimensão infinita)
V_h: Espaço finito de funções teste ($V_h \subset V$)
H_h: Dimensão do espaço V_h
$S_{i,j}$: Matriz de rigidez
C, C_1, C_2: constantes pertencentes a $\mathbb{R}$
$\mathcal{P}$: Partição do domínio

Letras Gregas Minúsculas

α_{hi}: coeficiente multiplicador da iésima função de forma do espaço V_h
η: norma da função erro calculada sobre o domínio Ω
η_k: estimativa de erro para o k-ésimo elemento
ρ_k: diâmetro do maior círculo inscrito no elemento k

Letras Gregas Maiúsculas

Γ: contorno do domínio Ω
Γ_D: contorno do domínio submetido a condição de contorno Dirichlet
Γ_N: contorno do domínio submetido a condição de contorno Neumann
κ_k: índice de forma do elemento k
Ω: domínio de interesse

Ω^k: sub domínio do element k

Θ : índice de efetividade do estimador de erros

Capítulo 1

Introdução

1.1 Introdução

Os métodos *hp* adaptativos apresentam como atrativo ao seu uso uma alta taxa de convergência em relação aos refinamento h ou p isolados. O problema a ser resolvido é como minimizar o erro de aproximação inserindo o menor número de graus de liberdade. Em 1987 Devloo e Oden [23] apresentaram uma implementação do método, entretanto, a elevada complexidade de uma implementação de um código *hp*, bem como o custo computacional associado à busca automática de um padrão *hp* ótimo, restringiu a pesquisa a poucos centros.

Outro ponto a destacar é o fato dos novos modelos industriais estar se tornando cada vez mais realistas, reduzindo as simplificações de cálculo, procurando assim resultados mais próximos à realidade podendo desse modo justificar a otimização do uso de materiais. Tal abordagem pode levar a uma redução dos coeficientes de segurança relativos à precisão do cálculo. Isso implica na necessidade de aumento da precisão do modelo numérico. Ferramentas auto adaptativas podem ser utilizadas com esse intuito.

Com relação ao tempo para obtenção de um modelo discreto, com uma dada precisão, vale destacar que em um ciclo de projeto industrial, a cada novo modelo gerado há a necessidade de se realizar todo um conjunto de testes de validação e qualificação do modelo antes de se utilizar seus resultados, tais atividades requerem um tempo que nem sempre está disponível ao projetista. Assim, o tempo para se obter a malha adequada é reduzido, o que justifica o desenvolvimento de ferramentas auto-adaptativas.

Para problemas de transporte com evolução temporal, a necessidade de auto-adaptatividade é ainda mais notada, uma vez que as regiões da malha que necessitam de uma maior discretização mudam ao longo dos passos de tempo. Caso não se tenha uma estratégia de aglomerar elementos de baixo erro e refinar elementos de alto erro, a aproximação poderá não ter a precisão adequada ou o problema poderá ter uma dimensão que irá inviabilizar a obtenção da solução em um tempo aceitável.

1.2 Objetivos

O principal objetivo do trabalho é disponibilizar uma estrutura para o desenvolvimento de métodos *hp* auto-adaptativos no ambiente PZ [18], que possa ser utilizada independentemente de: formulação variacional, tipo de elemento utilizado, método de resolução etc. Propõe-se uma estrutura onde se define a interface requerida de um estimador de erros, bem como a interface para a seleção do padrão de refinamento. Tal interface contempla a possibilidade de análise de malhas com elementos contínuos ou descontínuos.

O segundo objetivo desse trabalho é prover uma implementação que possibilite o processamento do método em máquinas paralelas, de modo que o tempo de obtenção de uma malha adaptada seja aceitável para aplicações industriais. A estrutura implementada possibilita que o processo de cálculo do erro e a definição dos padrões de refinamento sejam feitos utilizando processamento paralelo, em ambientes com memória compartilhada ou distribuída.

Adicionalmente, em termos de escolha de padrão de refinamento, propõe-se agregar um método de seleção do padrão de refinamento tendo por base uma biblioteca de padrões admissíveis. A escolha do padrão de refinamento é um dos desafios da auto-adaptatividade (ver Zienkiewicz [55]). A possibilidade de escolha entre diversos padrões agrega flexibilidade mas a definição de que padrão utilizar implica em um desafio que, caso bem solucionado, poderá levar a malhas cujo erro devido à discretização do domínio e escolha do espaço de interpolação tenham uma relação de erro por graus de liberdade próxima à ótima.

1.3 Motivação

Métodos auto adaptativos podem conduzir a malhas com precisão adequada com um número de graus de liberdade ótimo. Para tal, utiliza-se um estimador de erros para indicar quais elementos precisam ser adaptados e também para se ter um parâmetro indicativo da qualidade da solução. O problema é que o custo computacional do estimador de erros pode ser elevado, independentemente da abordagem utilizada tais como "*patch recovery techniques*", "*goal oriented*" etc.

Com relação à definição de como adaptar os elementos, os métodos tradicionais: h, p, hp e r, serão acrescidos de outros modos de adaptação, incluindo a definição por utilização de espaços de aproximação mistos com elementos com espaço de interpolação contínuos e outro utilizando espaços de aproximação do tipo Galerkin Descontínuo, tal qual está sendo desenvolvidas no grupo de pesquisa do LabMeC.

Mesmo no caso de refinamento h, a disponibilidade de padrões de refinamento direcionais agregam complexidade a tomada de decisão sob a forma de adaptação do elemento, conforme descrito em [55].

A implementação de métodos auto adaptativos utilizando processamento paralelo é uma abordagem natural para resolver a questão do custo do processo.

O processo de análise do padrão de refinamento, proposto originalmente em [16], tomando por base a análise do erro nas arestas de um elemento conduz a parâmetros que podem ser utilizados na análise do padrão de refinamento a adotar, inclusive no caso de utilização de padrões de refinamento direcional.

A estrutura aqui proposta implementa um processo adaptativo baseado na análise do erro em *patches* de elementos. Tal análise baseia-se na comparação de dois espaços de aproximação, um o espaço original e outro o correspondente a solução para uma malha uniformemente refinada. A utilização de *patches* simplifica a implementação utilizando processamento paralelo, enquanto a solução uniformemente refinada é utilizada não só no cálculo do erro como também na definição dos padrões de refinamento.

1.4 Organização do Trabalho

O presente trabalho está estruturado em cinco partes: introdução, revisão bibliográfica e descrição da implementação proposta, casos de validação e conclusões conforme descrito na seqüência:

- Parte 1: Capítulo 1 - Introdução:
  - Introdução, objetivos, motivação e organização do trabalho;
- Parte 2: Capítulo Revisão Bibliográfica:
  - Revisão geral sobre os temas envolvidos:
    - Métodos numéricos e computacionais, com foco nos assuntos que são necessários para o desenvolvimento do trabalho
    - Ferramentas Computacionais:
      - Programação Orientação a Objetos
      - UML
      - Log
      - Programação paralela
    - Ambiente de programação de elementos finitos orientado a objetos PZ ([20, 21, 22, 24]);
    - Ambiente de programação científica em paralelo OOPar ([10]);
- Parte 3: Implementações:
  - Capítulo 3 - Padrões de refinamento baseados em malhas contendo um exemplo de padrão de refinamento no ambiente PZ;

  - Capítulo 4 - Método auto adaptativo implementado

- Parte 4: Capítulo 5 - Casos de validação
- Parte 5: Capítulo 6 - Conclusões e perspectivas de desenvolvimento
- Anexos: reproduz-se os textos do trabalho de mestrado [49] que iniciou essa pesquisa por serem necessários ao entendimento desse trabalho:
  - Metodologia de definição de malhas patches e
  - Metodologia de definição do padrão de refinamento baseado em análise de arestas.

Capítulo 2

Revisão Bibliográfica

Esse trabalho aborda tópicos de diversas áreas, destacando-se a ciência da computação, tópicos de análise numérica, métodos *hp* em paralelo e conhecimento de bibliotecas de programação científica. Assim, de modo a poder organizar essa seção, dividir-se-á a seção em quatro partes a saber: aspectos relacionados a análise numérica, aspectos relacionados à ciência da computação, métodos *hp* em paralelo e bibliotecas de programação científica utilizadas no trabalho. A descrição dos tópicos acima listados está organizada da seguinte maneira:

1. Análise Numérica
   (a) Método dos Elementos Finitos,
   (b) Estimadores de Erro,
   (c) Adaptatividade / Critérios para auto-adaptatividade, baseados em análise de estimativas de erro,
2. Ciência da Computação:
   (a) Programação Orientada a Objetos
   (b) UML - *Unified Modeling Language*,
   (c) *Logging*,
   (d) Comunicação e *Sockets*,
   (e) Modelos de Paralelismo,
   (f) Performance;
3. Desenvolvimento de adaptatividade *hp* em paralelo: definições, abordagens e modelos existentes.
4. Bibliotecas de programação científica:

(a) Ambiente de Programação Científica Orientado a Objetos PZ;

(b) Ambiente de Programação Paralela Orientado a Objetos OOPar.

Os ambientes de desenvolvimento, de elementos finitos PZ [24], bem como o de paralelismo OOPar [10], são brevemente descritos aqui. Uma descrição mais aprofundada desses ambientes pode ser obtida em Apostilas PZ e OOPar `http://labmec.fec.unicamp.br/pz`.

2.1 Aspectos Relacionados à Análise Numérica

2.1.1 O Método dos Elementos Finitos (MEF)

Histórico

O MEF é um método sistemático para a obtenção de uma aproximação para um problema de valor de contorno. Os problemas de valor de contorno são encontrados em diversos problemas físicos regidos por leis de conservação.

Historicamente, há uma certa dificuldade em precisar os primórdios do MEF. Nessa descrição, tomar-se-á por base os trabalhos: Babuska [53], Devloo [17], Soriano [52] e Assan [3]. No trabalho de Devloo [17], que baseou-se em Williamson [33], coloca-se como primórdios do MEF os métodos de Leibniz e Schellbach para solução do problema de caminho ótimo de descida de um corpo sobre uma superfície sujeito somente a força gravitacional (*brachistochrone problem*). Nesses trabalhos é utilizado o conceito de discretização do domínio e a aproximação em cada um dos subdomínios gerados é linear.

Em 1928, Courant, Friedrichs e Lewy apresentaram o artigo fundamental para o método de diferenças finitas [47], onde eles utilizavam o sistema algébrico gerado para demonstrar a existência da solução. A solução do sistema algébrico indicava uma aproximação para a resposta do problema.

Há consenso na bibliografia de que o primeiro artigo que definiu um método muito próximo ao que se conhece hoje por MEF foi o artigo publicado em 1943 por Courant [12]. Foi definida uma formulação variacional equivalente à equação diferencial original. Notadamente mais simples de se resolver que a equação diferencial original, Courant analisou os diversos métodos numéricos de resolução do problema variacional. A primeira abordagem, foi o método de Rayleigh-Ritz [3], o qual apresenta baixa convergência para problemas com altas ordens de derivadas. Isso indicou que a convergência do método dependia da função de aproximação escolhida. Foi apresentada uma forma alternativa ao método de diferenças finitas, onde se destaca que, apesar de o método alternativo levar a sistemas de equações mais complicadas, esse método pode ser utilizado em um conjunto maior de problemas executando-se uma quantidade pequena de cálculo.

Até aqui, os estudos citados são todos relacionados a estudos matemáticos. Com algum atraso os engenheiros chegaram a uma abordagem próxima às anteriormente citadas. Da mesma forma que se identificam os primórdios matemáticos do MEF em trabalhos relacionados a discretização

de domínios, em 1941 surgiu o primeiro trabalho desenvolvido por um engenheiro relacionado ao MEF, (Hrennikoff, citado por [52]), onde é proposto um método para aproximação de uma placa por meio de um número arbitrário de barras. Cada barra possuía formulação conhecida e tal formulação serviu como função de aproximação. Vale ressaltar que esse problema recaia na resolução de um sistema algébrico.

Em 1954, Argyris apresentou um conjunto de artigos na revista *Aircraft Engineering* onde foi formulado o Princípio dos Trabalhos Virtuais (PTV) e Trabalho Virtual Complementar. O interessante é que o resultado do PTV em problemas elásticos corresponde a formulação variacional do problema. Um dos exemplos de aplicação do método foi a definição de um método matricial para a resolução de um problema de placa.

O trabalho de Turner, Clough, Martin e Topp [39] apresenta o MEF na forma como se conhece hoje, sendo o trabalho de Clough, em 1959, o primeiro trabalho a utilizar a denominação de "Elementos Finitos".

Ao final da década de 1960, o MEF já estava generalizado e consagrado como uma ferramenta de engenharia. Com o interesse pelo método, pesquisas foram desenvolvidas para tentar provar a convergência e unicidade do método. Nesse período foram redescobertos trabalhos como o de Courant e dentre outros o de Galerkin.

As pesquisas relacionadas ao MEF se expandiram desde então. No início as pesquisas concentraram-se em padrões de armazenamento de matrizes com objetivo de minimizar o uso de memória. Posteriormente, preocupações relacionadas a precisão da aproximação tornaram-se o foco das pesquisas. Métodos de cálculo de estimativas de erro ainda hoje são foco de pesquisa, tais como os métodos *patch revorery.* Segundo o Prof. Zienkiewicz, em seu artigo [55], o problema de estimativa de erro já é um problema bem conhecido e estudado devendo as pesquisas de elementos finitos se preocupar em como adaptar a malha para reduzir o erro de maneira efetiva. Tal aspecto é um dos focos desse estudo.

Apresentação do MEF

Segundo [43], o MEF é uma técnica de obtenção de soluções aproximadas para problemas de valor de contorno. O método envolve a divisão do domínio em um número finito de sub-domínios, os elementos finitos, e utilizando conceitos variacionais construir uma aproximação da solução sobre a partição de elementos. Simplificadamente, a metodologia do MEF, conforme descrita em [43], consiste nos seguintes passos:

- definição da equação diferencial: em geral provém de uma lei de conservação e inclui as informações de contorno;
- obter a formulação fraca do problema equivalente;
- escolha de um espaço de funções de aproximação adequado;

- aplicação do método de Galerkin para o espaço de funções adotado, resultando em um sistema algébrico;

- resolução do sistema algébrico: consiste na resolução do sistema linear gerado pelo método de Galerkin;

- análise da solução obtida: o resultado obtido é uma aproximação para a solução real do problema. Assim, há a necessidade de se verificar a precisão dessa aproximação.

Cabe ressaltar que a descrição acima difere de abordagens tradicionais, tais como as apresentadas em [55],[56], [52] e [3] onde a formulação fraca e o espaço de aproximação adotados são embutidos no cálculo da matriz de rigidez e do vetor de carga do elemento, sendo o método descrito para cada tipo de elemento, i.e. quadriláteros de 4, 8 e 9 nós, triângulos de 3 e 6 nós etc. Tal abordagem é utilizada em programas de elementos finitos comerciais tais como o Ansys `http://www.ansys.com` e o Nastran `http://www.mscsoftware.com/products/msc_nastran.cfm`, onde o tipo de elementos escolhido indica a formulação a ser resolvida e o tipo de função de forma que faz parte do espaço de aproximação. A vantagem no uso dessa abordagem é que a facilidade de implementação de um tipo específico de equação. Em contrapartida, a implementação de métodos adaptativos *hp* torna-se uma tarefa complexa.

A abordagem proposta por Becker, Carey e Oden [43] é adequada ao entendimento do MEF na forma como ele está implementado no ambiente PZ [24], ambiente esse que será utilizado no desenvolvimento do trabalho.

Para descrever a metodologia proposta por Becker, Carey e Oden é utilizado um problema modelo, sendo desenvolvidas todas as etapas consideradas acima.

Definição Problema Modelo - Equação de Laplace

O problema que se pretende aproximar é um problema diferencial de valor de contorno. Consideremos um domínio Ω em $\Re^3$, submetido a duas condições de contorno (CDC) em suas faces denominadas: Γ_D e Γ_N, sendo:

- Γ_D: Condição de contorno de Dirichlet ou essencial, onde o valor da variável de estado é fixada;

- Γ_N: Condição de contorno de Neumann ou natural, onde o valor da função fluxo no contorno é fixada;

- $\Gamma = \Gamma_D \cup \Gamma_N \, e \, \Gamma_D \bigcap \Gamma_N = \emptyset$ o contorno completo do domínio é uma partição das fronteiras submetidas às condições de Neumann e de Dirichlet.

O problema modelo consiste em encontrar uma função $u(x, y, z)$, $(x, y, z) \in \Omega$ tal que:

$$\begin{cases} -\Delta u = f \ \forall \ (x,y,z) \in \Omega \\ u = u^D \ \forall \ (x,y,z) \in \Gamma_D \\ \frac{\partial u}{\partial n} = g \ \forall \ (x,y,z) \in \Gamma_N \end{cases} \tag{2.1}$$

onde:

- $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$: Operador Laplaciano;
- $\frac{\partial u}{\partial n} = \frac{\partial u}{\partial x}\overrightarrow{n_x} + \frac{\partial u}{\partial y}\overrightarrow{n_y} + \frac{\partial u}{\partial z}\overrightarrow{n_z}$[1] : indica o fluxo normal ao contorno de Ω;
- $f = f(x,y,z),\ (x,y,z) \in \Omega$: função definida para todo o domínio.
- $u^D = u_o(x,y,z),\ (x,y,z) \in \Gamma_D$: condição de contorno Dirichlet, onde u_0 é uma função que define um valor fixado para a variável de estado nesta região do contorno;
- $g = g(x,y,z),\ (x,y,z) \in \Gamma_N$.: função fluxo imposto na fronteira submetida à CDC de Neumann, onde $g(x,y,z)$;
- $\Omega = \Gamma_D \bigcup \Gamma_N,\ \Gamma_D \bigcap \Gamma_N = \emptyset$.

Formulação Fraca

Por simplicidade, inicialmente, será aqui mostrada a metodologia para a obtenção da formulação fraca para o caso da CDC Dirichlet homogênea, ou seja: $u_0(x,y,z) = 0 \, \forall \, (x,y,z) \in \Gamma_D$. Neste caso, o espaço de aproximação é formado pelo espaço linear cuja funções satisfazem a CDC Dirichlet.

Sendo $v = v(x,y,z)$ uma função teste utilizada, esta se anulará no contorno Γ_D, ou seja:

$$v(x,y,z) = 0 \ \forall \ (x,y,z) \in \Gamma_D \tag{2.2}$$

Multiplicando-se a expressão (2.1) pela função teste v e integrando o resultado sobre todo o domínio Ω, temos:

$$\int_\Omega -\Delta u \ v \, d\Omega = \int_\Omega f \ v \, d\Omega \tag{2.3}$$

onde Δ indica o laplaciano.

Considerando que a função teste v é contínua e que $v(s) = s \ s \in \Gamma_D$, pode-se integrar por partes o lado esquerdo da expressão acima, tendo como resultado:

$$\int_\Omega \nabla u \nabla v \, d\Omega - \int_{\Gamma_N} \frac{\partial u}{\partial n} v \, d\Gamma_N = \int_\Omega f \ v \, d\Omega \tag{2.4}$$

onde ∇ denota o gradiente da função.

[1] $\overrightarrow{n_x}$, $\overrightarrow{n_y}$ e $\overrightarrow{n_z}$ indicam os vetores unitários normais às superfícies de contorno.

A integral no contorno fica reduzida apenas ao trecho onde é aplicada a CDC Neumann pelo requerimento de que a função v é nula na região do contorno onde está aplicada a CDC Dirichlet (homogênea). Define-se $\overset{0}{H}{}^1$ o espaço de funções v que atende os requerimentos da CDC Dirichlet.

Desta forma, o problema inicial passa a ser representado agora pelo seguinte problema equivalente:

$$\begin{cases} \text{Encontrar } u(x,y,z) \in H^1(\Omega), \text{ tal que} \\ \int_\Omega \nabla u \, \nabla v \, d\Omega = \int_{\Gamma_N} g \, v \, d\Gamma_N + \int_\Omega f \, v \, d\Omega \\ \text{para toda função teste } v,\ v \in \overset{0}{H}{}^1 \end{cases} \tag{2.5}$$

Para que ocorra a equivalência entre a formulação fraca e a formulação forte, há a necessidade de que a função teste adotada deve pertencer ao *espaço de funções teste admissíveis*, ou seja respeite as CDC Dirichlet e que as funções f e g devem ser quadrado integrável em Ω e Γ_N respectivamente.

Um aspecto importante a ser destacado é a possibilidade de representar a formulação fraca através de um termo bilinear e um termo linear conforme mostrado abaixo:

$$\begin{cases} B(u,v) = \int_\Omega \nabla u \, \nabla v \, d\Omega \\ L(v) = \int_{\Gamma_N} g \, v \, d\Gamma_N + \int_\Omega f \, v \, d\Omega \\ \text{e assim:} \\ B(u,v) = L(v) \end{cases} \tag{2.6}$$

Essa representação é útil no estudo de convergência de estimadores de erro que será abordado posteriormente.

CDC Dirichlet Não Homogênea Neste caso teremos que $u_0 \neq 0$ e assumindo que a função u_0 admite a "translação" para $\widetilde{u}_0$, definida em todo o domínio Ω e satisfazendo todas as condições de regularidade impostas para a solução, temos que $\widetilde{u}_0$ deverá estar contida dentro do espaço de Sobolev $H^1(\Omega)$.

Desta forma, com a aplicação de $\widetilde{u}_0$ à formulação fraca obtida para o caso homogêneo teríamos:

$$\begin{cases} \text{Encontrar } u(x,y,z) \ \in \widetilde{u}_0 + V, \text{ tal que} \\ B(u,v) = L(v) \\ \text{para toda função teste } v \in V \end{cases} \tag{2.7}$$

onde:

$$\widetilde{u}_0 + V = \{\widetilde{u}_0 + v,\ v \in V\} \tag{2.8}$$

Desta forma, tendo-se uma função $\widetilde{u}_0$ que satisfaça as condições impostas acima, temos que a função resultante u independe desta extensão, o que torna possível a seguinte substituição de variáveis: $u = \widetilde{u}_0 + w$, com $w \in V$ e satisfazendo todas as condições de continuidade impostas a u.

Com isto, a formulação variacional pode ser reescrita da seguinte forma:

$$\begin{cases} \text{Encontrar } w(x,y,z) \in V, \text{ tal que} \\ B(w,v) = L(v) - B(\widetilde{u}_0, v) \\ \text{para toda função teste } v \in V \end{cases} \tag{2.9}$$

Ressalta-se que, desta forma, pode-se calcular a função $\widetilde{u}_0$ que satisfaz a condição de extensão e então, definindo-se : $L_{mod} = L(v) - B(\widetilde{u}_0, v)$, como sendo a parte linear modificada, podemos substituir na expressão acima e voltar a ter o mesmo padrão de expressão que aquele encontrado para as CDC homogêneas.

Método de Galerkin

Sendo disponível a formulação fraca para o problema, o método de Galerkin consiste na restrição do espaço de funções teste V, utilizando um espaço $V_h \subset V$, espaço este composto por uma base de dimensão finita. Assim o problema a ser resolvido passa a ser:

$$\begin{cases} \text{Encontrar } u_h(x,y,z) \in \widetilde{u}_0 + V_h, \text{ tal que} \\ B(u_h, v_h) = L(v_h) \\ \text{para toda função teste } v_h \in V_h \end{cases} \tag{2.10}$$

Com relação à base de funções utilizada, esta pode ser representada por:

$$V_h = \{e_{hi}\},\ i = 1, 2, \ldots, N_h \tag{2.11}$$

onde $N_h = dim(V_h)$ indica a dimensão do espaço de aproximação.

Dentro deste espaço, procura-se a solução para o problema sob a forma da seguinte combinação linear:

$$u_h = \sum_{i=1}^{N_h} \alpha_{hi} e_{hi} \tag{2.12}$$

Os coeficientes α_{hi}, a serem determinados, são denominados *graus de liberdade* (d.o.f).

Substituindo a expressão acima na formulação fraca do problema e, ainda, adotando como funções teste a mesma base de funções utilizada para representar u_h, ou seja $v = e_{hj}\ \ j = 1, 2, \ldots, N_h$, chega-se à seguinte representação do problema:

$$\begin{cases} \text{Encontrar } \alpha_{hi},\ i = 1, 2, \ldots, N_h, \text{ tal que} \\ B(\sum_{i=1}^{N_h} \alpha_{hi}(e_{hi}, e_{hj})) = L(e_{hj}) \\ j = 1, 2, \ldots N_h \end{cases} \tag{2.13}$$

O método de Galerkin consiste assim em uma estratégia de aproximação de equações diferenciais, em que, partindo-se da formulação fraca do problema, utiliza-se um espaço de funções teste

de dimensão finita. Nesse caso, o problema pode ser escrito como um problema algébrico, onde se buscam os coeficientes multiplicadores das funções teste.

Em princípio, a aproximação depende somente do subespaço adotado V_h, independendo da base de funções escolhida e_{hi}. Na prática, a escolha da base de funções afeta o condicionamento do sistema final, implicando diretamente sobre os erros de arredondamento, os quais podem ser significativos [14, 15].

Para o problema em questão, podem ser realizadas algumas mudanças de notação conforme segue:

- Definindo-se a matriz de rigidez global S_{ij} como sendo:

$$S_{ij} = B(e_i, e_j) = \int_\Omega \nabla e_i \nabla e_j \, d\Omega \tag{2.14}$$

- O vetor de carga modificado L_j^{mod} será dado por:

$$L_j^{mod} = L(e_j) - B(\widetilde{u}_0, e_j) = \int_\Omega f e_j \, d\Omega + \int_{\Gamma_N} g e_j \, d\Gamma_N - \int_\Omega \nabla \widetilde{u}_0 \nabla e_j \, d\Omega \tag{2.15}$$

- O vetor de carga original L_j é dado por:

$$L_j = L(e_j) = \int_\Omega f e_j \, d\Omega + \int_{\Gamma_N} g e_j \, d\Gamma_N \tag{2.16}$$

E assim o problema pode ser escrito da forma usual:

$$\begin{cases} \text{Encontrar } \alpha_{hi},\ i = 1, 2, \ldots, N_h, \text{ tal que} \\ \sum_{i=1}^{N_h} \alpha_{hi} S_{ij} = L_j^{mod} \\ j = 1, 2, \ldots N_h \end{cases} \tag{2.17}$$

Este sistema pode ser representado por $[S][\alpha] = [L]$, ou seja, um sistema linear.

Precisão da Aproximação e Estimativa de Erro

O resultado do MEF é uma função que aproxima a função procurada, utilizando o método de Galerkin tomando um subespaço V_h conforme descrito acima. O erro da aproximação é uma função definida pela diferença entre a solução exata e a solução aproximada:

$$e(x) = u(x) - u_h(x) \tag{2.18}$$

onde:

- $u(x)$: valor da função u(real) calculada no ponto x;
- $u_h(x)$: valor da aproximação calculada no ponto x;

Para analisar a qualidade da aproximação é necessário se mensurar essa função erro, sendo o meio natural para tal a utilização de uma norma [6]. Para funções, uma norma é um funcional associado à função, representado por ||f||, onde se lê norma de f, que satisfaz um conjunto de axiomas, a saber [36]:

1. Uma norma é sempre positiva: $||f|| \geq 0$;
2. Se a função $f \equiv 0$ então: $||f|| = 0$. Por outro lado, se a norma é nula então a função é igual a função 0;
3. Multiplicação por escalar: $||\alpha\, f|| = |\alpha|\, ||f||$, $\alpha \in \mathbb{R}$
4. Desigualdade triangular: $||f + g|| \leq ||f|| + ||g||$

Conforme [43], as três principais normas utilizadas em análises de elementos finitos a saber:

- Norma L^2: $||f||_2 = \sqrt{\int_0^1 f^2 dx}$
- Norma infinito: $||f||_\infty = \underset{0 \leq x \leq 1}{max} |f(x)|$
- Norma energia $||f||_E = \sqrt{\frac{1}{2} B(f, f)}$

Um fato importante para o uso do MEF é a convergência do método. Ou seja, dado um problema inicial, sendo discretizado o domínio e definidas as condições de contorno, tem-se uma aproximação inicial e associda a essa um erro inicial. Refinando-se uniformemente a malha, ter-se-á uma nova solução e um novo erro. Repetindo-se esses procedimento, pode-se definir uma seqüência de erros em função da discretização da malha. Conforme [43], tal seqüência tende para zero, uma vez que a solução aproximada tende para a solução real, sendo mostrado que a estimativa da norma do erro pode ser colocada da seguinte forma:

$$||e|| \leq Ch^p$$

onde C é uma constante que depende do problema analisado, h representa a dimensão do elemento adotado e p é um inteiro que depende da base de funções escolhidas para a aproximação.

Dimensão e forma de um elemento

Tanto dimensão como forma do elemento são parâmetros que influenciam a aproximação. Entretanto, para definir estes parâmetros, devemos ter em mente que diversos tipos de elementos finitos podem ser utilizados, sendo necessário assim uma padronização da nomenclatura utilizada. Aqui será adotada a definição dada em [1].

Assim, define-se:

- $h_k = max_l\{h_l\}$, $h_l = sup_{x_i,x_j \in k}|x_i - x_j|$, sendo h_l a máxima distância entre os nós do elemento k, distância esta tomada dois a dois. Desta forma este parâmetro indica o diâmetro do círculo circunscrito ao elemento;

- ρ_k: indica o diâmetro de um círculo inscrito ao elemento k;

- $\kappa_k = \frac{h_k}{\rho_k}$: índice que indica a forma do elemento, definido como a regularidade do elemento k.

2.1.2 Estimadores de Erro

Para se estimar o erro tem-se duas possibilidades: uma o conhecimento prévio do comportamento do erro para uma determinada norma ou ainda, partindo-se do princípio de que quanto maior o enriquecimento de um espaço de aproximações, melhor será o resultado.

Assim, caso se compare o resultado obtido da utilização de um espaço de aproximação com parâmetros h_0 e p_0, com aqueles obtidos através de um espaço enriquecido $\widetilde{h}_n$-$\widetilde{p}_n$, onde $\widetilde{h}_n \leq {h_0}^2$ e $\widetilde{p}_n \geq p_0$, poderíamos calcular o erro entre as duas aproximações sabendo que a última é de melhor qualidade que a primeira.

Os estimadores de erro podem ser divididos em dois grupos: *a priori* e *a posteriori* [1]. Os estimadores de erro do tipo *a priori* são baseados em estimativas, tais quais o valor da segunda derivada da função procurada por exemplo. No caso de aplicações de maior complexidade, onde o comportamento das funções analisadas não é totalmente conhecido, o uso desse tipo de estimador não é adequado.

Os estimadores de erro do tipo *a posteriori* tem sido desenvolvidos nas últimas três décadas, tendo como primeiro trabalho de destaque [4], onde o estimador de erro baseia-se nas aproximações da norma de energia do erro em cada elemento K da malha.

O uso da formulação de energia complementar para a obtenção *a posteriori* do erro foi feita por [13]. Entretanto, uma vez que seu método era baseado no computo global da solução, este tornava-se muito oneroso em termos de tempo de processamento. Este problema foi contornado por Ladevèze e Leguillon [37] que realizaram os cálculos baseados na energia complementar de cada elemento, além do uso do conceito de dados de equilíbrio de contorno.

Diversos trabalhos foram feitos utilizando diversos tipos de estimadores de erro, destacando-se o de Zienkiewicz e Zhu [57], cujo estimador de erro, baseado na obtenção da diferença entre o gradiente de uma solução suavizada e o gradiente calculado originalmente, mostrou-se eficaz para aproximações de ordem $p = 1$, tornando-o muito popular.

[2]Ressalta-se que o parâmetro h está relacionado a dimensão do elemento e assim, um parâmetro h menor implica em um domínio com maior discretização

Requisitos de Estimadores de Erros "*a posteriori*"

Para analisar as características básicas de estimadores de erro do tipo *a posteriori*, recorre-se a formulação fraca do problema modelo, mostrado anteriormente na equação (2.5).

Utilizando o subespaço V_h tem-se:

$$B(u_h, v_h) = L(v_h) \,\forall\, v_h \in V_h \tag{2.19}$$

Desta forma, conforme [1], a função erro, apresentada em (2.18), pertence ao espaço V e satisfaz:

$$B(e, v) = B(u, v) - B(u_h, v) = L(v) - B(u_h, v) \,\forall\, v \in V \tag{2.20}$$

Ainda, dada a condição de ortogonalidade do erro para projeções de Galerkin (ver [1]), tem-se que:

$$B(e, v_h) = 0 \,\forall\, v_h \in V_h \tag{2.21}$$

Levando-se em consideração as expressões (2.20) e (2.21), integrando-se a primeira por partes, em cada elemento:

$$\int_{\Omega^K} (\nabla e \nabla v) d\Omega^K = \int_{\Omega^K} (rv) d\Omega^K + \oint_{\partial\Gamma^K} (v\overrightarrow{n_k}.\nabla e_{|k}) d\Gamma^K \tag{2.22}$$

sendo:

e: função erro procurada;

v: função teste ou peso;

r: função residual ou resíduo $r = f + \triangle u$;

$\overrightarrow{n_k}$: vetor normal à face do elemento;

k: elemento em análise.

Resolvendo esta equação, considerando condições de continuidade das funções e de suas integrais, teremos:

$$\|e\| \leq C_1 \|r\|_{L_2(k)} + C_2 \|R\|_{L_2(\partial k)} \tag{2.23}$$

onde:

$R \approx \overrightarrow{n_k} \nabla e$,

C_1, C_2: constantes que dependem da malha, principalmente do tamanho do elemento (h_k);

Ainda, definindo-se:

- η_k: como a estimativa de erro em cada elemento k.
- η: como a estimativa de erro na malha de elementos finitos.

A estimativa de erro na malha pode ser obtida através da seguinte expressão:

$$\eta = \sqrt{\sum_{k \in P} \eta_k^2} \tag{2.24}$$

onde P é a partição adotada e k um determinado elemento da partição.

Para que um estimador de erros seja utilizável este deve respeitar a seguinte propriedade:

$$C_1 \|e\| \leq \eta \leq C_2 \|e\| \tag{2.25}$$

ou seja, a estimativa do erro deve convergir para zero à mesma taxa que o erro real converge.

A qualidade de um estimador de erros é medida através do índice de efetividade, dado pela relação:

$$\Theta = \frac{\eta}{\|e\|} \tag{2.26}$$

2.1.3 Adaptatividade

A adaptatividade é um processo pelo qual se busca a melhoria da qualidade de uma aproximação por meio do enriquecimento do espaço de aproximação. O enriquecimento do espaço de aproximação pode ser feito de diversos modos. Tradicionalmente os parâmetros que são alterados para modificação do espaço de aproximação são, conforme [23]:

- h: parâmetro relacionado ao "tamanho" do elemento, sendo associado à discretização da malha;
- r: parâmetro relacionado à disposição dos nós dos elementos na malha;
- p: parâmetro relacionado à ordem dos polinômios, de cada elemento, utilizados como base para o espaço V_h.

O refinamento h consiste na redução do tamanho dos elementos componentes da malha, enquanto o refinamento p consiste na elevação da ordem dos polinômios da base de funções teste. O refinamento hp, objeto deste trabalho, consiste no refinamento destes dois parâmetros para a mesma malha.

Não será tratado do refinamento r neste trabalho. O refinamento r consiste no posicionamento ótimo dos nós que definem a malha de elementos finitos.

Uma outra possibilidade de adaptação da malha que está sendo estudada no grupo de pesquisa é a mudança do tipo de espaço de aproximação em cada elemento, podendo se utilizar formulações contínuas ou descontínuas. Tal forma não será utilizada nesse trabalho.

2.1.4 Auto - Adaptatividade

A busca de formas de melhoria do espaço de aproximações de maneira automática está ligada à obtenção de um parâmetro que indique uma forma de enriquecimento do espaço de interpolação que melhore a qualidade da solução. O parâmetro que indica a qualidade da aproximação é a norma do erro. Entretanto, o erro real só pode ser obtido quando se tem o conhecimento prévio da solução do problema o que não é o caso prático, sendo assim necessária uma estimativa de erro. A obtenção desta estimativa de erro é, de certo modo, o cerne de qualquer método auto-adaptativo. Atualmente existem diversos estimadores de erro disponíveis na literatura conforme citado por Zienkiewicz em [55].

Os métodos *hp* adaptativos são os esquemas mais eficientes de refinamento para um grande conjunto de problemas. Entretanto, a definição de quais elementos refinar e com qual padrão de refinamento são duas incógnitas a mais quando se pretende usar esse método.

A definição de quais elementos refinar é um problema que é resolvido através do uso de algum medidor de erros, quer ele seja um indicador[3] ou um estimador de erros. Já a definição do padrão de refinamento a ser utilizado h, p ou hp é uma questão que depende do problema de valor de contorno em questão. O problema a ser resolvido nesse caso é como minimizar o erro inserindo o menor número de graus de liberdade / equações.

A criação de um problema de elementos finitos, *hp* adaptativo em paralelo não é algo simples. Para exemplificar o grau de complexidade de tal problema pode-se citar o projeto "*Modern Industrial Simulation Tools*" desenvolvido no Sandia Labs. [54] onde um projeto desse tipo foi cancelado devido ao término dos recursos e também por limitações de programação.

O ponto que se destaca aqui é que quando se quer definir quais são os problemas tecnológicos relacionados ao tema adaptatividade em paralelo, além das questões dos problemas *hp* adaptativos, outras questões relacionadas ao processamento paralelo, tais como balanceamento de carga e minimização de comunicação, são acrescidas ao problema e é nesse ponto que se pode verificar a diversificação do tema e, dessa forma, uma grande diversidade de abordagens para o tema.

2.1.5 Abordagens de Implementação de Métodos Auto-adaptativos Utilizando Paralelismo

Métodos hp

Patra [46] define como grande dificuldade da adaptatividade *hp* em paralelo a definição da estrutura de dados e a comunicação. Assim, formas de definição do particionamento inicial da malha e de como manter o balanceamento de carga nos processadores aproximadamente iguais são as principais contribuições dos trabalhos desse autor.

[3]Um indicador de erros é um medidor no qual não há prova matemática de que esse indicador converge para o erro real a medida que o espaço de aproximação é aumentado.

Demkowicz [45, 44], inicialmente apresentou um código auto adaptativo para malhas bidimensionais e posteriormente o código *hp* adaptativo para malhas tridimensionais compostas por hexaedros. Ambos trabalhos partiram de um código serial existente. Durante o desenvolvimento do código para malhas 2D, dados os requisitos de comunicação, as modificações na estrutura de dados aumentaram de tal forma que uma nova estrutura de dados foi proposta, sendo aproveitado da implementação original os algoritmos. Tal abordagem serviu de base para o desenvolvimento do código tridimensional.

Métodos h e Galerkin Descontínuo

Remacle [48] propõe uma forma de definição de uma malha em um ambiente paralelo, ou seja, mostra uma abordagem para a definição de uma estrutura de dados contemplando o caso de memória distribuída. A idéia proposta na biblioteca AOMD é a definição de vizinhanças através de vértices, onde os vértices tem uma numeração única nos diferentes processadores. A consistência da malha durante os refinamentos (apenas h) é feita através do refinamento do elemento, com a criação de vértices em todos os domínios onde existam cópias dos vértices do elemento refinado. Dessa forma, pode-se identificar como ponto de interesse no trabalho desse autor a forma de comunicação da necessidade de criação de um determinado vértice em um determinado domínio. A comunicação é feita exclusivamente pelos elementos de interface entre os subdomínios, de modo que alterações em suas estruturas de dados são propagadas para suas cópias remotas.

Flaherty, Loy, Shephard e Teresco [26] mostram uma abordagem de adaptatividade em paralelo para o método de Galerkin Descontínuo. Tal método captura de maneira eficiente descontinuidades tendo pequena necessidade de comunicação, uma vez que cada interface só precisa conhecer os dois elementos que contribuem para o cálculo de seu fluxo. Destaca-se nesse trabalho o fato de que os elementos divididos, por necessidade de balanceamento de carga, podem migrar para outros processadores. Entretanto, em caso de necessidade de aglomeração, todos os filhos devem ser migrados para o mesmo processador para se proceder a aglomeração. Dessa forma, há a necessidade de reconstrução de partes das malhas que contém elementos filhos a aglomerar. Flaherty e Teresco [27] mostram especial interesse no problema de balanceamento de carga causado pela adaptatividade nesse artigo, abordando inclusive questões relacionadas a redes heterogêneas, ou seja, redes compostas por máquinas com características de processamento diferentes.

Narula [40] apresenta a biblioteca CHARM++, um ambiente para paralelização de adaptação de malhas. O código proposto apresenta uma abordagem para aproximações por diferenças finitas. A proposição dessa biblioteca, em termos de implementação é a utilização de mensagens[4] para a comunicação, onde, quando em uma arquitetura com memória distribuída, a mensagem é inserida como conteúdo de um *socket*, para a sua transmissão para um processador outro que não o seu

[4]Uma mensagem consiste em uma estrutura de dados representando um objeto encapsulada sob a forma de um vetor de *bytes*. De maneira geral, um objeto que necessite ser comunicado por meio de mensagens precisa saber como se "escrever" e, posteriormente, como se "ler" (*pack*/*unpack*).

processador corrente. As vantagens e desvantagens dessa abordagem em relação a pacotes de paralelização tradicionais são discutidas na seção relativa a Paralelismo. Em termos de adaptatividade, os padrões de adaptação são restritos a divisão uniforme de linhas, quadriláteros e hexaedros. O algoritmo proposto para adaptação impõe um nível de refinamento como máxima diferença entre níveis de refinamento de elementos vizinhos.

2.2 Aspectos Relacionados à Ciência da Computação

O desenvolvimento de um projeto como o aqui proposto envolve a existência e o domínio de diversas ferramentas computacionais. Aqui se opta pela utilização de ferramentas GNU (ver GNU `http://www.gnu.org`), de código livre, de modo a não atrelar o desenvolvimento à obtenção de licenças e autorizações de terceiros. Da mesma forma, o código gerado também deverá ser disponibilizado sob licença GNU.

Assim, o sistema operacional a utilizar é o *Linux*, os compiladores *GNU-gcc*, editores *Latex (Lyx* Lyx `http://www.lyx.org` e o *software* de visualização de resultados Open DX DX `http://www.opendx.org`, todos de domínio público e com documentação disponível para estudo. Os códigos gerados terão documentação escrita em padrão JavaDoc, de tal forma que geradores automáticos de documentação (e.g. Doxygen Doxygen `http://www.doxygen.org`) possam ser utilizados. Em termos de interface de desenvolvimento, propõe-se a utilização do ambiente KDevelop (Kdevelop `http://www.kdevelop.org`), cujo gerenciamento de projetos orientado a objetos (através de Makefiles) é facilitado.

Além dos *softwares* acima descritos, há a necessidade de gerenciamento de um grande número de tecnologias, em sua maioria de grande complexidade, assim, opta-se pela utilização de uma linguagem orientada a objetos de modo a tornar possível a utilização de todo o ambiente já desenvolvido pelo grupo de pesquisa no qual o projeto está inserido.

A descrição dos códigos a gerar utilizar-se-á da combinação de documentos descritivos, diagramas e da linguagem de modelagem unificada - UML. De modo a organizar os códigos implementados, será utilizado um serviço de controle de versão (CVS) para o armazenamento dos códigos gerados.

Outra tecnologia abaixo descrita essencial ao desenvolvimento do projeto é um sistema de controle de *log*, no qual foi utilizado o Log4cxx (Log4cxx `http://logging.apache.org/`) sem o qual a depuração de códigos distribuídos torna-se inviável.

Abaixo faz-se uma breve descrição da filosofia de orientação por objetos, da linguagem UML e dos sistemas de *log*, uma vez que a utilização desses é imprescindível ao desenvolvimento aqui proposto.

2.2.1 Programação Orientada a Objetos

As linguagens de programação consistem de uma sintaxe que um determinado compilador é capaz de transformar em um conjunto de instruções de máquina. Em geral, as diversas linguagens de programação implementam um conjunto de normas sintáticas similares (instruções para laços, verificações lógicas etc). Até os anos 80, quando do aparecimento do SMALLTALK [29, 35, 34], as linguagens de programação dominantes baseavam-se na programação procedural, que consistia de código seqüencial. O SMALLTALK inseriu o conceito de Orientação a Objetos (OO).

A filosofia de OO difere da programação procedural, comum em outros tipos de linguagens científicas tal como *Pascal*, por seu comportamento não ser ditado pela seqüência do código e sim pelo comportamento dos objetos componentes do programa. As principais vantagens da programação orientada a objetos estão relacionadas ao gerenciamento do código e à sua potencial reutilização.

A definição "clássica" de OO apresenta como características dessa abordagem os seguintes pontos:

- Encapsulamento: consiste em cada objeto apresentar uma série de dados e funções, cujo acesso é controlado, sendo somente permitido a cada objeto o acesso aos dados e funções públicas. Desse modo, o acesso e a modificação dos dados do objeto podem ser controlados, restritos a determinadas funções, tornando assim o gerenciamento dos dados mais seguro. Esse controle de acesso é o diferencial em relação aos conceitos existentes em programação procedural, tais como os *struct* em C e os *common blocks* em Fortran;

- herança / derivação: O conceito principal aqui envolvido é o conceito de especialização por meio de herança. Permite-se que sejam criadas classes derivadas de outras já existentes, tendo as classes derivadas a herança de todas as características da classe mãe, podendo ser implementados apenas os novos métodos específicos da classe e podendo ser reimplementados (sobrecarga) os métodos cujo comportamento na classe derivada é diferente do comportamento previsto na classe mãe. Em termos de métodos numéricos, um bom exemplo de derivação é o conceito de matriz. Toda e qualquer matriz tem alguns dados básicos que são suas dimensões e dados, já a forma de armazenar a matriz pode variar, desde o armazenamento esparso onde não são armazenados valores nulos até matriz cheia passando pelo armazenamento em banda. Em termos de OO, os dados básicos de uma matriz podem ser definidos na classe mãe, os requisitos específicos de cada armazenamento podem ser definidos nas classes derivadas.

- polimorfismo: esse conceito é uma extensão do conceito de sobrecarga de funções. O conceito de sobrecarga implica que diversos métodos com parâmetros distintos mas com mesma nomenclatura são distinguidos pelo compilador em função do "contexto" em que sejam utilizados, dessa forma a definição de um método não é dada apenas pelo seu nome, más sim pelo conjunto de nome mais parâmetros mais retorno. O polimorfismo é uma extensão desse

conceito, onde se pode definir métodos com nomes e parâmetros idênticos aos da classes mãe, sendo utilizada a função relativa ao tipo de objeto quando da chamada (*i.e.* objeto mãe acessa o método da classe mãe e objeto filho acessa o método definido na classe filha);

- *template*: o conceito de *template* é relativamente recente na linguagem de programação C++. Os *templates* implementam um conceito similar ao da derivação, sem o contraponto de acesso à métodos por meio de tabelas virtuais, o que torna a implementação com *templates* mais eficiente em termos de performance.

A opção pela utilização da filosofia de OO se deve ao fato da biblioteca de elementos finitos (PZ) bem como o ambiente de paralelização (OOPar) ser desenvolvidos em linguagem C++, utilizando-se de OO.

2.2.2 UML - *Unified Modeling Language*

A UML consiste da tentativa de criar uma linguagem para modelar um código computacional desde o seu planejamento até a sua execução. Em termos de documentação a UML pode ser vista como uma extensão dos fluxogramas, que são representativos para códigos procedurais mas podem não representar bem um código orientado a objetos.

Segundo [28] a UML surgiu através de um projeto da Rational Corporation que desenvolveu e unificou métodos de especificação e projeto de software OO, tendo sido aprovada pela OMG - Object Management Group em 1997.

A seqüência de implementação por meio dessa linguagem consistem em descrever (documentar) o comportamento das diversas partes do código que se deseja produzir, partindo dos níveis mais altos de abstração, onde são descritos comportamentos globais desejados, passando pela especificação dos tipos de objetos necessários gerar para se ter o comportamento anteriormente proposto. Cada objeto especificado tem então descritas suas interfaces externas (públicas) e internas (protegidas ou privadas), sendo essas interfaces responsáveis por implementar os algoritmos base do código.

As principais vantagens na utilização da linguagem UML diz respeito aos seguintes aspectos:

- independe da linguagem de programação a utilizar;

- todo o código pode ser planejado através de uma série de diagramas que podem descrever o comportamento global de um código, os comportamentos de objetos e a implementação de métodos propriamente ditos. Conjuntamente a descrição do comportamento do código podem ser gerados casos de uso que em termos de implementação consistem em possíveis testes de validação;

- induz ao desenvolvimento da documentação previamente à implementação do código, o que em grande parte dos casos conduz à identificação de problemas e inconsistências do código antes que estes ocorram;

- os diagramas UML representam o comportamento de programas orientados a objetos de melhor maneira que os fluxogramas tradicionais.

2.2.3 *Logging*

A inserção de *logs* em um código é considerada uma forma antiquada de depuração, entretanto em alguns casos, como no caso de computação paralela ou da depuração de códigos adaptativos, onde os problemas podem surgir no enésimo passo de refinamento, é uma das poucas formas de se obter a informação necessária para o entendimento e correção de falhas.

Programas *multithread* e em memória distribuída representam um desafio no aspecto relacionado a depuração, uma vez que não há mais um código onde as operações são seqüenciais e cujos depuradores tradicionais (*e.g.* gdb, ddd etc) tratam de maneira satisfatória [31]. Técnicas tradicionais tais como imprimir informações relativas ao código em tela ou em arquivo não são totalmente eficazes pelos seguintes aspectos:

1. Funções I/O (entrada e saída) tem um tempo de execução considerável, uma vez que há a necessidade de se transporta a informação da memória *cache* do processador até o dispositivo em questão (console, disco etc). Durante esse transporte se tem como restrição as velocidades de barramentos, a velocidade de escrita no dispositivo bem como a latência requerida pelo processo [8];

2. O tempo de execução de uma chamada I/O pode alterar a ordem de execução do código, e com isso alterar os resultados do código [8]. Para exemplificar isso, consideremos apenas duas instruções que estão em threads independentes: A e B. Sem chamadas I/O, suponha que a instrução A foi iniciada antes de B e termine antes de que B seja iniciada. Ao colocar uma chamada I/O em A, pode ocorrer que B seja iniciada antes de que A tenha sido finalizada. Caso A modifique informações que serão utilizadas por B, a estrutura de dados manipuladas por B com e sem as instruções de I/O em A são diferentes e por conseqüência seus resultados também o serão [25];

3. Em termos de programas em memória distribuída, há uma grande dificuldade em sincronizar informações de saída em processadores distintos e dessa forma, torna-se difícil a compreensão do estado global da estrutura de dados em cada momento [31].

De modo a tornar exeqüível a depuração de códigos paralelos, há a necessidade da utilização *logs* estruturados, de tal forma que seja possível se identificar o estado da estrutura de dados e a

seqüência de instruções que está sendo executada. Tal tarefa implica em planejamento de onde inserir as saídas de *log* e como gerenciar a informação produzida.

De maneira geral, é desejável algumas funcionalidades em uma ferramenta de *log,* conforme descrito em [31], a saber:

1. Possibilidade de implementação de níveis de informação: ou seja desde dados informativos sobre a seqüência de informações, mensagens de inconsistência de dados até a detecção de erros;

2. Tempo de execução e em qual processo (*thread*) está se gerando a informação;

3. Em caso de erros, em que arquivo e linha foi detectada a inconsistência;

4. Rastreamento de processos (*Tracing*);

5. Gerenciamento único dos *logs* produzidos, ou seja todos as informações geradas nos diversos processadores gerem as saídas de resultados em um único ponto.

A ferramenta que se propõe utilizar para tal finalidade é o Log4cxx (Log4cxx `http://logging.apache.org/log4j/docs/index.html`) biblioteca em linguagem C++ parte do projeto Log4j da Apache Software Foundation. As principais características dessa ferramenta são: configuração de destino do *log* (console, arquivo, *log* de sistema ou *socket*). Configuração de formato de *log* (texto simples, html ou xml). Hierarquia de *logs* (do nível mais alto para o mais baixo: FATAL, ERROR, WARNING, INFORMATION e DEBUG). Filtros de seleção de *logs* (por nível, por intervalos de nível e por verificação de *strings*). Outro aspecto interessante na utilização dessa ferramenta é que todas as configurações de saída são fornecidas por meio de um arquivo de configuração externo ao executável, ou seja não há necessidade de recompilar o código caso se queira um *log* mais detalhado de uma determinada função. Em termos de performance, o custo requerido para a verificação da necessidade de *log* é documentada, podendo assim ser medida em termos de performance total do código.

Essa ferramenta está, atualmente, sendo inserida tanto no ambiente PZ como no projeto OOPar tendo sido realizados testes para a verificação do envio de *logs* em modo remoto, sendo os resultados apresentados satisfatórios.

2.2.4 Conceitos de memória relacionados ao processamento

A eficiência de transferência de dados da memória para o processador, bem como o gerenciamento de memória são temas de muitas pesquisas. Isso é justificado pela performance de um código ser medida com base na relação entre o número de operações de ponto flutuante por segundo realizados e tal número depender diretamente da disponibilidade do dado a ser utilizado pelo processador.

O fato é que a velocidade de acesso à memória cresceu em velocidades muito menores que as velocidades de processamento nos últimos anos (ver [25]) o que torna esse tópico ainda mais importante para problemas de cálculo numérico.

Quando se fala de aqui de memória, deve-se ter em mente que a memória de grande parte dos computadores é composta por uma fila de sistemas de armazenamento, onde a velocidade de acesso à memória varia em cada um desses trechos em função de aspectos tecnológicos. Para exemplificar, [25] coloca os dados relativos a um computador DEC 21164 Alpha:

1. Registro de processamento: velocidade de acesso 2ns
2. Memória Cache L1 *on chip*: velocidade de acesso 4ns
3. Memória Cache L2 *on chip*: velocidade de acesso 5ns
4. Memória Cache L3 *off chip*: velocidade de acesso 30ns
5. Memória principal: velocidade de acesso 220ns

O preenchimento de cada nível de memória é feito em ciclos, onde em cada ciclo são realizadas dois tipos básicos de operação: preenchimento das pilhas de memória e execução das operações do código compilado. Caso o código necessite de um dado que não está no registro um ciclo de processamento é perdido (não é realizado o cálculo) para que o processador obtenha o dado na memória "*heap*" que pode ser a memória RAM ou memória virtual (disco rígido).

2.2.5 Paralelismo

O conceito básico por traz do paralelismo é o conceito de divisão, a qual pode ser de dados, de tarefas ou ambas. Um código paralelo pode ter como objetivo acelerar o tempo de resposta ou a divisão de uma estrutura de dados que não seria possível processar em um único computador [25].

A divisão de dados é utilizada quando a estrutura de dados do problema global não pode ser armazenada em um único computador (nó). O foco da resolução desse tipo de problema é a forma de dividir (particionar) o problema de modo a se obter a solução em menor tempo.

O problema típico de divisão de tarefas consiste de uma estrutura de dados, não muito grande, que pode ficar residente em um único computador. Os cálculos são em geral seqüencias de operações pré-determinadas. Problemas de otimização com diversas variáveis, onde não se tem informação sobre a sensibilidade do problema a cada variável, tem como possível abordagem a variação das diversas variáveis de modo a se ter uma resposta. O problema é que há a necessidade do cálculo de uma solução para cada combinação de variáveis. Nesse caso cada combinação pode ser processada em um nó.

Arquitetura do computador

Como descrito anteriormente, existem diversos tipos de problemas e soluções para cálculos em paralelo. Em termos de implementação, o fator preponderante é a arquitetura do computador paralelo, pois o ganho de performance, em relação a um código serial, está ligado ao conhecimento de tal arquitetura.

Em termos de arquiteturas paralelas, os principais modelos de arquitetura, conforme [25], são:

- memória compartilhada: são os computadores com mais de um processador (nó), onde estes processadores tem acesso ao mesmo banco de memória. A abordagem de programação para este tipo de arquitetura é a programação *multithreading*. A grande vantagem desse tipo de equipamento é a facilidade de programação, uma vez que não há problemas de minimização de comunicação. A desvantagem desse tipo de arquitetura é que o número de processadores é limitado e também o elevado custo dessas máquinas;

- memória distribuída: são computadores com diversos processadores, onde cada processador tem o seu próprio banco de memória. Os processadores são interligados internamente por meio de interfaces de comunicação de alto desempenho, aumentando a velocidade de tráfego de dados internamente [25]. A abordagem de programação para esse tipo de arquitetura é a de implementação com o uso de *sockets*. A desvantagem desse equipamento é que o desenvolvimento de programas tem um acréscimo de complexidade, em função da necessidade de gerenciamento dos diversos espaços de memória, bem como da comunicação entre processadores;

- maciçamente paralelos: consiste de uma arquitetura com memória distribuída, entretanto, a diferença para um computador com memória distribuída é que todos os processadores recebem a mesma seqüencia de instruções. Por se tratar de uma arquitetura para fins específicos, torna-se difícil o desenvolvimento de aplicações diversas àquelas destinadas a esse tipo de equipamento. Como nosso caso não engloba tais aplicações e não se tem disponível tal tipo de equipamento esse não será doravante considerado.

Nesse projeto será utilizado nas etapas de desenvolvimento um "cluster" o qual consiste de um conjunto de computadores interligados por meio de uma rede (ETHERNET, GIGABIT, MYRINET etc), formando um "meta computador distribuído". O cluster que será utilizado no projeto consiste de 24 computadores (nós), sendo cada computador biprocessado, podendo ao longo do projeto os processadores ser atualizados para processadores do tipo *dual core*, tornando-se então cada nó quadriprocessado. Os nós estão interligados por meio de uma rede GIGABIT.

Arquitetura de código

A obtenção de uma melhor performance tem como contrapartida o aumento da complexidade de implementação, uma vez que um código paralelo necessita ter implementadas rotinas de geren-

ciamento. Para evitar que essa complexidade seja aumentada de tal modo que o gerenciamento comprometa a performance, há a necessidade de se levar em conta a granularidade: a granularidade é mais fina quanto menor o número de operações desempenhadas entre os ciclos de comunicação [9].

Uma granularidade fina facilita o "balanceamento de carga" dos processadores entretanto, requer uma maior quantidade de comunicação. Por outro lado, uma granularidade alta tem baixa comunicação tendo como contrapartida a dificuldade no gerenciamento do balanceamento de carga [9].

Arquitetura de código para memória compartilhada

A discussão sobre sistemas paralelos em computadores de memória compartilhada passa pela definição dos conceitos de processo e pelo conceito de "*thread*". Um processo inicia-se como um processo sendo executado em um *thread* único, podendo ou não gerar outros *threads* durante sua execução. O ponto de diferença diz respeito ao espaço de memória disponível durante a execução. Um processo pode acessar apenas o seu próprio espaço de memória enquanto um "*thread*" pode acessar o espaço de memória do processo que o criou e o seu próprio espaço. O problema gerado pela programação utilizando multiprocessamento é justamente que qualquer "*thread*" pode acessar a memória "*heap*", onde são armazenadas as variáveis estáticas do programa, gerando assim um problema de gerenciamento de dados, de modo a evitar que "*threads*" modifiquem dados simultaneamente.

Com a distinção desses conceitos, em função de como é feito o gerenciamento de memória, pode-se caracterizar aplicativos que se utilizam de memória compartilhada em um dos seguintes tipos[25]:

- suporte ao multi-processamento através de paralelização automática: os modernos sistemas operacionais tais como os da família Windows, distribuições Linux e UNIX são sistemas multitarefa. Um sistema multitarefa não requer a existência de mais de um processador para poder executar vários aplicativos simultaneamente. O que ocorre é que esses sistemas tem meios de gerenciar a troca de processos no sistema a cada intervalo de tempo fixo ou a cada sinal de interrupção ou operação de acesso a algum dispositivo. Em geral cada processo tem associado algum nível de prioridade de modo a tornar possível o gerenciamento do tempo de CPU entre os processos. Mesmo quando o computador tem apenas um processador é importante se ter funcionalidades de multiprocessamento. O melhor exemplo de requisitos de multiprocessamento ocorre no caso de comunicação síncrona. Caso ocorra problema de comunicação, a chamada para a conexão vai ocupar o sistema até que o tempo limite de conexão. Já em sistema multiprocessado, apenas o código de requisição permaneceria aguardando. Existem compiladores que, em se tendo disponibilidade de multiprocessamento, geram códigos *multithread*, notadamente através de técnicas de *unrolling*. Um exemplo desse tipo de

compilador é o Open MP `http://www.openmp.org/`;

- suporte ao multi-processamento por meio de programação *multithread*: no caso de computadores multiprocessados, a programação de *threads* pode gerar um aumento de performance. O exemplo mais fácil de ser visualizado é a divisão de trechos de "*loops*" pelo número de processadores disponíveis. O aumento de performance depende da estratégia utilizada pelo programador. A programação *multithreading* implica em alguns requisitos a mais durante o planejamento de implementação, de modo a evitar *threads* concorrentes ao mesmo dado, evitando assim necessidade de "*locks*" e "*signals*" e também cuidado especial na identificação dos pontos de sincronização.

Arquitetura de código para memória distribuída - "*Message Passing*"

As bibliotecas de *message passing* implementam a comunicação entre computadores por meio de troca de mensagens. São implementadas as seguintes rotinas:

- rotinas de gerenciamento de processos, início, término identificação de um processo etc;
- rotinas de comunicação *point to point*;
- rotinas de comunicação de grupos (*broadcast*).

A principal vantagem no uso dessas bibliotecas está na utilização de um nível mais alto de programação. Os níveis mais baixos de programação de troca de mensagens tem de levar em conta a forma de "empacotamento" de tipos, uma vez que a forma de interpretação para um determinado tipo pode ser diferente de um máquina para outra (sistemas operacionais e arquiteturas diferentes podem ter formas de representação de inteiros, números de ponto flutuante etc). Outro aspecto a ser destacado é o do tipo de protocolo utilizado em cada tipo de comunicação. Protocolos específicos são criados de modo que a verificação da consistência de dados transmitidos seja feita de uma maneira indireta, evitando o tráfego excessivo de mensagens de recebimento de dados.

2.2.6 MPI - *Message Passing Interface*

O MPI é um padrão de implementação de *message passing* definido de modo a tornar-se portável entre diversas plataformas [51, 30], principalmente de memória distribuída. No MPI a implementação do paralelismo é explícita, sendo o programador responsável pela identificação dos pontos de paralelismo, notadamente transmissão e recebimento de dados, bem como sincronismo.

A biblioteca MPI definida para comunicação no projeto é a MPICH2 `http://www.mcs.anl.gov/research/projects/mpich2/`, versão de implementação do padrão MPI de domínio público.

As funções de interesse na implementação do MPI para esse trabalho são:

- MPI_INIT: inicializa o ambiente, sincronizando os processos;

- MPI_COMM_RANK: retorna o identificador de cada processo. Através desse parâmetro podem ser mapeados os processadores da rede;

- MPI_COMM_SIZE: retorna o número total de processos de um determinado grupo;

- MPI_SEND: rotina para envio de mensagens do tipo *blocking send*, ou seja a rotina só retorna após o término da transmissão do dado ou algum tipo de erro ser identificado. As mensagens consistem de um envelope e de um conteúdo. O envelope contém as informações de destino, um rótulo para a mensagem e um campo para identificação do tipo de erro caso esse ocorra. O conteúdo consiste de um *buffer* com tamanho identificado e de um identificador para o tipo de dado que está sendo transmitido.

- MPI_RECV: rotina para o recebimento de mensagens, sendo do tipo *blocking receive*, ou seja a rotina só retorna após o final do recebimento do dado. Da mesma forma que para a função de envio, essa função tem basicamente dois argumentos, um conteúdo e um envelope. As informações de envelope dizem respeito ao identificador do processo que enviou a mensagem, um rótulo para a mensagem e uma variável para definição do tipo de erro, caso este ocorra. Para o recebimento do conteúdo, é informado um buffer de destino cuja dimensão é a dimensão informada na mensagem.

- MPI_FINALIZE: finaliza o processo de maneira organizada, sincronizando todos os processos para a finalização.

2.2.7 Serialização de Dados

A habilidade de armazenar um objeto sob a forma de um vetor de bytes que pode ser transmitido ou armazenado sob alguma forma e posteriormente reconstituído sob a forma do objeto novamente é um aspecto essencial em diversas aplicações [41]. Ao processo de converter um objeto em um vetor de bytes é denominado serialização, enquanto a reconstrução de um objeto a partir de um vetor de dados é denominado deserialização [41].

Os tipos de dados que podem ser enviados por meio de uma mensagem MPI são os tipos primitivos (*int, long, double, char*, etc) que tem definida sua forma de serialização. Assim, qualquer objeto definido pelo usuário, para ser transmitido por meio do MPI precisa ser convertido em um vetor de *bytes* de objetos primitivos e identificados de maneira única, de tal modo que ao ser recebido por outro processador, com base no identificador do dado seja possível restaurar o objeto inicial.

A linguagem Java, Java `http://www.sun.com/java/`, implementa de modo nativo classes para serialização e deserialização de objetos primitivos (inteiros, reais, caracteres etc) e é tal implementação que foi tomada por base para a implementação utilizada nesse trabalho.

A deserialização representa um problema, pois no momento de restaurar um objeto a partir do vetor de *bytes* não se tem informação de que tipo de objeto está ali serializando. Assim há a necessidade de se gerar algo como um protocolo de comunicação, onde as primeiras informações do vetor de *bytes* são utilizadas justamente para identificar o objeto que ali está representado. Adicionalmente, ao final do vetor pode-se colocar alguma informação extra, tal como o tamanho dos dados ou novamente o tipo do objeto, de modo a identificar possíveis inconsistências de dados.

Em termos de orientação a objetos, a implementação do protocolo de serialização é feito utilizando uma classe base que implementa dentre outros aspectos o "cabeçalho de identificação do objeto". Desse modo, as classes derivadas precisam implementar apenas a serialização de seus dados, sem se preocupar com o cabeçalho da mensagem (protocolo).

Adicionalmente, há a necessidade de se informar ao método responsável pelo recebimento de mensagens sobre a relação entre identificadores passados nos *streams* e classes. Tal relação deve ser única. No caso da implementação aqui proposta tal informação é feito por meio da criação de um objeto estático que no seu construtor faz o registro de duas informações: identificador e nome da classe.

2.2.8 Performance

A performance de um código pode ser medida com base na relação entre o número de operações de ponto flutuante por segundo que é obtido pelo código em relação ao máximo teórico que o computador pode realizar. Assim, um código tem a performance melhor do que outro se este requisitar menos tempo para obter uma resposta com precisão adequada nas mesma condições de equipamento e carga de processador.

Existem diversos modos de medir o desempenho de um código. O modo mais direto é medir o tempo de resposta do código pelo tempo de relógio ("*wall clock time*"). Este tempo é subjetivo, porque ele depende da disponibilidade do processador naquele momento. Outra possibilidade é medir o tempo de CPU, ou seja o número de ciclos de cálculo dedicado ao processo. Este dado também é subjetivo, pois não é computado o tempo alocado para "paginações" (transferência de dados da memória para o *cache* do processador). A relação entre o tempo de relógio e o tempo de CPU depende dos padrões de acesso a memória [25].

Nesse trabalho, a medida de tempo será feita por meio de *wall clock time*. Como os testes serão realizados em um cluster com controle de acesso por meio de sistema de filas, espera-se que os resultados sejam representativos.

Quando se trata de computação paralela, o conceito de performance sofre uma pequena alteração. Aqui não basta apenas saber a quantidade de operações de ponto flutuante (FLOPS) do código e sim a vantagem obtida em executar um código em vários processadores quando comparado a execução do mesmo processo em um único computador (processo serial) [32, 38].

A esse ganho de velocidade por meio da utilização de processamento paralelo denomina-se

speedup [25]. A lei de Amdahl preconiza que:

$$speedup = \frac{1}{1-f} \tag{2.27}$$

f: porcentagem paralelizável do programa

speedup: aumento da velocidade de processamento

Quando o código não é paralelizável ($f = 0$) e portanto o *speedup* =1, já para um código completamente paralelizável ($f = 1$) e $speedup = \infty$.

Inserindo o número de processadores na lei de Amdahl podemos ter o *speedup* teórico para uma aplicação com grau de paralelização conhecido [9], ou por outro lado estimar a porcentagem de código paralelizado:

$$speedup = \frac{1}{\frac{f}{np} + s} \tag{2.28}$$

np : número de processadores

s: porcentagem de código serial

2.3 Ambiente de Programação Científica Orientado a Objetos PZ

A filosofia de Orientação a Objetos surgiu como um requisito de engenharia de software e desde então vem sendo utilizada em diversos projetos, prova disso é a tentativa de transformação de algumas linguagens de programação originalmente voltadas a programação estruturada em linguagens orientada a objetos (e.g. Delphi - object pascal; Fortran 90).

Apesar dos ganhos oferecidos em termos de gerenciamento de código, geração de códigos reutilizáveis etc. o uso da orientação a objetos continua limitada nos meios científicos. Isso pode ser explicado, em maior parte, pela disponibilidade de bibliografia e algoritmos utilizando linguagens tradicionais (principalmente Fortran).

A limitação das linguagens estruturadas levou o Prof. Devloo a desenvolver o projeto do ambiente PZ. Tal idéia surgiu pela verificação, durante a implementação de procedimentos adaptativos, de que diversos códigos de elementos finitos, abordando diversos problemas físicos, ser muito similares entre si, diferindo em termos de problema variacional, número de graus de liberdade por nó e método numérico de resolução do problema. Tal constatação mostrou a possibilidade de uma "parametrização" de uma aproximação por elementos finitos em alguns procedimentos bem definidos:

- Definição da formulação fraca;
- Discretização do domínio;

- Escolhas do espaço de funções de aproximação;
- Cálculo de matriz de rigidez (incluindo massa ou não) e vetor de cargas;
- Método de cálculo de resíduo / resolução do sistema linear;
- Pós-processamento da solução para a obtenção de variáveis de interesse.

Outras motivações para o desenvolvimento do projeto foram:

- possibilidade de disponibilizar um ambiente de programação onde o ensino do método de elementos finitos fosse facilitado pela disponibilidade de ferramentas numéricas que tornem possível ao aluno se concentrar em pontos específicos do método durante o aprendizado, tais como poder ao desenvolver a formulação fraca implementá-la em um material e poder gerar uma aproximação numérica sem ter de se preocupar em implementar uma malha, uma matriz, um método de cálculo de integral numérica etc;
- ter um ambiente de programação de elementos finitos compartilhado com a comunidade científica. Tal motivação pode ser melhor compreendida quando analisamos as publicações científicas sobre aproximação numérica. Em geral, os algoritmos são bem descritos, entretanto a verificação dos resultados obtidos sem acesso ao código fonte é praticamente impossível. A reprodução de resultados torna-se algo inviável. Por outro lado, tendo um arcabouço como o disponibilizado pelo PZ, tendo o algoritmo utilizado, bem como os métodos numéricos descritos, a reprodução do problema torna-se o problema de implementar a formulação e juntar os módulos de análise numérica selecionados.

O ambiente PZ é um código livre, desenvolvido com ferramentas GNU, disponível para a comunidade científica através de repositório CVS livre (ver Download PZ `http://labmec.fec.unicamp.br/pz/download`). O suporte pode ser obtido por meio de *e-mail* aos membros do projeto. A documentação do código está sendo organizada, sendo atualmente disponível a documentação gerada automaticamente por meio do programa Doxygen (Documentação PZ `http://labmec.fec.unicamp.br/pz/doxygen`). As publicações sobre o PZ estão sendo catalogadas e organizadas para ser disponibilizadas por meio de *download*. Uma descrição do ambiente PZ, por completo, focando os aspectos relacionados aos métodos adaptativos pode-ser encontrada em: PZ não oficial `http://labmec.fec.unicamp.br/~cesar/Publications/PZ_nonOfficial-PT_BR.pdf`.

2.3.1 Conceitos Topológicos no PZ

A implementação dos métodos adaptativos no ambiente PZ está baseada em conceitos topológicos cuja definição faz-se necessária para o entendimento de termos que serão utilizados no decorrer do trabalho.

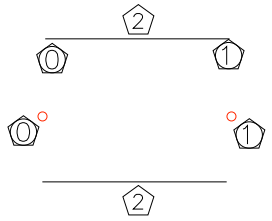


Figura 2.1: Aresta - topologia e definição de lados

Tabela 2.1: Aresta - conectividades e dimensão por lado do elemento

Lado	Nó-Id	Dim
0	{0}	nó - 0
1	{1}	nó - 0
2	{0,1}	aresta - 1

Parte-se da definição de um elemento uma partição de lados. Tal conceito é fundamental para o entedimento da definição de vizinhança, haja vista que as vizinhanças são definidas entre lados de elementos. Finalmente passa-se para a definição de transformações paramétricas, as quais podem ser de coordenadas paramétricas de um vizinho para o outro, ou ainda entre lados de um elemento e o lado correspondente de um elemento proveniente de sua divisão.

Elemento Como Partição de Lados

No ambiente PZ, um elemento consiste em uma partição de lados. Onde os lados podem ser pontos (nós de canto), linhas (arestas), faces (triângulos ou quadriláteros) e volumes (tetraedros, pirâmides, prismas e hexaedros). Cada lado é um conjunto aberto cuja união (partição) forma o elemento (conjunto fechado).

Os lados são enumerados em uma seqüência pré-estabelecida, começando pelos lados de menor dimensão (nós) até chegar ao lado de dimensão correspondente a dimensão do elemento. Para exemplificar, a Figura 2.1 apresenta a topologia e a definição de lados para uma aresta padrão, sendo mostrado na Tabela 2.1 as conectividades e dimensão por lado do elemento. Adicionalmente mostra-se na Figura 2.2 a topologia de um hexaedro com suas connectividades sendo apresentadas na Tabela 2.2. As definições de todas as topologias disponíveis no ambiente PZ estão em Bravo [7].

O conceito do par elemento-lado é central para o desenvolvimento de métodos adaptativos no ambiente PZ. A partir desse par é que são identificadas as vizinhanças entre elementos e

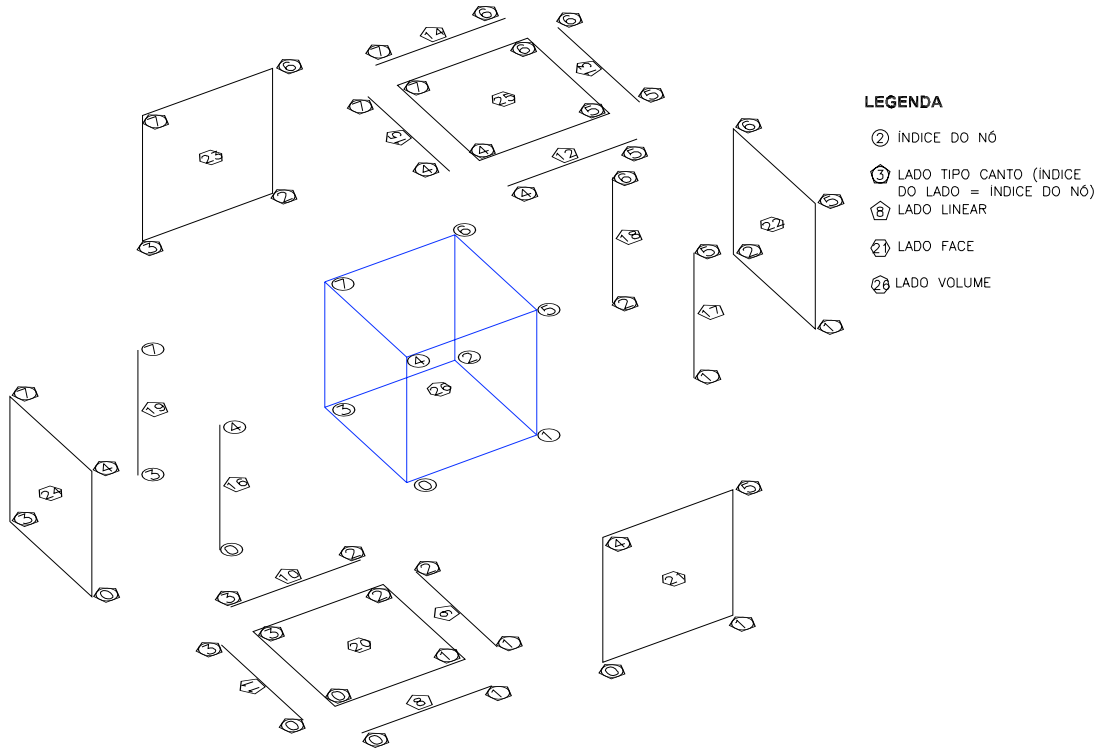


Figura 2.2: Hexahedro - topologia e definição de lados

Tabela 2.2: Hexaedro - Conectividades e Dimensão por Lado do Elemento

Lado	Nó-Id	Dim	Lado	Nó-Id	Dim	Lado	Nó-Id	Dim
0	{0}	nó	9	{1,2}	aresta	18	{2,6}	aresta
1	{1}	nó	10	{2,3}	aresta	19	{3,7}	aresta
2	{2}	nó	11	{3,0}	aresta	20	{0,1,2,3}	face
3	{3}	nó	12	{4,5}	aresta	21	{0,1,5,4}	face
4	{4}	nó	13	{5,6}	aresta	22	{1,2,6,5}	face
5	{5}	nó	14	{6,7}	aresta	23	{2,3,7,6}	face
6	{6}	nó	15	{7,4}	aresta	24	{3,0,4,7}	face
7	{7}	nó	16	{0,4}	aresta	25	{4,5,6,7}	face
8	{0,1}	nó	17	{1,5}	aresta	26	{0,1,2,3,4,5,6,7}	volume

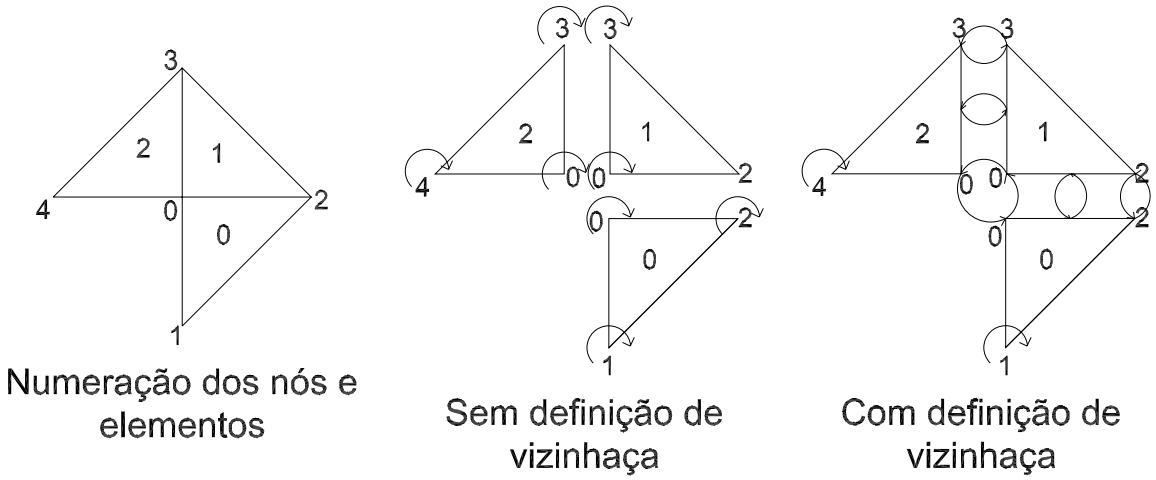


Figura 2.3: Exemplos de vizinhaça entre elementos/lado

também, através da vizinhança interna elementos refinados e seus elementos pai que se pode definir transformações entre esses.

Vizinhança

A definição de vizinhança em formulações contínuas é essencial para um método adaptativo, uma vez que as funções entre elementos vizinhos devem coincidir nos lados compartilhados pelos elementos.

A definição de vizinhança no ambiente PZ é feita a partir da seguinte análise, um elemento é vizinho por um determinado lado de outro elemento por determinado lado se o conjunto de pontos dos dois lados forem coincidentes. O caso mais simples de se vizualizar é a vizinhança por nós, caso dois elementos compartilhem um nó eles serão vizinhos pelos seus lados correspondentes a esse nó. A Figura 2.3 exemplica esse conceito.

A análise e definição das vizinhanças no ambiente PZ estão implementadas em um método denominado *BuildConnecitivity* da classe que define uma malha geométrica. Esse método será muito citado no decorrer do trabalho.

Vizinhaça Cíclica

O aspecto aqui ressaltado é a forma de armazenamento de vizinhanças. Tomando o caso da vizinhança por nós como exemplo temos que é comum que mais de dois elementos compartilhem um determinado nó.

Caso cada elemento/lado armazene todos os elementos que estão a ele ligados, teria-se uma grande quantidade de dados a armazenar e que seria de difícil gerenciamento, pois a cada refina-

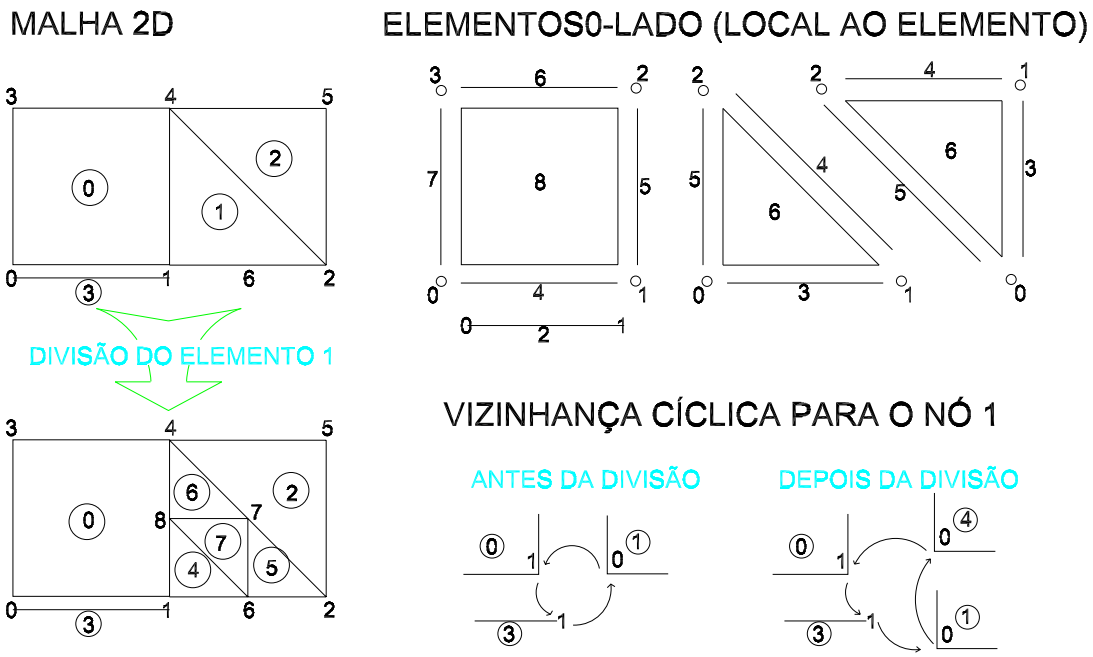


Figura 2.4: Processo de ajuste de vizinhança de um elemento durante o refinamento

mento do elemento os elementos filhos deveriam atualizar as informações de todos os elementos que tinham seu elemento ancestral como vizinho.

Tal aspecto é contornado na implementação do ambiente PZ por meio do conceito de vizinhaça cíclica, onde se garante que, em se tendo um conjunto de elementos que compartilham um lado, todos os elementos/lado desse conjunto apontaram para um vizinho distinto daqueles definidos por outros elementos e distinto de si próprio. Caso não haja vizinhos, ou seja apenas um elemento "compartilha" um lado, esse elemento/lado indica a si próprio como vizinho.

A idéia principal é a de poder percorrer toda a vizinhança sem ter de com isso replicar a informação armazenada. Outro aspecto importante é que, em caso de refinamento do elemento, o acerto da vizinhança é algo simples, bastando que o elemento pai aponte para um lado do elemento filho e esse aponte para o elemento/lado para o qual o elemento/lado do pai apontava. A Figura 2.4 ilustra esses conceitos.

Transformações Paramétricas Entre Elementos/Lado

As transformações paramétricas entre elementos/lado são um fator muito importante em um processo adaptativo onde se utiliza espaços de interpolação contínuos. Nesses casos há a necessidade de se garantir que as funções de forma dos elementos sejam contínuas nos lados compartilhados por dois elementos. No caso de adaptação do espaço de aproximação para um elemento, deve-se garantir, por meio de uma função de restrição, que o espaço de aproximação continue continuo. Para

o cálculo das funções de restrição é essencial o uso de transformação de coordenadas paramétricas.

Há dois tipos básicos de transformações que são necessárias: cálculo entre coordenadas paramétricas entre elementos/lado vizinhos e cálculo de transformação entre elementos/lados inclusos.

O primeiro caso consiste simplesmente em encontrar a rotação que leva de um sistema de coordenadas ao outro, uma vez que ambos compartilham o mesmo conjunto de pontos.

Já o segundo caso se aplica ao caso de transformação de coordenadas de um lado de um elemento filho para um lado de um elemento pai. Nesse caso, a tranformação será do tipo:

$$\widehat{\xi} = \widehat{\xi_0} + J(\xi) \tag{2.29}$$

ou seja, a coordenada paramétrica do elemento filho acrescido do jacobiano da transformação do sistema do filho para o do pai. Caso o jacobiano seja constante a transformação também o será, caso contrário, pode-se calcular o jacobiano no ponto de interesse ou utilizar o jacobiano de um ponto dado, tal como o centro do elemento, e utilizar a transformação considerando que há um erro de aproximação.

Refinamento de Elementos - Inclusão Única

O processo de divisão de um elemento, denominado refinamento h é um processo comum em códigos adaptativos, consistindo em particionar o elemento original em um conjunto de de elementos cuja união, ou partição, constitui o elemento original.

A forma como o elemento original é dividido, incluindo as relações de vizinhança entre o elemento original e o elemento filho, relações de inclusão de lados de elementos filhos em lados do elemento pai e as transformações paramétricas entre esses elementos/lado denominamos padrão de refinamento.

Um aspecto topológico a ser ressaltado é o fato de que cada lado de um elemento filho está incluso em um único lado do elemento pai. Tal fato torna possível o cálculo das trasnformações paramétricas, conforme descrito acima.

Nesse trabalho, utilizaremos a denominação de refinamento "uniforme" como sendo um refinamento h cujos ângulos internos dos elementos filhos são no mínimo iguais ao do elemento pai, ou seja, não ocorre degeneração do elemento no processo de refinamento. Adicionalmente, todos os aspectos de um padrão de refinamento são definidos por meio de código (*hard coded*).

2.4 Ambiente de Programação Paralela Orientado a Objetos - OOPar

O OOPar é um ambiente de programação científica paralela orientado a objetos que vem sendo desenvolvido pelo grupo de pesquisa do Prof. Devloo há quase uma década ([10, 11]). A principal

característica do OOPar é o provimento de uma interface orientada a objetos para programação paralela. Essa interface é única, podendo o programa ser executado em um ambiente de memória distribuída ou compartilhada.

No ambiente OOPar, um programa paralelo é estruturado como uma seqüência de tarefas que atuam sobre dados. O encapsulamento de bibliotecas de *message passing* é outra característica interessante do OOPar propiciando uma interface de mais alto nível para a implementação e o gerenciamento da distribuição de dados e de tarefas. Dentre as motivações principais para o desenvolvimento do OOPar, podem ser destacadas:

- encapsulamento das funções de *message passing* em um ambiente de alto nível orientado à computação científica, com enfoque na resolução de problemas de conservação por meio aproximações numéricas (FEM, BEM, FV, FD etc);
- propiciar um ambiente de programação paralelo utilizando orientação a objetos;
- propiciar um ambiente de programação paralelo capaz de utilizar toda a capacidade de processamento de *clusters* onde cada nó pode ou não ser multiprocessado (ambiente denominado por alguns autores como "misto").

A principal motivação no uso do OOPar é que, a partir de um código serial, a implementação de um código paralelo com o OOPar é muito simples, uma vez que uma tarefa pode consistir na execução de um método já implementado no código serial, sendo o objeto sobre o qual será executada a função um atributo da tarefa ou uma dependência dessa. Resta como trabalho a desenvolver o processo de serialização dos dados que podem vir a ser transmitidos, bem como a criação e controle dos dados distribuídos.

Um estudo mais aprofundado do OOPar pode ser encontrado em OOPar não oficial `http://labmec.fec.unicamp.br/~cesar/Publications/OOPar-nonOfficial-PT_BR.pdf`, onde são descritos os principais componentes do OOPar, bem como a forma como são gerenciados acesso a dados e execução de tarefas.

Capítulo 3

Geração de Padrões de Refinamento A Partir de Malhas Exemplo

Diversos problemas de engenharia apresentam um comportamento para o qual a divisão adequada do elemento, tal qual o seu alinhamento com um choque por exemplo, pode ser muito mais eficaz na redução dos erros de aproximação quando comparado a uma divisão uniforme do elemento original.

Outro exemplo interessante é o de refinamento de elementos para capturar uma camada limite em um problema de escoamento de fluidos. A divisão do elemento do elemento por igual ao longo de um único eixo irá proporcionar resultados similares ao do refinamento uniforme , mas com uma grande vantagem: o número de elementos gerados, e por conseqüência de equações, é muito menor. Isso pode ser visto no exemplo mostrado na Figura 3.1, onde se quer dividir apenas os elementos adjacentes às condições de contorno indicadas.

Para reduzir a camada limite do problema utilizando padrões de refinamento uniformes a 1/4 da dimensão original, requer-se uma malha com 133 elementos, conforme mostrado na Figura 3.2 (a) enquanto a mesma redução é possível utilizando-se uma malha com 38 elementos no caso de utilizarmos um padrão de divisão que divide o elemento original a partir das arestas que estão conectadas ao contorno de interesse conforme mostrado na Figura 3.2 (b).

Conforme descrito na introdução desse trabalho, a escolha do padrão de refinamento é um dos desafios da auto-adaptatividade conforme descrito pelo Prof. Zienkiewicz [55]. A Figura 3.3 exemplifica essa idéia mostrando diversas formas de se dividir um elemento quadrilateral.

A abordagem descrita na seqüência é um passo na direção da flexibilização do refinamento h tradicional. A contrapartida a flexibilidade de escolha da forma com que se irá dividir o elemento é o aumento na complexidade do código necessário para o gerenciamento dos padrões disponíveis, bem como a necessidade de verificação de compatibilidade entre padrões de refinamento para lados comuns de elementos vizinhos.

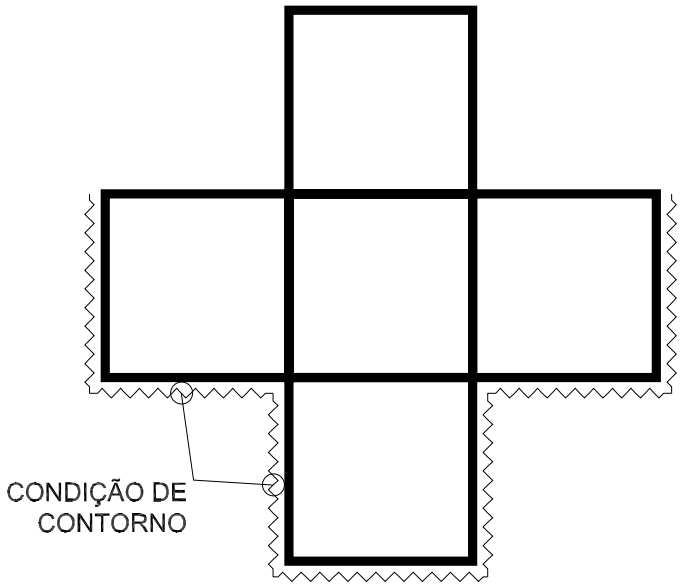


Figura 3.1: Exemplo de utilização de padrões de refinamento uniforme e direcional

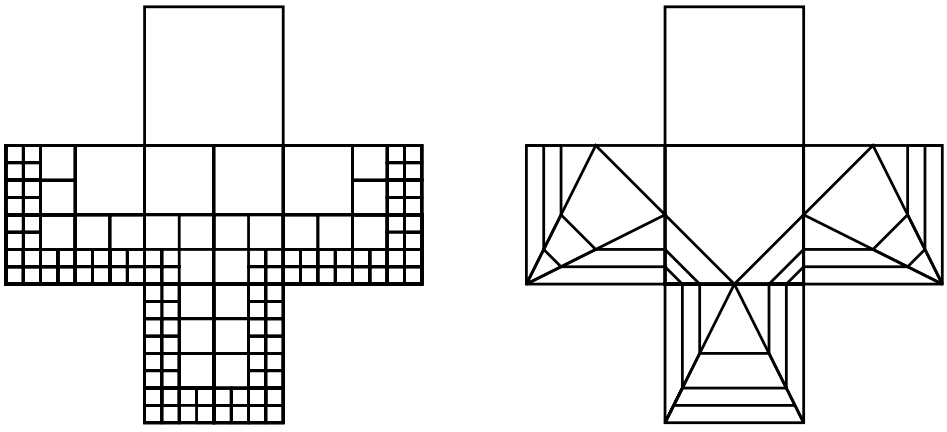


Figura 3.2: Resultados de malhas de camada limite utilizando padrões de refinamento uniforme (a) e padrões de refinamento direcional (b)

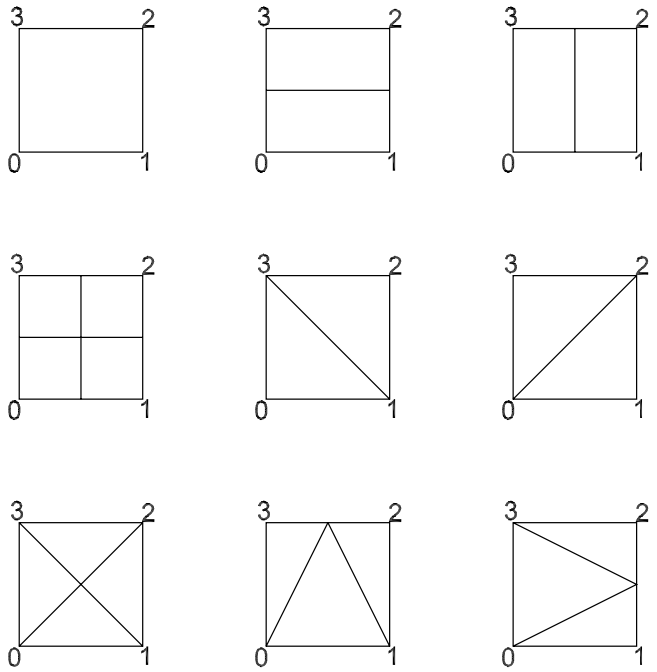


Figura 3.3: Exemplos de divisão de um elemento quadrilateral

3.1 Conceitos Básicos

Retornando à seção que descreveu o conceito de refinamento no ambiente PZ, temos que um padrão de refinamento consiste na definição de uma partição de um elemento, incluindo as informações de relacionamento topológico entre elemento pai e filho.

Assim, o primeiro passo necessário ao "cálculo" de um padrão de refinamento é a definição do relacionamento de vizinhança entre os elementos filhos e o elemento pai.

Tendo-se a partição de elementos lado dos elementos filho que compõe cada elemento lado do pai, a próxima informação necessária é a transformação paramétrica entre pontos do elemento lado filho para o respectivo elemento lado do pai.

A idéia aqui apresentada consiste em como obter as informações acima a partir de uma malha que contenha um elemento representando o elemento original e a sua partição de filhos.

3.1.1 Relacionamento Topológico - Definição da Vizinhança

Como descrito anteriormente, um elemento é vizinho de outro caso esses possuam um elemento lado em comum. No caso de uma malha geométrica qualquer, o processo de identificação de elementos vizinhos está implementado (método *BuildConnectivity* da classe *TPZGeoMesh*).

Assim, tendo-se uma malha contendo um elemento original e sua partição de subelementos, basta executar o método *BuidConnectivity* sobre a sua malha geométrica para se ter como resultado a definição da vizinhança.

Com a vizinhança montada é possível identificar a partição de elementos lado de filhos que

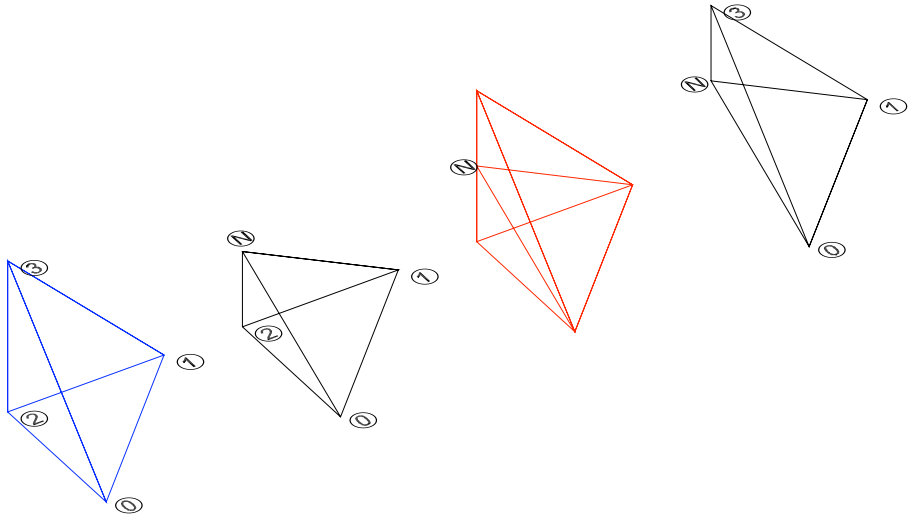


Figura 3.4: Exemplo de Divisão

compões cada elemento lado do elemento pai, ou ainda, em qual lado pai está incluso cada lado do filho, podendo-se assim obter as seguintes informações:

- Número total de elementos filhos: durante o processo de divisão de um elemento, a primeira informação necessário é o número de subelementos, de modo a tornar possível a alocação de memória para os elementos filhos;
- Número de elementos/lado filhos por elemento/lado pai: tal informação contém implicitamente a verificação da necessidade ou não de criação de nós em um determinado lado do elemento. Outra utilidade dessa informação será descrita posteriormente na abordagem de padrões de refinamento compatíveis. No exemplo mostrado na Figura 3.4 há a necessidade de criação de novos nós apenas na aresta entre os nós $\{2\,e\,3\}$;
- Vizinhança entre os elementos filhos: após a criação do elemento faz-se necessário o ajuste das vizinhanças dos novos elementos, de modo a manter a estrutura de vizinhança cíclica da da malha consistente;
- Vizinhança entre o elemento pai e os elementos filhos: a manutenção da consistência das vizinhanças requer que o elemento pai tenha como vizinho um elemento filho e que este elemento filho herde o vizinho de seu pai.

Transformações entre elementos lado pai e elementos lado filhos

Conforme mencionado anteriormente na seção relativa ao ambiente PZ, a transformação paramétrica entre pontos do filho para pontos do pai é necessária para o cálculo de funções restrição e também para cálculos de transferência de solução.

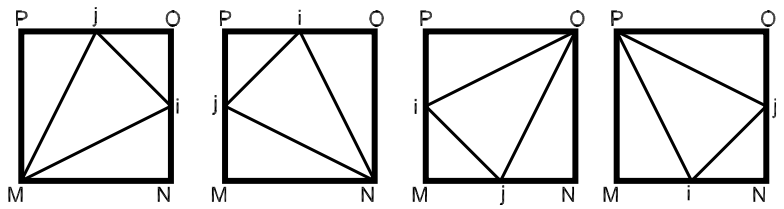


Figura 3.5: Exemplo de permutações de um padrão de refinamento

Com base na informação de em qual lado do elemento pai está incluso o lado do elemento filho, podemos utilizar a expressão apresentada na seção "Transformações Paramétricas Entre Elementos/Lado" na Equação 2.29 para aproximar a transformação. Como já descrito, se o jacobiano da transformação for constante, a transformação calculada será exata.

Padrões de Refinamento Iguais / Permutação de Um Padrão de Refinamento

A definição de determinadas características para um refinamento, tal qual o refinamento de camada limite por exemplo, requer que uma grande quantidade de padrões sejam gerados.

Outro aspecto é que uma vez determinado um padrão, existe uma série de padrões “similares”, onde o dado que muda é o lado que é dividido. Cada diferente lado dividido gera um padrão diferente. A Figura 3.5 mostra um exemplo desse tipo para um quadrilátero, onde a partição de elementos é a mesma mas os padrões de refinamento são "distintos".

Um conceito que se faz necessário é o conceito de igualdade de padrões de refinamento. Em uma primeira abordagem definimos dois padrões iguais como sendo aqueles que respeitam as seguintes condições:

1. O número de elementos/lado de filhos para cada elemento/lado do pai é igual;

2. A tranformação paramétrica de uma mesma coordenada de elementos/lados filhos correspondentes, leva a mesma coordenada paramétrica do elemento pai.

Tendo a definição de igualdade, podemos utilizá-la para definir um padrão permutado como sendo um padrão que, por meio de permutações válidas dos nós do elemento pai, pode-se chegar a igualdade com um padrão dado.

Nesse trabalho foi feito o caminho inverso, ou seja foi criada uma metodologia para gerar todas as permutações de um dado padrão de maneira automática. Destaca-se que o custo de gerar uma permutação de um padrão de refinamento é menor que o custo de calcular essa mesma permutação com base em um arquivo.

Padrões de Refinamento Compatíveis

O problema que se tem em mente aqui é novamente a criação de espaços contínuos de aproximação de elementos finitos. Para que esses espaços sejam criados, há a necessidade de que em caso de adaptação, seja possível calcular uma função de restrição entre o espaço de aproximação do elemento adaptado e o espaço original, no elemento vizinho. Tal cálculo é possível caso os refinamentos entre dois elementos/lados vizinhos sejam conformes.

O dado que se coloca aqui é que cada padrão de refinamento sabe definir os padrões de refinamento para cada um de seus lados baseado no seu próprio padrão.

A compatibilidade entre dois padrões de refinamentos por determinados lados ocorre quando os padrões de refinamento desses lados são iguais. Assim, diz-se que esses padrões são conformes. A Figura 3.6 mostra exemplos de padrões conformes e não conformes.

3.2 Implementação - Classe TPZRefPattern

A classe *TPZRefPattern* define a estrutura de dados para um padrão de refinamento. A interface da classe toma por base uma "malha exemplo" onde o primeiro elemento da malha representa o elemento pai ou elemento não dividido e o conjunto dos demais elementos representa a partição de filhos. O objetivo é definir e utilizar um padrão de refinamento h a partir de um arquivo de dados com informações necessárias a respeito da divisão do elemento, como o número de filhos, a localização dos novos nós no elemento padrão que correspondem a divisão, a seqüência de índices nodais dos sub-elementos, etc.

A simplicidade dos requisitos de definição de um padrão de refinamento são exemplificados abaixo a partir da descrição de um arquivo de entrada de dados. Na seqüência é mostrado qual o processamento que é feito com esses dados para gerar um padrão de refinamento.

Arquivo de Dados de Entrada

A idéia aqui é a de definir um formato de arquivo de dados que deve possibilitar a geração de uma malha geométrica com base em suas informações. A simplicidade desse arquivo pode ser comprovada a partir da descrição de sua estrutura, feita na seqüência.

1. Primeira linha:
   (a) número de nós,
   (b) número de elementos,
2. Segunda linha: Texto definindo o padrão / nome do padrão de refinamento;
3. Próximas n linhas (número de nós = n): dados dos nós onde cada linha contém as coordenadas de cada nó:

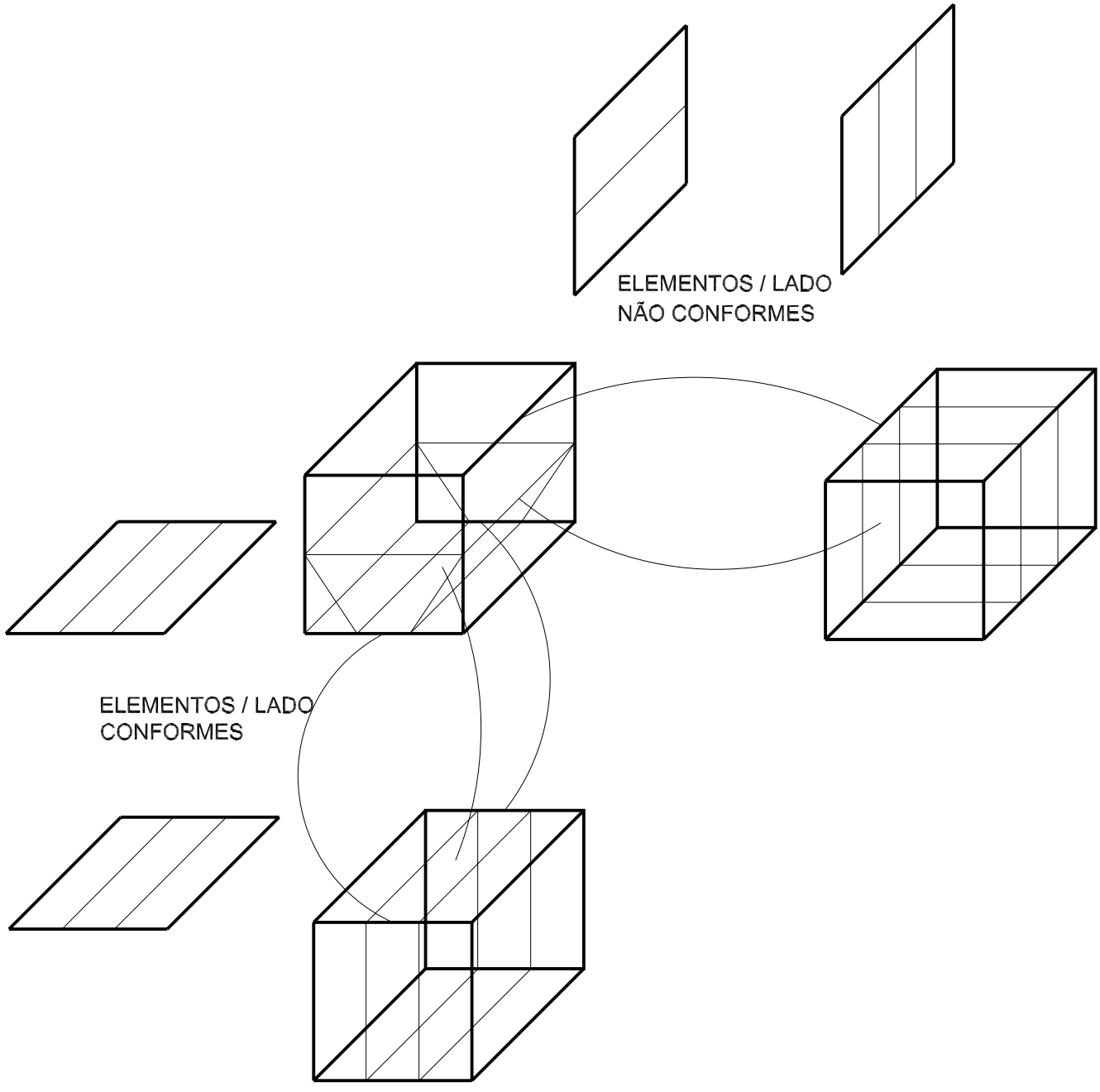


Figura 3.6: Exemplos de padrões de refinamento conformes e não conformes

(a) (x_0, y_0, z_0),

(b) (x_1, y_1, z_1),

(c) ...

(d) $(x_{n-1}, y_{n-1}, z_{n-1})$.

4. Próximas m linhas (número de elementos = m): dados dos elementos onde cada linha contém uma informação que representa o tipo de elemento (0 = ponto; 1=linha; 2=triângulo; 3=quadrilátero; 4=tetraedro; 5=pirâmide; 6=prisma; 7=hexaedro), a próxima informação é o índice do material associado a cada elemento, e as conectividades do elemento:

   (a) $element{-}Type_0 \quad material{-}Id_0 \quad Connectividade_0^0 \; Connectividade_0^1 \; ... \; Conectividade_0^k$

   (b) $element{-}Type_1 \quad material{-}Id_1 \quad Connectividade_1^0 \; Connectividade_1^1 \; ... \; Conectividade_1^k$

   (c) ...

   (d) $elem.{-}Type_{m-1} \, material{-}Id_{m-1} \, Connectividade_{m-1}^0 \, Connectividade_{m-1}^1 \, ... \, Conectividade_{m-1}^k$

3.2.1 Implementação dos métodos relacionados à divisão

A classe ***TPZRefPattern*** é responsável pela manutenção das informações necessárias para a divisão de um elemento na forma como indicada na malha exemplo. O requisito dessa classe é saber "calcular" as informações necessárias à divisão de um elemento com base nas informações dessa malha exemplo.

Em termos de atributos, a classe tem duas estruturas básicas:

- relação topológica entre elementos lado do pai e elementos lado dos filhos: tal informação é armazenada em uma estrutura esparsa, onde o vetor de localização armazena o número de elementos lado filhos por lado do elemento pai (ver Figura 3.7);
- transformações topológicas entre elementos lado filhos e elemento lado pai: tal informação é armazenada em uma estrutura esparsa, similar a anterior, conforme mostrado na Figura 3.8;

Além dessas estruturas que armazenam o resultado dos cálculos, a classe ainda tem como atributos:

- malha geométrica: baseada na malha exemplo do arquivo. Daqui é possível extrair as seguintes informações:
  - número de subelementos;
  - localização / posicionamento dos nós;
- nome / designação: cada padrão de refinamento pode ter definido um nome de modo a auxiliar na identificação de um padrão de refinamento durante a depuração;

- permutações (dado estático para cada tipo de elemento): permutações de nós do elemento pai geram, mantendo os nós dos elementos filhos nas mesmas posições, padrões de refinamento diferentes dos padrões originais. Essa implementação facilita o acréscimo de novos padrões na biblioteca, uma vez que identificado um padrão, todas suas possíveis permutações podem ser geradas e inseridas na biblioteca;

- padrões de refinamento dos lados: para cada lado do elemento pai pode-se definir uma partição de lados de elementos filhos. A metodologia para tal definição é a mesma utilizada para geração de um padrão de refinamento. Com tal informação torna-se possível averiguar a compatibilidade entre padrões de refinamento de elementos vizinhos.

De modo a ilustrar quais são as informações necessárias de um objeto padrão de refinamento, descreve-se na seqüência as operações executadas pelo método *Divide* de um elemento geométrico:

1. Criação dos objetos nós dos filhos dentro do elemento pai (contendo as coordenadas do nó), caso estes ainda não tenham sido criados pelo elemento vizinho: A atribuição do padrão de refinamento é definir a posição onde devem ser criados os novos nós para a definição dos elementos filhos. A verificação da existência ou não de um nó naquela posição fica a cargo do método de divisão implementado no elemento geométrico;

2. Determinação da seqüencia de nós que definem os sub-elementos: Ao se criar os novos nós ou identificar os nós existentes[1], é preenchida uma lista contendo o mapeamento entre os nós do padrão de refinamento (numerados de zero até o número de nós do padrão de refinamento menos 1) para os índices de nós da malha geométrica. Tendo essa lista, define-se as conectividades dos elementos filhos como sendo os valores correspondentes à numeração local da malha do padrão de refinamento;

3. Criação dos novos sub-elementos geométricos baseado no conhecimento da seqüência de nós: Tendo a lista de subelementos, seu tipo (linear, triângulo etc) e suas conectividades, basta criar novos elementos geométricos na malha geométrica;

4. Sub-elementos devem conhecer quem é o elemento pai e vice-versa: após a criação dos subelementos, deve se fazer com que esses apontem para o seu elemento pai e que a lista de filhos do elemento pai seja preenchida com os elementos gerados. A tarefa do padrão de refinamento aqui é definir quantos subelementos tem o elemento pai;

5. Definição da vizinhança interna (entre sub-elementos): O padrão de refinamento deve identificar a vizinhança interna dos subelementos. Isso é feito copiando as vizinhanças da malha do padrão de refinamento. Essa identificação é feita por meio do método *BuildConnectivity* da malha geométrica;

[1]Ressalta-se que a verificação da existência de um nó é feito através de uma análise de proximidade.

Estrutura ↓	Lados do elemento →	lado l0			..	lado ln		
Início no vetor fPartitionEl	fInitSide	Init0			..	Initn		
Partição por lados	fPartitionElSide	$sub_i/side_i$	..	$sub_k/side_k$	..	$sub_n/side_n$	...	$sub_z/side_z$

Figura 3.7: Estrutura de dados formando a partição do elemento por sub-elementos e lados

6. Os elementos filhos devem apontar para o pai pelos cantos: O passo final é a inserção dos subelementos na estrutura de vizinhança da malha. Isso é feito com base na vizinhança entre elemento pai e subelementos. A cada vizinhança de pai para filho é feita a seguinte operação: o elemento lado filho aponta para o vizinho atual do elemento lado pai e o elemento lado pai passa a apontar o elemento lado do subelemento. Desse modo, a consistência da vizinhança é mantida.

7. Procura-se para cada elemento lado do pai um vizinho dividido, achado este deve-se atualizar a vizinhança entre os subelementos de ambos os elementos, isto é feito para: faces, arestas e para o nó de centro de cada lado aresta e face.Estes passos fecham a divisão do elemento.

Em termos de implementação do método *Divide*, temos para a classe *TPZRefPattern*:

O passo 1 precisa do conhecimento das coordenadas do nó a ser criado, dado mantido na malha geométrica da divisão, *fMesh*. A função *NewInterSideNode(..)* se encarrega desta tarefa, ela utiliza a transformação *X(..)* para a localização do nó real.

No passo 2 a informação é extraída da malha geométrica do padrão de refinamento *fMesh* e corresponde a ordem em que são lidos os nós de cada sub-elemento.

No passo 3 a função *CreateGeoEl(..)* cria os elementos baseada na malha geométrica do MEF, no tipo de elemento e na seqüência local de nós que definem o sub-elemento.

O passo 5 precisa do conhecimento da vizinhança entre sub-elementos, estes dados podem ser calculados com o método *BuildConnectivity()*. Assim pela estrutura de dados cada elemento guardado na malha da partição estará conectado a sua vizinhança.

No passo 7, deve-se conhecer a correspondência entre os subelementos do vizinho e os subelementos do elemento que está sendo dividido. Isto depende do conhecimento dos cantos associados ao lado atual assim como o lado do elemento vizinho, além de quais são os sub-elementos associados a esses lados. Tal tarefa é resolvida com o método *BuildConnectivity()* e com ajuda dos métodos *GetSubElement(..)* e *WhichSide(..)*. Para tal convenciona-se uma enumeração local para todos os lados dos elementos e sub-elementos e define-se por meio de uma estrutura de dados da forma mostrada na Figura 3.7. A variável *fInitSide* provê a posição de início no vetor *fPartitionElSide* de todos os subelemento/lado que formam a partição do lado i, a posição final é dada pelo valor seguinte de *fInitSide* ou o comprimento total de *fPartitionElSide* no caso do lado n . Desta forma para cada lado do elemento pai têm-se a partição por lados contendo os sub-elementos associados.

3.2.2 Implementação dos métodos relacionados ao cálculo das transformações

A implementação de métodos *hp* adaptativos no ambiente PZ está baseada na restrição de espaços de funções de elementos adaptados de modo a manter o espaço de aproximação sobre o domínio discretizado contínuo. As restrições podem ocorrer quando elementos vizinhos tenham tamanhos diferentes ou quando suas ordens de aproximação são diferentes. Isto implica na necessidade do conhecimento das transformações entre os diferentes lados de elementos para os lados correspondentes de seu elemento pai. Os métodos que o padrão de refinamento implementa para o cálculo dessas transformações são:

1. Uma interface que permita implementar a função *GetSubElement(..)* que procura os sub-elementos associados a determinados lados do elemento pai. Este método utiliza o conhecimento da vizinhança associada a cada lado do elemento. A implementação dessa operação é feita no método *SideSubElement(int side,TPZVec<int> &subelsides)*, onde dado um lado do elemento pai se retorna o vetor de elementos lado filhos associados. O método *Father(int side, int sub)*, faz a operação inversa, isto é, dado um subelemento e seu lado retorna a qual elemento lado do pai esse é associado e qual é o índice do subelemento no elemento pai;

2. O ponto anterior precisa da definição da função *SideSubElement(..)*, que relaciona, localmente, lados e sub-elementos. Isto pode ser retornado cada vez que for necessário. Esta função precisa do conhecimento dos filhos associados a cada lado do elemento pai. O conjunto dos subelmentos/lado é calculado uma única vez e preservada na estrutura apresentada na Figura 3.7.

3. Após a divisão do elemento deve-se procurar pela vizinhança dos sub-elementos a existência de sub-elementos menores ainda os quais devam apresentar dependência para os elementos obtidos atualmente. Esta operação pressupõe o conhecimento dos lados de dimensão menor associados com cada lado do elemento, isto exige uma função que retorne estes lados, *SideSubElement(..)*. De novo esta a informação é extraída dos vetores da Figura 3.7.

4. Para efetuar o cálculo das restrições é preciso calcular as transformações entre o lado do subelemento e o lado do elemento pai que o contém. Cabe ressaltar que caso o Jacobiano do subelemento não seja constante a transformação entre o subelemento e o elemento pai não será linear. Caso a transformação não seja linear o valor aqui calculado será uma aproximação para a transformação entre filho e pai. Com essas informações pode se definir um objeto *Transform(..)*. Assim obtém-se o cálculo da transformação entre o lado de um elemento e o lado do elemento contíguo que o contém por meio de um acúmulo de transformações (*i.e.* transformação do lado do subelemento para o lado do pai até que seja obtido o mesmo nível do vizinho e então a transformação do lado do elemento pai para o lado do elemento vizinho).

Relação ↓	Sub-elementos →	sub 0			..	sub n		
Vetor início em fTransform	fInitTransf	Inito			..	Initm		
Transformação sub/side para pai	fTransformSides	Tr_{00}	..	Tr_{0k}	..	Tr_{n0}	..	Tr_{ns}

Figura 3.8: Estrutura de dados formando a partição do elemento por sub-elementos e lados

A Figura 3.8 mostra o padrão de armazenamento das transformações entre elemento lado pai e lados dos subelementos. Para cada subelemento *sub* o vetor *fInitTransf* indica o começo e fim das transformações dos lados do filho *sub*. O vetor de transformações *fTransform* guarda todas as possíveis transformações (objetos) entre os lados dos subelementos e os lados do elemento pai.

3.3 Gerenciamento da biblioteca de Padrões

3.3.1 Armazenamento

Quando um elemento é criado de modo que sua divisão seja informada por meio de um padrão de refinamento (*TPZGelElRefPattern<TShape,TGeo>*), é necessário que o objeto padrão de refinamento (*TPZRefPattern*) esteja definido em algum lugar. Um lugar possível seria no próprio elemento, entretanto, como mais de um elemento pode ter o mesmo padrão de refinamento, tal informação deve estar disponível em um nível em que todos os elementos geométricos tenham acesso.

Desse modo, a responsabilidade pela manutenção da biblioteca de padrões de refinamento recai sobre a malha, tendo cada elemento uma referência para o objeto lá definido. Desse modo vários elementos podem apontar para um mesmo padrão, reduzindo assim a quantidade de memória necessária para tal armazenamento.

Os padrões são armazenados sob a forma de um mapa onde para cada tipo de elemento é mapeada uma lista de objetos do tipo padrão de refinamento para aquele tipo de elemento. Com essa implementação a localização e a inserção da lista de padrões disponíveis para um determinado tipo de elemento é agilizada.

3.3.2 Importação e exportação de padrões de refinamento

A leitura de um padrão através de uma malha, conforme demonstrado no início desse capítulo é muito simples e de fácil implementação e depuração. Entretanto, tal abordagem, quando se tem uma biblioteca com dezenas de padrões não é recomendada, tendo sido assim apresentada de modo a facilitar o entendimento da definição de um padrão de refinamento.

Atualmente, os padrões de refinamento são importados e exportados sob a forma de uma biblioteca, ou seja, diversos padrões de refinamento são colocados em um único arquivo.

A medida que o arquivo é lido, cada padrão é definido e inserido na biblioteca da malha

geométrica. Do mesmo modo, existe a possibilidade de se gerar um arquivo contendo todos os padrões de refinamento existentes na biblioteca da malha geométrica.

3.4 Exemplo de Aplicação: Geração de malhas de camada limite

Para um grande conjunto de problemas de mecânica dos fluidos computacional, a qualidade da malha em determinadas regiões é essencial. A definição de malhas de camada limite suaves, ou seja, malhas onde a dimensão do elemento em uma determinada direção aumenta gradualmente é um problema que consome muito esforço.

3.4.1 Definição do padrão de refinamento para cada elemento

No caso do problema especificamente analisado, o processo de definição da forma como um elemento deve ser dividido passa primeiramente pela identificação das arestas que precisam ser divididas.

O processo de definição da forma de divisão de um elemento é ilustrada na Figura 3.9, consistindo das seguintes etapas:

1. Identificação das condição de contorno onde a camada limite é importante;
2. Identificar os elementos adjacentes às condições de contorno selecionadas;
3. Identificar nos elementos selecionados acima os nós que estão na condição de contorno;
4. Identificar as arestas que estão ligadas a esses nós;
5. Todas as arestas selecionadas acima que não estejam inteiramente na condição de contorno são marcadas como arestas a dividir;
6. Identificar uma partição de elementos que divida exatamente as arestas marcadas para divisão.

3.4.2 Busca por padrões compatíveis e Definição do melhor padrão disponível

A definição de um padrão de refinamento com base nas arestas marcadas para divisão segue duas as seguintes etapas:

1. em primeiro lugar se busca por padrões de refinamento cujo padrões de refinamentos para os elementos/lado relativos a cada aresta tenham o padrão de divisão requerido pela aresta;

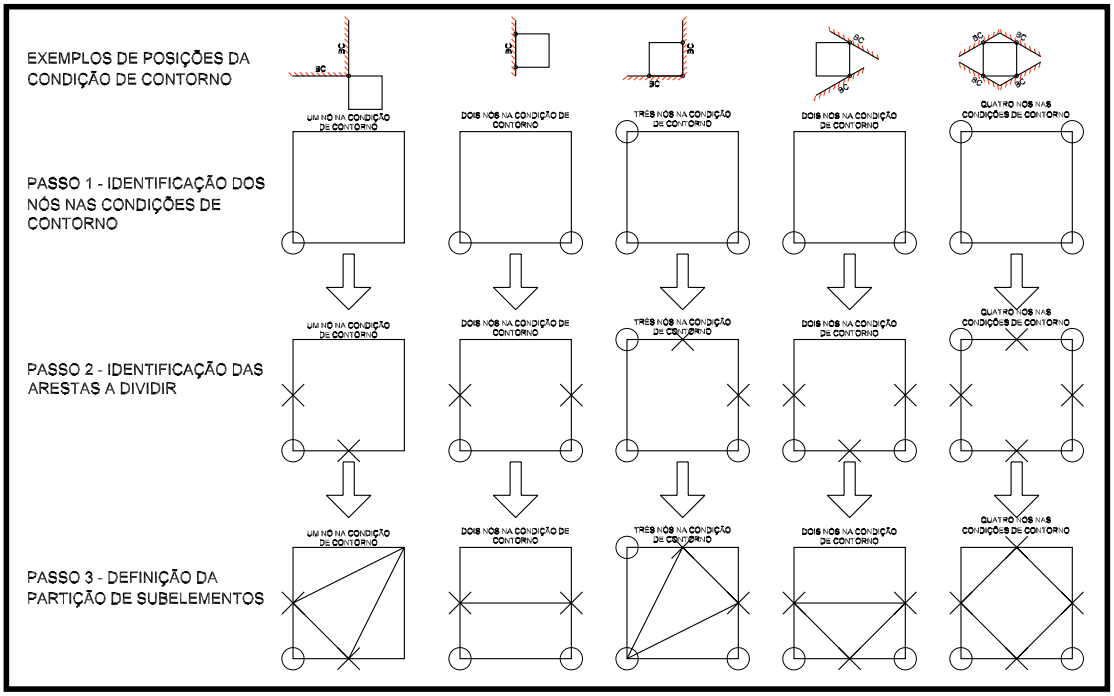


Figura 3.9: Processo de identificação de um padrão de refinamento

2. caso já se tenha elementos vizinhos divididos, retira-se no conjunto de elementos selecionados aqueles com refinamentos não compatíveis com os refinamentos dos elementos vizinhos;

3. Dentre os elementos que satisfazem as condições acima, deve-se definir um único padrão. Tal escolha é um problema ainda não estudado em uma profundidade necessária para se ter alguma conclusão. Atualmente, se define pelo padrão aceitável que conduza ao menor número de graus de liberdade, más mesmo aqui pode ocorrer de mais de um padrão satisfazer tal condição, nesse caso, o primeiro da lista é o selecionado.

3.4.3 Aplicação do Processo à Malha do Projeto de Aeronave yf17

Esse exemplo consiste em uma malha de tetrahedros de um projeto de aeronave militar. Essa malha é de domínio público e foi cedida pelo Prof. Álvaro Coutinho da Coppe-UFRJ. Os padrões de refinamento utilizados foram gerados em um projeto de iniciação científica, desenvolvido pelo aluno Luís Guilherme Decourt.

A malha original é apresentada na Figura 3.10, enquanto a Figura 3.11 mostra a malha após o refinamento direcional. As Figura 3.12 e 3.13 apresentam um detalhe da malha antes e depois do refinamento.

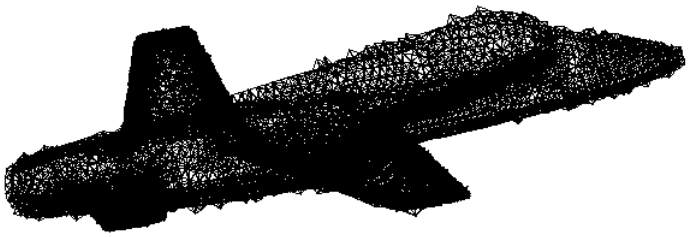


Figura 3.10: YF 17 - Malha Original

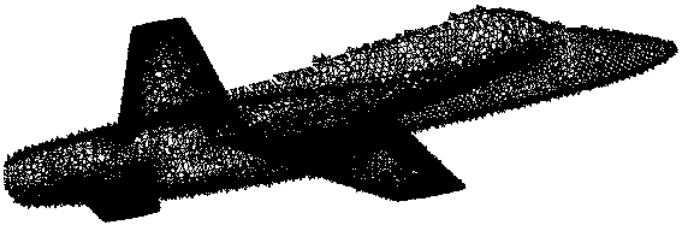


Figura 3.11: YF 17 - Malha Refinada com Refinamento Direcional

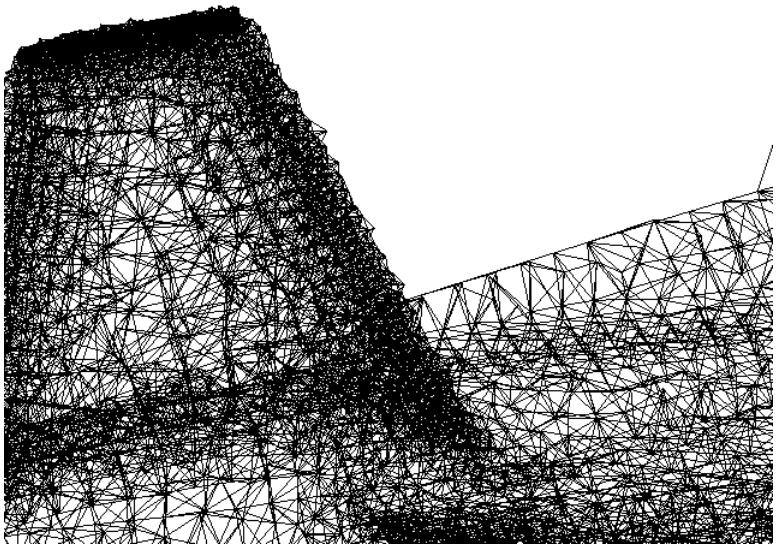


Figura 3.12: Detalhe da malha não refinada

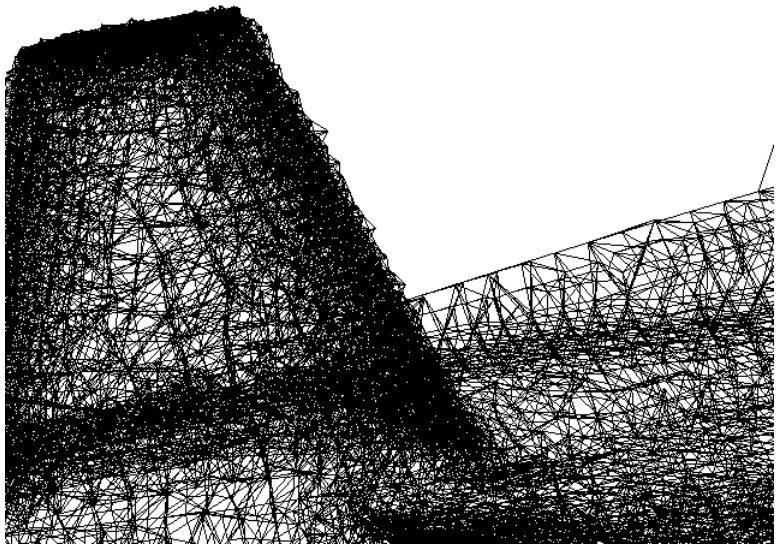


Figura 3.13: Detalhe da malha refinada

Capítulo 4

Método auto-adaptativo

Métodos auto-adaptativos servem visam definir uma malha para a qual o erro da aproximação está dentro de um certo limite. Esses métodos são utilizados em fases iniciais de projeto, de modo a definir uma malha ótima, a qual será utilizada nas análises posteriores.

O método auto-adaptativo aqui proposto é uma extensão da dissertação de mestrado do autor [49], a qual baseou-se em um trabalho de Demkowicz, Devloo e Rachowicz [16], onde foi apresentado um estimador de erros baseado no calculo da diferença entre duas soluções de malhas hierárquicas. A diferença entre [49] e [16] consiste no fato de que o trabalho [16] utiliza uma solução uniformemente refinada ao longo de todo o domínio, enquanto [49] calcula utiliza-se de "patches" de elementos para o cálculo da solução uniformemente refinada.

Outro ponto de destaque do trabalho de Demkowicz, Devloo e Rachowicz foi a apresentação de uma metodologia para a definição do padrão *hp* a ser utilizado na adaptação de um elemento. O processo baseia-se na análise para cada aresta do elemento de qual padrão, com número de graus de liberdade menor que aquele resultante de um refinamento uniforme, mais se aproxima da solução uniformemente refinada.

Aqui se propõe a implementação dessas tecnologias, cálculo da estimativa de erro e definição de padrões de refinamento em um sistema orientado a objetos paralelo, organizado de modo que outros estimadores baseados em patches de elementos possam ser facilmente acoplados, usufruindo assim de um ambiente de auto-adaptação já paralelizado.

4.1 Definições iniciais

Um método *hp* auto-adaptativo tem duas etapas de cálculo :

1. Definição dos elementos a refinar - Estimador de Erros: A definição dos elementos a refinar é proveniente de algum indicativo sobre a qualidade da solução nos elementos. Para tal usa-se estimadores ou indicadores de erro. Existem diversos estimadores e indicadores de erro na bibliografia de elementos finitos, onde alguns são melhores em alguns problemas e piores em

outros. O estimador de erros aqui utilizado é baseado em diferença de soluções, tendo sido proposto inicialmente em [16] e a modificado de modo a utilizar *patches* de elementos ao invés da malha global em [49].

2. Definição de como refinar os elementos selecionados: Tendo se definido quais elementos devem ter seu espaço de aproximação refinado resta definir como refinar o espaço de aproximação: por meio de refinamento h ou p. A análise aqui utilizada, desenvolvida originalmente em [16], baseia-se em verificar qual padrão hp em uma dada aresta leva a melhor aproximação da solução obtida de um espaço uniformemente refinado, porém com um número de graus de liberdade menor que este.

3. Patch de elementos: a definição de patch para esse trabalho é a de um conjunto de elementos aqui denominados elementos de referência do patch, acrescidos de seus vizinhos denominados integrantes do patch. O conjunto de todos os elementos de referência dos patches deve formar uma partição da malha;

Esquematicamente, o método é mostrado na Figura (4.1), onde dada uma malha uniformemente refinada hp e a malha não refinada, é feita a estimativa do erro em cada elemento. Os elementos com erro "relevante" são selecionados para a análise do padrão de refinamento hp. Tendo-se o padrão ótimo de refinamento de cada elemento, este padrão é imposto na malha não refinada, obtendo-se desta forma a malha adaptada.

Esse modelo foi validado e está incorporado em ambientes auto-adaptativos, tais como o 2Dhp90 e 3Dhp90 do ICES - The University of Texas at Austin (ver [14, 15]). No ambiente PZ, esta técnica foi implementada pelo Prof. Devloo, com utilização de análise *mutigrid*. No trabalho de mestrado, o método foi estendido, de modo a utilizar a diferença de subespaços de soluções.

4.1.1 Estimador de Erros Baseado em Diferença de Soluções

Existem trabalhos publicados com diversos tipos de estimador de erro, conforme pode ser observado em [42], [57] e [5], dentre outros. O estimador de erro base desse trabalho, baseado em [16], consiste no cálculo da diferença da solução de duas malhas com espaços de interpolação hierárquicos e distintos. O valor estimado para o erro é calculado elemento a elemento, como sendo a diferença entre o valor do gradiente da solução na malha fina e o valor do gradiente da solução na malha grossa. O valor do gradiente é facilmente obtido para cada elemento, sendo de fato necessário para os cálculos da aproximação.

A principal motivação da utilização desse tipo de estimador é o fato dele ser independente do tipo de problema analisado. O seu problema, na versão original, era o elevado custo da solução uniformemente refinada para todo o domínio, o que motivou a utilização de cálculos locais em patches de elementos.

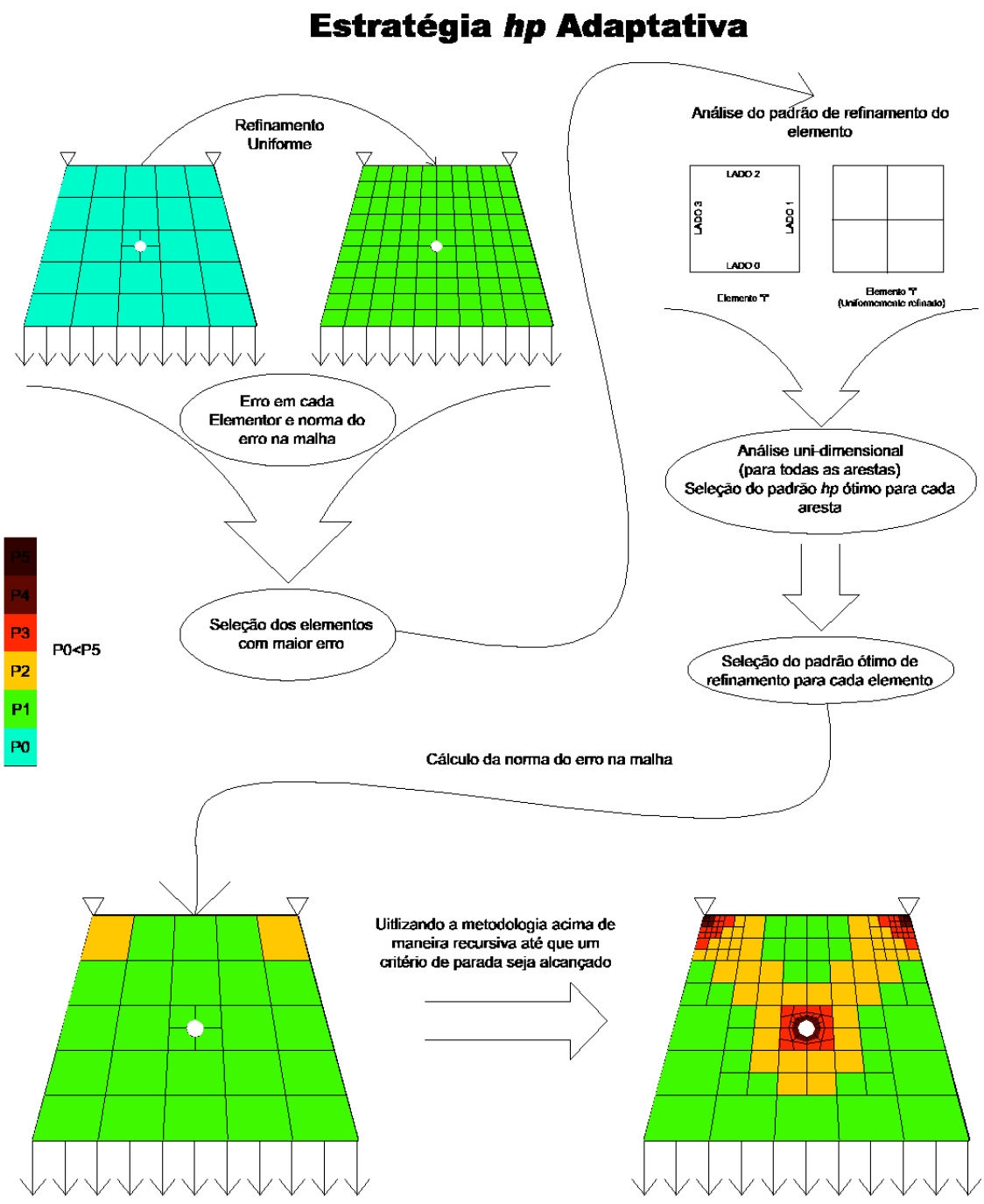


Figura 4.1: Método Auto-adaptativo Base

No caso implementado, onde o cálculo do erro é feito através da diferença dos resultados entre duas malhas hierárquicas, prova-se a convergência do estimador de erro para o caso unidimensional. Esta demonstração para malhas unidimensionais é feita em [16] e reproduzido abaixo. Dado o seguinte problema modelo:

$$\begin{cases} u \in \widehat{u}_D + V \\ B(u,v) = L(v) \,\forall\, v \in V \end{cases} \tag{4.1}$$

onde:

$V \subset H^1(0,l)$: espaço de funções teste;

$\widehat{u}_D \in H^1(0,l)$: função solução na CDC de Dirichlet;

$B(u,v)$, $L(v)$: formas bilinear e linear dos termos da equação diferencial do problema de valor de contorno, podendo ser escritas da seguinte forma:

$$\begin{cases} B(u,v) = \int_0^l (a.u'v' + bu'v + cuv)dx + \beta u(l)v(l) \\ L(v) = \int_0^l (fv)dx + gv(l) \end{cases} \tag{4.2}$$

com $a(x)$, $b(x)$, $c(x)$, $f(x)$ e β satisfazendo as condições de regularidade necessárias.

Dada uma malha $h-p$ com o seu respectivo espaço $V_{hp} \subset V$ a solução aproximada pelo método de Galerkin será:

$$\begin{cases} u_{hp} \in \widehat{u}_D + V_{hp} \\ B(u_{hp}, v_{hp}) = L(v_{hp}) \,\forall\, v_{hp} \in V_{hp} \end{cases} \tag{4.3}$$

Temos, para este caso, que o estimador de erros converge, sendo limitado por:

$$\|u - u_{hp}\|_{L_\infty} \leq C. \inf_{v_{hp} \in V} \|u - (\widehat{u}_D + v_{hp})\|_{L_\infty} \tag{4.4}$$

com $C > 0$, sendo uma constante que depende do espaço de interpolação (ver [16, pág. 4]).

4.1.2 Determinação do Padrão de Refinamento dos Elementos

A técnica apresentada em [16], consiste na minimização da projeção do erro de interpolação, devido ao refinamento *hp* em cada aresta do elemento.

Tendo uma solução uniformemente refinada, o passo seguinte é a projeção da diferença entre a solução da malha refinada e da solução com número de graus menor nas arestas dos elementos da malha, $V_h(e)$, e dada que a solução nos nós externos é igual para a malha refinada e a não refinada $u_{\frac{h}{2},p+1} - u^n_{h,p} = 0$ para estes nós.

Desta forma, o problema passa a ser encontrar a combinação *hp* que minimiza a diferença entre a solução para o padrão hp de teste e a projeção da solução obtida do refinamento *hp* no interior da malha.

Observe que, por trabalhar apenas com espaços de funções e com projeções de espaços, esta metodologia pode ser aplicada a uma série de elementos, independente de sua topologia.

O objetivo do método é definir um refinamento, com parâmetros *hp*, com número de graus de liberdade menor que o número de graus de liberdade da solução uniformemente refinada *hp*, com resultados próximos a esta, ou seja minimizando esse erro.

As combinações nas quais será procurada a combinação *hp* ótimas são todas as combinações possíveis de p^1_{Kref} e p^2_{Kref}, sendo estas as ordens dos subelementos da aresta uniformemente refinada, tais que:

$$p^1_{Kref} + p^2_{Kref} = p_K + 1 \tag{4.5}$$

Também é analisado o erro obtido devido ao refinamento p, sendo:

$$p_{Kref} = p_K + 1 \tag{4.6}$$

O erro entre a aproximação fornecida u e a aproximação calculada $\widetilde{u}$ no elemento K, utilizando o refinamento com número de graus de liberdade menor será:

$$\|e\|^2_{L_2(K)} = \int_{-\Delta_x}^{\Delta_x} (u'_K - \widetilde{u_K}')^2 dx \tag{4.7}$$

A minimização desse erro, que representa um "resíduo" pode ser aproximada por:

$$\left\{ \begin{array}{c} \widetilde{u} \in H^1_0(-\Delta_x, \Delta_x) \mid R = \int_\Omega \psi'.(\widetilde{u}' - u').d\Omega = 0 \forall \psi \in H^1_0 \\ \widetilde{u}(-\Delta_x) = 0 \\ \widetilde{u}(\Delta_x) = 0 \end{array} \right\} \Rightarrow \int_\Omega \psi'_j.(\psi'_i.\widetilde{u})d\Omega - \psi'_j.u' = 0 \tag{4.8}$$

Algebricamente:

$$S_{i,j} = \int_{d\Omega_K} \psi'_i.\psi'_j dx \tag{4.9}$$

$$F_i = \int_{-\Delta_x}^{\Delta_x} \psi'_i.u' dx \tag{4.10}$$

$$\|e\|^2_{L_2(K)} = \int_{-\Delta_x}^{\Delta_x} \sum_i (u_i - \widetilde{u}_i).\psi'_i. \sum_j (u_j - \widetilde{u}_j).\psi'_j dx \tag{4.11}$$

A combinação de refinamento *hp* que apresentar o menor erro para cada aresta será retornada em termos dos parâmetros de refinamento *hp* para cada subelemento da aresta uniformemente refinada, ou ainda um refinamento exclusivamente p.

Esquematicamente, o processo é demonstrado na Figura (4.2).

Dois pontos são destacados em [14] a respeito deste modelo de definição do padrão de refina-

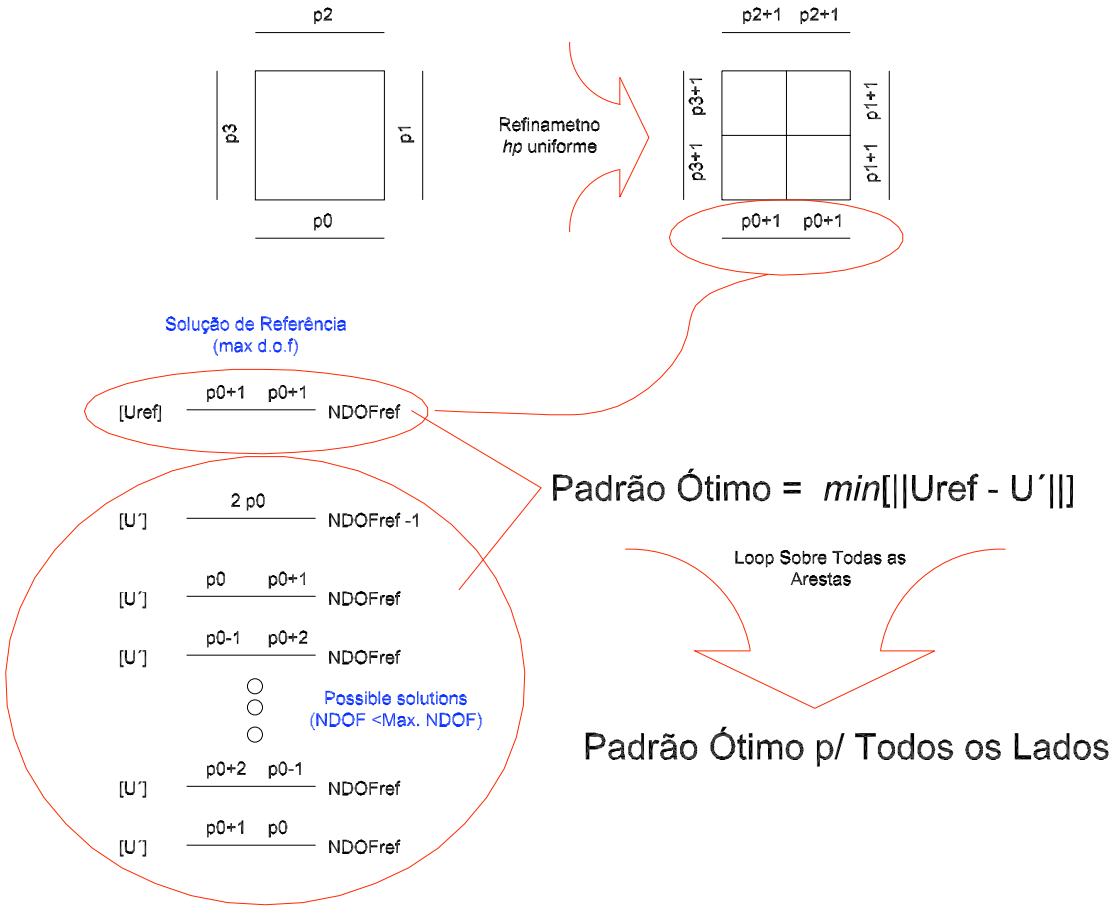


Figura 4.2: Definição do Padrão Ótimo de Refinamento para um Elemento

mento:

1. Esta projeção do erro interpolado leva a padrões de convergência h ótimos e a padrões p próximos ao ótimo. Assim, sua minimização em todos os elementos deve conduzir a uma malha próxima à malha ótima;

2. A estimativa de erro não necessariamente necessita ser calculada globalmente, podendo isto ser feito localmente, elemento a elemento. É este aspecto que motivou o trabalho de mestrado [49].

Um aspecto ressaltado nas possibilidades de trabalhos futuros do trabalho de mestrado [49] é a possibilidade de paralelização do método.

Determinação do padrão de refinamento no caso de utilização de padrões de refinamento uniforme

A técnica descrita anteriormente mostra como obter os padrões de refinamento ótimos para cada aresta, sendo necessário compatibilizar estes resultados dentro de cada elemento e de seus vizinhos.

A análise do padrão final do elemento é feita tomando por base um nó e todos os subelementos que chegam a esse nó, sendo necessárias as seguintes considerações:

- Caso qualquer aresta que contribua para o nó requeira refinamento h em seu padrão ótimo, o refinamento h é adotado, passando-se então à análise do padrão de refinamento p em cada subelemento. Caso não exista refinamento h nos padrões de refinamento ótimos é então adotado o refinamento p uniforme, com p igual ao maior refinamento p de todos as arestas;

- No caso de refinamento hp definem-se os padrões p_K^1 e p_K^2, levando-se em consideração as combinações nas arestas.

Tendo-se refinado os elementos, a malha obtida é considerada como a malha adaptada.

Determinação do padrão de refinamento no caso de utilização de padrões de refinamento direcional

Como proposta de desenvolvimentos futuros, coloca-se o desafio da análise do padrão de refinamento hp para padrões de refinamento não uniformes.

A determinação de um padrão de refinamento direcional deve levar em consideração os padrões de refinamento de seus vizinhos, de modo que os refinamentos entre esses sejam conformes. Tal análise precisa ser feita na malha original, uma vez que ali podem ser identificados restrições à escolha do padrão.

Para essa determinação, primeiramente se busca na base de dados de padrões de refinamento os padrões cujo padrão de divisão das arestas corresponde ao dado obtido da análise unidimensional

feita anteriormente. Em qualquer caso o padrão uniforme deve fazer parte da resposta. O próximo passo é obter dentro do conjunto de padrões admissíveis o padrão que atende aos requisitos de refinamento dos elementos vizinhos.

4.2 Implementação serial

A implementação apresentada na seqüência mostra a estrutura desenvolvida para o processo de auto adaptação, incluindo a descrição da estrutura de dados e também da interface pública de cada classe criada.

Em termos gerais o processo implementado consistindo em:

1. Seleção dos elementos de referência para formação dos *patches*;
2. Definição da partição de elementos de cada *patch*;
3. Criação de uma malha geométrica com base na partição de elementos definida no passo anterior;
4. Criação de uma malha computacional e definição de condições de contorno Dirichlet nos bordos da malha, sendo o valor da condição de contorno igual à solução dos elementos vizinhos que não fazem parte do *patch*;
5. Geração de uma malha uniformemente refinada em *hp* e cálculo da aproximação para a malha uniformemente refinada;
6. Cálculo da estimativa de erro (diferença entre a solução uniformemente refinada e a solução original de cada elemento);
7. Definição dos elementos a ser refinados;
8. Definição do padrão de refinamento de cada elemento;
9. Refinamento da malha original segundo os padrões definidos anteriormente.

4.2.1 Definições iniciais

Na descrição que segue, o termo malha deve ser entendido como sendo um objeto malha computacional, conforme implementado pela classe TPZCompMesh (ver [19]).

Quatro malhas são citadas a saber:

1. Malha original: é a malha que se deseja analisar e, com base na estimativa de erros, adaptar;

2. Malha adaptada: é o resultado do processo de adaptação da malha original. Os padrões de refinamento h, p ou hp são aqueles cuja solução mais se aproxima de uma solução uniformemente refinada, más com um número de graus de liberdade menor que essa;

3. Malha *patch*: consiste em uma "sub-malha" da malha original. É definida tomando por base um conjunto de elementos (pode ser apenas um elemento), para o qual se irá estimar o erro acrescido do conjunto de elementos do entorno desses. A metodologia para a definição dos elementos de referência bem como a definição dos vizinhos que irão compor a malha patch é apresentada em [49];

4. Malha uniformemente refinada: consiste em uma cópia da malha *patch*, onde todos os elementos da malha são adaptados em h e p.

4.2.2 Estrutura de dados para adaptatividade

O código desenvolvido em [49] implementa todos os algoritmos necessários ao cálculo do erro baseado em *patches* de elementos. No desenvolvimento da estrutura de dados proposta, as seguintes diretrizes foram seguidas:

- redução e simplificação das atribuições de cada classe e
- eliminação do armazenamento de ponteiros.

A estrutura de dados e suas principais funcionalidades está descrita na seqüência.

Definição da estrutura de dados

Como diretriz inicial, tem-se uma classe responsável pelo gerenciamento do processo de cálculo do erro e adaptação da malha. Tal abordagem permite que uma interface pública simples possa ser implementada. Essa interface é implementada na classe denominada *TPZAdaptiveProcess*, a qual tem um atributo principal que é a malha original a partir da qual deseja-se obter a malha adaptada, os demais atributos relativos ao processo podem ser obtidos a partir da malha original.

A partir da malha original é possível definir a partição da malha original em um conjunto de elementos que serão tomados por referência para formar com os elementos de seu entorno o conjunto de elementos que definirá a malha *patch*. A definição de quais elementos devem ser utilizados como elementos de referência, bem como, para cada elemento de referência, o conjunto de elementos de entorno que devem ser considerados foi implementado na classe *TPZPatchesGenerator*. Os algoritmos de definição de seleção de elementos de referência e definição dos elementos do patch serão descritos na seção relativa a essa classe.

De modo simplificado, a classe *TPZPatchesGenerator* analisa a malha original e retorna uma lista de objetos do tipo *TPZPatch*. A classe *TPZPatch* armazena duas conjuntos de índices, um para os elementos de referência do *patch* e outro para o conjunto de elementos vizinhos.

Com a definição dos elementos que irão compor a malha *patch*, define-se uma malha *patch*, através da classe *TPZPatchMesh*, essa malha consiste de uma cópia dos elementos da malha original selecionados acrescida de dados que correlacionam os índices dos elementos da malha *patch* com os elementos da malha original. Destaca-se que a cópia dos elementos inclui a cópia de sua solução.

O procedimento de cálculo do erro foi encapsulado na classe *TPZPatchErrorEstimator*, o atributo é um objeto *TPZPatchMesh*, o procedimento de cálculo gera uma malha uniformemente refinada em h e p e calcula o erro como sendo a diferença entre as soluções da malha uniformemente refinada e da malha *patch*. Ressalta-se que outros estimadores de erro, baseados em malhas patch, podem ser derivados dessa classe, bastando implementar o método de cálculo de erro.

Tendo se a estimativa do erro para cada elemento, o passo seguinte é determinar um valor de erro limite para se definir se um elemento deve ou não ser adaptado. Esse cálculo é feito na classe de gerenciamento do processo adaptativo - *TPZAdaptiveProcess*. Tal erro limite é um parâmetro para o método de análise dos padrões de refinamento implementado na classe *TPZPatchErrorEstimator*. A análise do padrão de refinamento propriamente dita é implementada pela classe *TPZComputeRefinement* que tem como dados de entrada: um elemento da malha *patch* e a solução para esse elemento uniformemente refinado. Para cada aresta do elemento verifica-se qual padrão de refinamento da aresta do elemento original mais se aproxima da solução uniformemente refinada com um número de graus de liberdade menor que essa, sendo tal análise feita na classe TPZOneDRef, cuja descrição pode ser encontrada em [49]. No caso de utilização de padrões de refinamento uniforme, os resultados de todas as arestas são pós-processados para a definição do padrão de refinamento final para o elemento.

No caso de utilização de refinamento direcional, a definição do padrão do elemento deve levar em consideração os padrões de refinamento dos elementos vizinhos, de modo a não gerar padrões de refinamento incompatíveis. Desse modo, a definição do padrão de refinamento não pode ser feita exclusivamente com os dados aqui analisados.

Classe *TPZAdaptiveProcess*

O objetivo dessa classe é o de gerenciar o processo de cálculo do erro, a definição dos elementos a refinar e a determinação do padrão de refinamento de cada elemento.

Exceção feita aos métodos de serialização e deserialização, essa foi a única classe que necessitou ser alterada para a paralelização do código, conforme destacado na descrição posterior do algoritmo. Tal alteração consistiu na definição da estrutura de dados específica ao algoritmo paralelo, a criação das tarefas e definição dos pontos de sincronismo. Essa possibilidade de paralelização por meio de poucas alterações pontuais mostra a efetividade da proposta do ambiente OOPar.

Atributos

A estrutura de dados dessa classe consiste dos dados processados a partir de uma malha original a ser adaptada e os resultados do processamento da estimativa do erro. Os atributos da classe são descritos na seqüência:

- *TPZAutoPointer<TPZCompMesh> fOriginalMesh*: ponteiro para a malha computacional original;
- Atributos relacionados a estimação de erro:
  - *vector<TPZPatchErrorEstimator> fPatchErrors*: vetor contendo os objetos de estimação de erro. Em caso de processamento paralelo esse atributo não é definido, sendo substituído por um vetor de identificadores de dados distribuído.
  - *REAL fError*: erro total estimado para a malha original;
  - *multimap<double,int> fErrorToIndex*: mapa múltiplo relacionando um erro calculado a um índice de elemento. A utilização do mapa múltiplo se deve ao fato desse manter os seus dados já ordenados em função do dado chave, o qual é representado pelo primeiro parâmetro do *template*, ou seja, o valor do erro;
  - *REAL fThresholdError*: corresponde ao erro mínimo que um elemento deve ter para ser adaptado. O processo de cálculo utiliza-se do dado acima. Como os elementos do mapas são ordenados em função do erro basta se percorrer os elementos do mapa do último elemento (maior erro) em direção ao primeiro (menor erro), até que a soma dos últimos elementos percorridos corresponda a uma dada proporção do erro total, aqui adotado em 65% do erro total;
  - *void (*fExact)(TPZVec< REAL > &loc, TPZVec< REAL > &result, TPZFMatrix &deriv)*: ponteiro para um função que implementa a solução analítica para um dado problema. Tendo-se a solução analítica pode-se calcular o erro real, o que é importante para avaliar a qualidade do estimador de erros para um determinado tipo de equação diferencial.
  - *TPZVec<REAL> fTrueErrorVec*: caso a solução analítica seja fornecida, esse vetor é inicializado e preenchido com o valor do erro real para cada elemento da malha.
  - *REAL fTrueError*: corresponde a soma dos erros reais, calculados quando se dispõe da solução analítica;
  - *REAL fEffectivity:* índice de efetividade do estimador de erros. É calculado quando se dispõe da solução analítica para o problema, sendo definido em função da norma adotada como sendo:

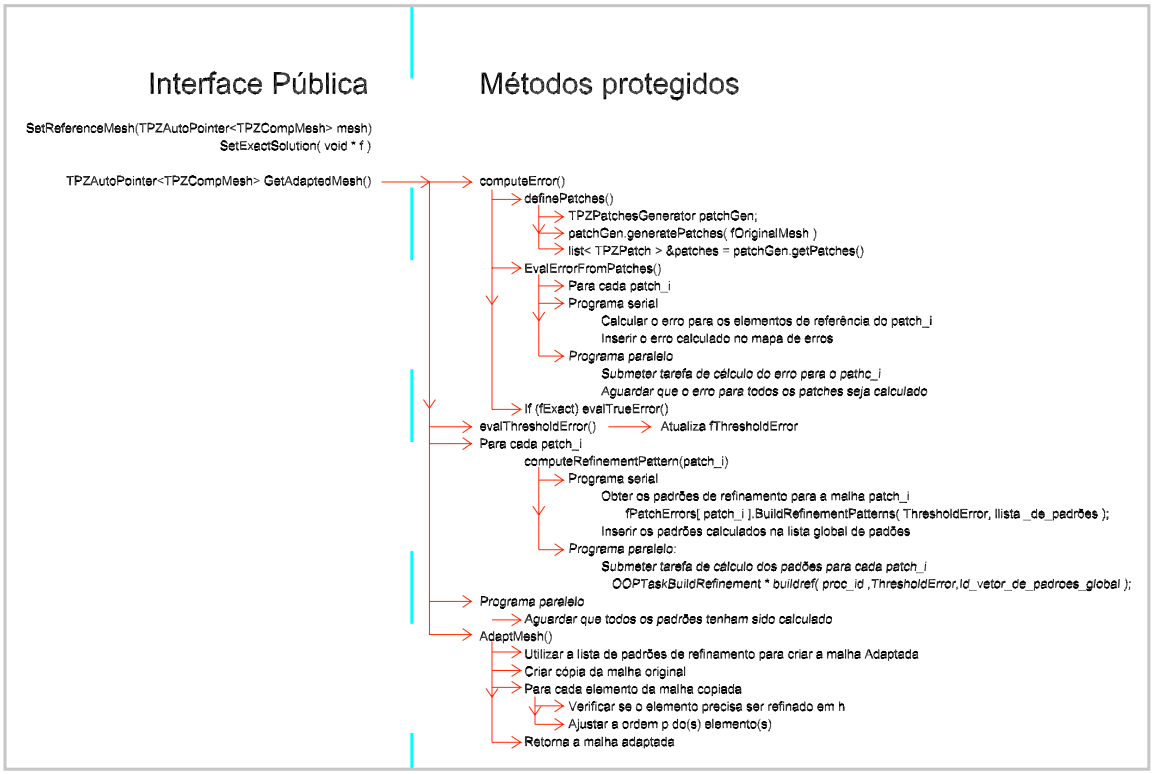


Figura 4.3: Fluxograma para a obtenção de malha adaptada

$$\Theta = \frac{\eta}{\|e\|}$$

ou seja a efetividade do estimador de erros é a relação entre a norma do erro estimado e a norma do erro real.

- *list< TPZAutoPointer<TPZRefineData> > fRefinements*: lista de objetos que definem o padrão de refinamento para todos os elementos da malha original. Caso um elemento não necessite ser adaptado, o seu padrão atual é inserido na lista.

Principais métodos e interface pública

O principal algoritmo implementado nessa classe é aquele para a obtenção de uma malha adaptada, sendo o dado de entrada a malha a ser analisada. Essa malha deve ter sua solução já calculada. O fluxograma para o método de obtenção da malha adaptada é mostrado na Figura 4.3.

Destacam-se quatro operações aqui realizadas:

- *REAL computeError()*: gerencia o processo de particionamento da malha original em *patches* de elementos (*void definePatches()*). Para cada *patch* identificado, gera-se uma malha *patch* (*TPZPatchMesh*), e utiliza-se essa para alimentar um objeto *TPZPatchErrorEstimator*, os quais implementam todo o processo de estimação do erro baseado em malhas *patch*;

- *REAL evalThresholdError()*: aqui é calculado o erro mínimo para que um elemento seja adaptado. O critério aqui adotado é o de que os elementos com maior erro, cuja soma da estimativa do erro corresponda a, no mínimo, 65% do erro total estimado sejam adaptados. Após a soma dos últimos elementos do mapa atingir o limite de 65% do total do erro, toma-se como erro limite ("*threshold error*") a média entre o erro do elemento atual, com cuja contribuição se atingiu o limite de 65%, e o próximo elemento do mapa;

- *void computeRefinementPattern(int patch_idx, list<TPZRefineData> &patchRefData)*: tendo-se o erro limite, para cada objeto *TPZPatchErrorEstimator*, solicita-se o cálculo dos padrões de refinamento, através do método *TPZPatchErrorEstimator :: BuildRefinement-Patterns(patch_idx, patchRefData)*.

- *TPZAutoPointer< TPZCompMesh > adaptMesh()*: utiliza a lista de padrões de refinamento calculada anteriormente para gerar uma malha adaptada. Isso é feito com base nas informações do objeto *TPZRefineDataUniform*, o qual tem um *flag* que indica ou não a necessidade de refinamento. O outro dado da classe é um vetor com as ordens p de cada sub elemento. Caso o elemento não seja dividido, o vetor com ordens p tem apenas um dado que é a ordem p para o elemento original.

Os demais métodos públicos da classe dizem respeito ao acesso a estrutura de dados conforme descrito na seqüência:

- *void setReferenceMesh(TPZAutoPointer<TPZCompMesh> CMesh)*: define a malha a ser adaptada

- *void setExactSolution (void(*f)(TPZVec< REAL > &loc, TPZVec< REAL > &val, TPZFMatrix &deriv))*: define uma função correspondente à solução analítica de um determinado problema;

- *TPZAutoPointer<TPZCompMesh> GetAdaptedMesh()*: retorna a malha adaptada, conforme descrito anteriormente;

- *REAL getError()*: retorna o erro estimado para a malha original;

- *REAL getTrueError()*: caso exista a solução analítica, esse valor corresponde ao erro real da solução da malha original;

- *REAL getEffectivity()*: corresponde a razão entre o erro estimado e o erro real, que só é calculado caso seja definida a solução analítica para o problema;

- *void Print(ostream &out)*: imprime a estrutura de dados no dispositivo indicado por *out*.

Classe *TPZPatch*

Essa classe é utilizada para armazenar os índices dos elementos da malha original que irão compor uma malha *patch*. Os elementos de uma malha *patch* são classificados em dois tipos: os elementos de referência para uma malha *patch*, para os quais o erro deve ser estimado, e os índices dos seus vizinhos que irão conjuntamente compor a malha *patch*.

O conjunto dos elementos de referência para todos as malhas *patch* formam uma partição do domínio. Um dado elemento pode constar de várias malhas *patch*, mas como elemento de referência de uma única malha *patch*.

Atributos

A estrutura de dados dessa classe consiste em dois conjuntos de inteiros, um para os índices de todos os elementos que irão compor a malha *patch* e outro para os elementos de referência. Ressalta-se que os índices dos elementos de referência devem constar em ambos os conjuntos.

- *set<int> fRefIdx*: conjunto de índices dos elementos de referência
- *set<int> fElToPatch*: conjunto de índices de elementos que irão compor a malha *patch*. Esse conjunto inclui os elementos de *fRefIdx*;

Principais métodos e interface pública

A interface pública dessa classe tem como finalidade o controle e gerenciamento do acesso às informações dos índices de elementos que fazem parte do *patch*. Desse modo, não há algoritmos implementados, apenas métodos para inserir ou acessar os dados, conforme descrito abaixo:

- *void insertReferenceIdx(int index)*: insere o índice informado nos conjuntos de elementos de referência do *patch* e no conjunto de elementos do *patch*;
- *void insertPatchIdx(int index)*: insere o índice informado no conjunto de elementos do *patch*;
- *set<int> & getRefIdx()*: retorna o conjunto de índices de elementos de referência para o *patch*;
- *set<int> & getPatch()*: retorna o conjunto de índices de elementos do *patch*;
- *bool IsPatchReference(int celIndex)*: retorna verdadeiro caso o índice informado esteja no conjunto de índices de elementos de referência do *patch* e falso caso contrário;
- *void Print(ostream &out)*: imprime a estrutura de dados da classe;
- *void Write(TPZStream &buf, int withclassid)*: serializa os dados da classe no vetor de dados *buf*;

- *void Read(TPZStream &buf, void *context)*: define a estrutura de dados do objeto a partir do vetor de dados *buf*.

Classe *TPZPatchGenerator*

Essa classe foi concebida para realizar o processo de partição da malha original em termos de conjuntos de elementos de referência e indicar para cada um desses conjuntos quais os vizinhos necessários para compor uma malha *patch*. O processo de definição dos elementos de referência aqui utilizado foi reproduzido de [49], sendo os seus algoritmos descritos na seqüência. O resultado final desse processamento é uma lista de objetos do tipo *TPZPatch*, descrito anteriormente.

Atributos

O único atributo da classe é a lista de objetos do tipo *TPZPatch*, que serão gerados durante o processamento. A malha de referência para o processo é fornecida como parâmetro. O atributo é o seguinte:

- *list<TPZPatch> fPatches*: lista de objetos patch para a malha fornecida como parâmetro.

Principais métodos e interface pública

O principal algoritmo aqui implementado é aquele que particiona a malha original em elementos de referência e, com base na análise dos grafos de nós para elementos define quais elementos devem fazer parte de um determinado *patch*. Tal algoritmo é mostrado na Figura 4.4.

A definição dos *patches* é composta de três etapas: particionamento da malha em elementos de referência, definição dos elementos vizinhos para compor o *patch* e verificação da necessidade de inserção de condições de contorno que não fazem parte da vizinhança "imediata" de um elemento de referência. Essas tarefas estão implementadas nas seguintes funções:

- *void getRefPatches(TPZAutoPointer<TPZCompMesh> CMesh, set<int> References)*: percorre-se todos os elementos da malha computacional dada. Cada elemento da malha identifica seu elemento de referencia geométrico e esse identifica seu elemento de referência para a formação de um *patch*. Essa identificação segue o seguinte procedimento:
  - elementos de condição de contorno não são analisados;
  - cria-se um vetor ordenado onde são inseridos o elemento e todos os seus ancestrais ordenadamente,
  - percorre-se o vetor do ancestral mais antigo para o mais recente. O elemento de referência corresponde ao primeiro ancestral encontrado que tenha pelo menos um elemento computacional vizinho,

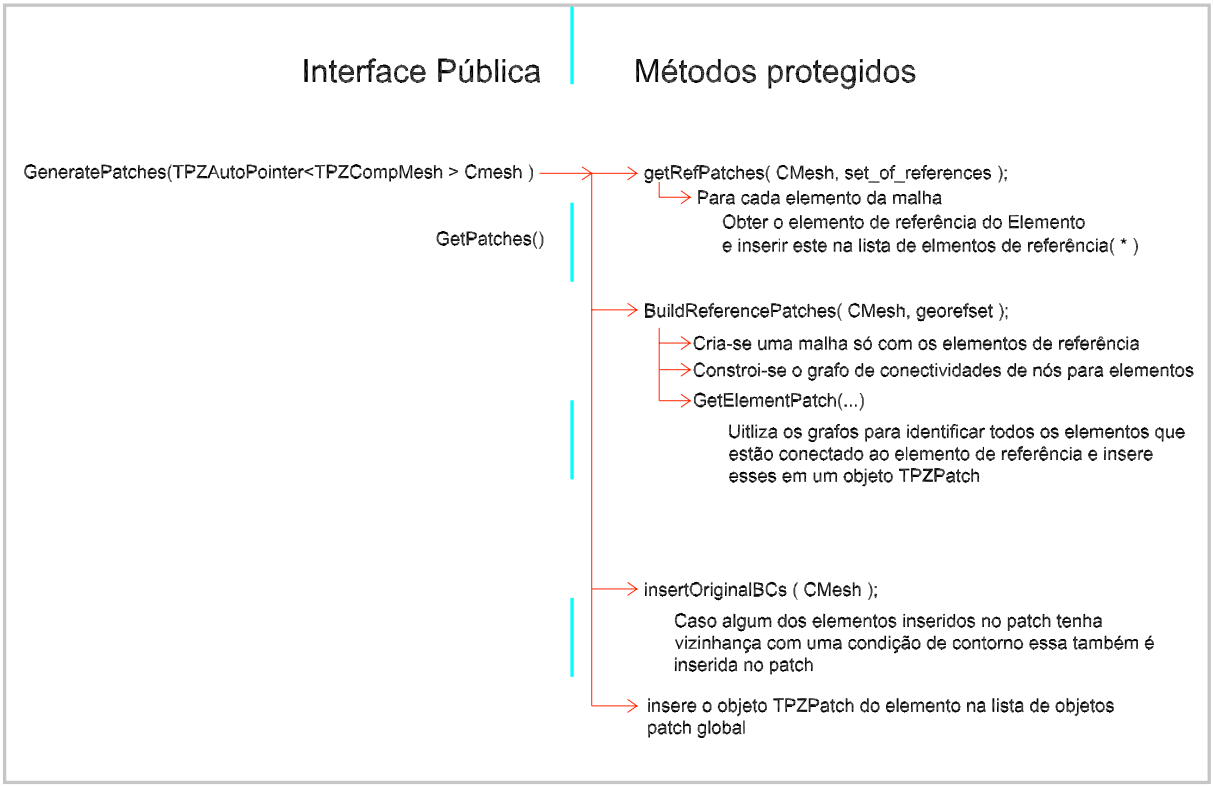


Figura 4.4: Definição dos *patches* de uma malha

– insere-se o índice do elemento identificado no conjunto de elementos de referência. Ressalta-se que diversos elementos da malha original podem ter um mesmo elementos de referência, por isso, aqui foi utilizado um conjunto, de modo que ao final do procedimento não se tenha índices repetidos e se possa verificar se esse conjunto representa de fato a partição da malha original;

- *void BuildReferencePatches(TPZAutoPointer<TPZCompMesh> CMesh, set<int> References)*: para cada elemento de referência identificado no método descrito acima, determina-se os elementos que devem fazer parte do patch desse elemento. Tal processo está descrito em [49].

Classe *TPZPatchMesh*

Essa classe implementa a representação de uma malha *patch*, incluindo o mapeamento dos índices de seus elementos para os índices dos elementos na malha original. Nessa classe é feita a cópia dos elementos designados no objeto *TPZPatch*, incluindo sua solução. Os índices locais dos elementos de referência são armazenados em um conjunto de modo a indicar para quais elementos é necessário o cálculo do erro.

Atributos

Os atributos dessa dizem respeito ao mapeamento entre índices de elementos locais e índices de elementos na malha original. Além desses dados há um dado para armazenar a estrutura de dados de uma malha computacional, que representa o conjunto de elementos do *patch*, além de um ponteiro para a malha *patch* uniformemente refinada. No caso de processamento paralelo, a malha uniformemente refinada não consta da lista de objetos a ser transmitidos, uma vez que essa pode ser construída localmente no processador remoto.

- *TPZAutoPointer<TPZCompMesh> fPatchCompMesh*: ponteiro para a malha computacional que representa o *patch* de elementos. Essa malha é construída como uma cópia da estrutura de dados da malha original para os elementos selecionados no objeto *TPZPatch*, incluindo os dados relativos à solução desses. Mesmo no caso de processamento paralelo, essa malha é construída no processador zero. Se por um lado tal decisão torna essa parte do código serial, por outro lado, a malha original não precisa ser transmitida para todos os processadores;

- *TPZAutoPointer<TPZCompMesh> fFineCompMesh*: esse ponteiro é construído inicialmente como uma cópia da malha clone acima descrita. Após a cópia todos os elementos da malha são refinados uniformemente em h e p. Após o refinamento, a solução para essa malha é recalculada. A diferença entre a solução para essa malha e para a malha original é tomada como a estimativa do erro;

- *vector <int> fLcToGlbIdx*: em cada posição desse vetor é armazenado o índice do elemento correspondente na malha original;

- *TPZPatch fPatch*: *patch* de elementos que irá compor a malha *patch*;

- *set < int > fLocalRefIndex*: conjunto de índices de elementos da malha *patch* cujo erro é necessário calcular, ou seja os índices na malha *patch* dos elementos de referência do *patch*;

Principais métodos e interface pública

O objetivo dessa classe é o de gerenciar a cópia dos elementos indicados no *patch* de elementos da malha original para a malha *patch*. Essa cópia inclui a solução associada às funções de forma dos elementos copiados, bem como os coeficientes de restrições entre funções restritas. Durante a cópia é feito um mapeamento entre índices de nós e elementos, geométricos e computacionais, da malha original para a malha clone e vice-versa.

O método que gerencia todo o processo de cópia da malha é: *void generatPatch(TPZCompMesh &refmesh, TPZPatch &patch)*, cuja seqüência de operações é mostrada na Figura 4.5.

Abaixo são descritas as operações necessárias:

- *void CloneGeometricMesh(TPZCompMesh &refMesh, map<int,int> &gl2lcGelIdx)*: realiza a tarefa de copiar os nós e elementos geométricos da malha original necessários para a criação da malha *patch*. Note que aqui não basta apenas clonar os nós e elementos selecionados para o *patch*, uma vez que, no caso de cálculo de restrições, há a necessidade de se utilizar relações de transformação de um ancestral para seu vizinho. Desse modo, todos os ancestrais dos elementos do *patch* são copiados para a malha *patch*. Tendo-se identificado os elementos necessários, pode-se definir quais precisam ser copiados e finalmente pode-se criar as cópias dos elementos da malha original para a malha *patch*. A implementação da cópia foi feita por meio de criação de construtores de cópia nas classes do ambiente PZ. Tais construtores gerenciam a cópia da estrutura de dados de um elemento da malha original para uma outra malha qualquer, utilizando os mapeamentos entre os índices de uma malha e outra que podem ser gerados *a priori*. De modo sucinto, o encadeamento de operações nesse método é o seguinte:

  - *void CloneGeometricNodes(TPZCompMesh & refMesh, map<int,int> & gl2lcNdIdx)*: clona os nós geométricos tomando as informações de índices e identificadores por meio do mapeamento feito *a priori*;
  - *void GetGeoList(TPZCompMesh &refMesh, list<int> &geltoclone)*: insere toda a cadeia de elementos ancestrais na lista de elementos à copiar;

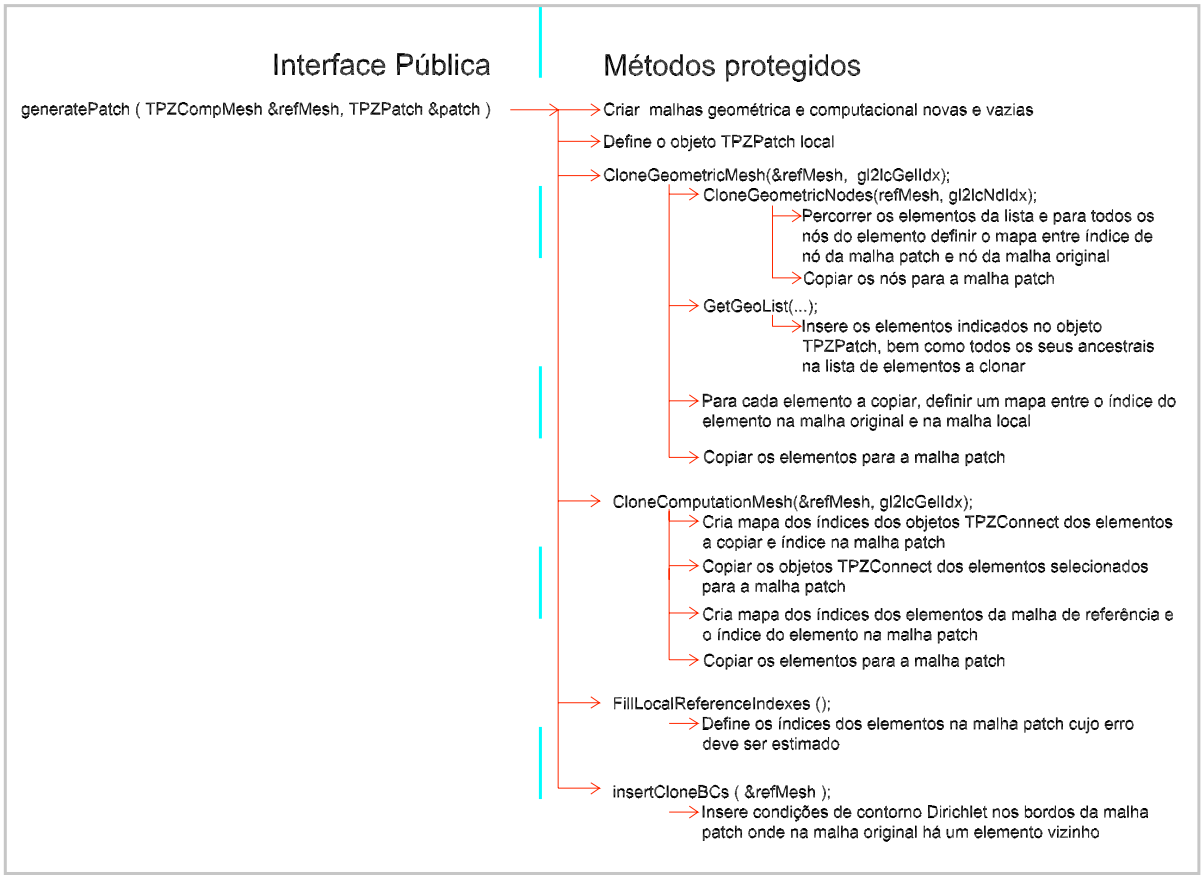


Figura 4.5: Geração de uma malha *patch*

  - define um mapeamento entre os índices dos elementos da malha original e da malha *patch* e dimensiona a estrutura de dados da malha *patch* para receber a cópia dos elementos selecionados da malha original;
  - Percorre-se o conjunto de elementos do *patch* criando uma cópia de cada um dos elementos selecionados;

- *void CloneComputationMesh(TPZCompMesh &refMesh, map<int,int> & gl2lcGelIdx)*: o processo de criação da malha clone computacional é similar ao da malha geométrica. O aspecto a destacar é o da cópia dos objetos *TPZConnect*. Em caso de restrição, há a necessidade de se copiar todo o conjunto de *connects* do qual o *connect* depende e reproduzir essa condição na malha *patch*. Os seguintes métodos são aqui realizados:
  - *refMesh->CopyMaterials(fPatchCompMesh)*: copia os materiais da malha original para a malha *patch*;
  - monta o mapa de índices de *connects* da malha original para a malha *patch*;
  - copia os connects da malha original para a malha *patch* incluindo a lista de dependências caso exista;
  - copia os elementos computacionais da malha original para a malha *patch*;
  - void *InitializeSolution(TPZCompMesh &refMesh, map<int,int> &gl2lcconIdx)*: copia a solução da malha original para a malha *patch*;

- *FillLocalReferenceIndexes()*: identifica os elementos da malha *patch* cujo erro precisa ser estimado;

- *insertCloneBCs (TPZCompMesh &refMesh)*: insere condições de contorno nos bordos da malha *patch* para os quais, na malha original, existiam vizinhos. Essas condições de contorno, impõem a solução do elemento vizinho original no bordo do elemento da malha *patch*.

Os demais métodos públicos tem por objetivo o controle de acesso à estrutura de dados, sendo descritos abaixo:

- *TPZAutoPointer<TPZCompMesh> getMesh()*: retorna a malha *patch*;

- *TPZAutoPointer<TPZCompMesh> getFineMesh()*: retorna a malha *patch* uniformemente refinada. Note que essa malha é gerada durante o processo de cálculo do erro, não sendo transmitida ou restaurada nos processos em paralelo;

- *void setFineMesh(TPZAutoPointer< TPZCompMesh > &fineMesh)*: define a malha uniformemente refinada. Tal operação é feita após a obtenção dessa malha, durante o processo de cálculo da estimativa do erro;

- *set < int > & getLocalReferenceIndexes()*: armazena o conjunto de índices de elementos da malha *patch* cujo erro deve ser estimado;

- *bool verifySolution()*: método utilizado para depuração do código. Aqui se compara a solução original, copiada para a malha *patch*, com uma solução obtida a partir do processamento da malha *patch*. Caso haja algum problema com a definição das condições de contorno, haverá diferença nas soluções, possibilitando que se analise a malha *patch* antes de qualquer outra operação necessária ao cálculo da estimativa de erro;

- *void Print(ostream &out)*: imprime a estrutura de dados da malha *patch*;

- *void ShortPrint(ostream &out)*: idem ao anterior excluindo a impressão da malha;

- *void Read(TPZStream &buf, void *context)*: preenche a estrutura de dados da classe a partir de um vetor de dados;

- *void Write(TPZStream &buf, int withclassid)*: serializa a estrutura de dados da classe para um vetor de dados.

Classe TPZPatchErrorEstimator

A principal atribuição dessa classe é o cálculo e gerenciamento do uso da solução de uma malha *patch* uniformemente refinada para a obtenção da estimativa do erro, bem como a determinação dos padrões de refinamento. O cálculo da estimativa do erro utiliza a comparação da solução da malha *patch* com a solução da malha *patch* uniformemente refinada. A solução da malha uniformemente refinada também é utilizada para a determinação do padrão ótimo de refinamento para cada aresta do elemento.

Há assim dois métodos principais aqui implementados: um para o cálculo do erro (*void evalError()*) e um para a obtenção dos padrões de refinamento dos elementos de referência (*void BuildRefinementPatterns(REAL &thresholderror, list< TPZAutoPointer< TPZRefineDada> >)*).

Atributos

A estrutura de dados da classe é simplificada, uma vez que o objeto *TPZPatchMesh* encapsula boa parte das informações necessárias aos cálculos. Além do objeto do tipo *TPZPatchMesh*, são armazenados dois mapas, um relacionando o índice de elementos refinados ao do elemento pai correspondente e um mapa de índice de elemento na malha patch para o erro calculado.

- *TPZPatchMesh fPatchMesh*: objeto que armazena a estrutura de dados de uma malha do tipo *patch*.

- *map< int, double > fLocalIdxError*: mapa relacionando índice do elemento ao erro estimado para esse elemento;

- *map<int,int> fFineToCoarse*: mapa relacionando índice do elemento filho ao índice do elemento pai. Esse mapa é utilizado por questão de facilidade de acesso à informação, uma vez que utilizando as referências entre malhas computacionais e geométricas, seria possível obter tal relação, entretanto, para tal seria necessário manipular a referência da malha geométrica para cada elemento, o que não seria eficiente.

Principais métodos e interface pública

Para cumprir com as duas atribuições principais dessa classe: o cálculo da estimativa do erro e o cálculo dos padrões de refinamento, o primeiro passo é a obtenção de uma malha uniformemente refinada, a qual é posteriormente processada de modo que sua solução possa ser utilizada tanto para a estimação do erro como para a análise dos padrões de refinamento.

A interface pública dessa classe é descrita abaixo, sendo as principais funções descritas nos algoritmos mostrados na seqüência:

- *void setPatchMesh(TPZPatchMesh &PatchMesh)*: define a malha *patch* sobre a qual será feito todo o processamento;

- *void evalError()*: gerencia todo o processo de cálculo da estimativa do erro, conforme descrito no Algoritmo 1.

- *void getGlobalError(multimap< double,int > &OrgIndxError)*: insere o mapa de erros calculado para a malha *patch* no mapa múltiplo global, passado como parâmetro;

- *TPZPatchMesh & getPatchMesh()*: retorna a malha *patch*;

- *void BuildRefinementPatterns(REAL errorthreshold, list< TPZAutoPointer<TPZRefineData> > &refpatterns)*: calcula os padrões de refinamento para os elementos de referência do *patch*. Tal cálculo é implementado na classe *TPZComputeRefinementPattern* que será descrita posteriormente, sendo aqui preenchida a estrutura de dados necessária ao cálculo;

- *void Print(ostream & out)*: imprime a estrutura de dados da classe;

- *void ShortPrint(ostream & out)*: idem ao anterior, porém sem imprimir as malhas;

- *virtual void Read(TPZStream &buf, void *context)*: preenche a estrutura de dados da classe a partir de um vetor de dados;

Algorithm 1 Cálculo da estimativa do erro

1. *TPZAutoPointer<TPZCompMesh> fineMesh = fPatchMesh.getFineMesh();*
   if (!fineMesh) generateRefinedMesh()
   Obtenção da malha uniformemente refinada. A rotina contida no método *void generateRefinedMesh()* é o seguinte:
   (a) Para todos os elemento da malha patch original - *fPatchMesh*
   (b) obter a ordem p do elemento
   (c) dividir o elemento (refinamento h), interpolando a solução do elemento original nos elementos filhos
   (d) Para cada elemento filho: definir a ordem p do elemento filho como sendo a ordem do elemento original + 1;
2. *void ComputeFinetoCoarse()*
   define o mapeamento entre índices dos elementos da malha uniformemente refinada (filhos) com os índices dos elementos da malha original (pai);
3. *processFineMesh()*:
   Calcula a solução para a malha uniformemente refinada;
4. Para cada elemento da malha uniformemente refinada:
5. Obter o elemento da malha uniformemente refinada (filho) e o elemento da da malha *patch* original (pai)
6. Construir o mapeamento entre os dois elementos: *transform*
7. *REAL error = ElementError(filho, pai, transform)*
   Cálculo a estimativa do erro no elemento como sendo a semi-norma H1 da diferença da solução dos dois elementos. A descrição do algoritmo pode ser obtida em [49].
8. Acrescer ao mapa do índice do elemento pai o erro calculado.

- *virtual void Write(TPZStream &buf, int withclassid)*: serializa a estrutura de dados em um vetor de dados;

- *virtual int ClassId() const*: identificador da classe para efeito de identificação do tipo de objeto a restaurar em caso de transmissão do objeto para outro processador.

Classe *TPZComputeRefinement*

Essa classe não tem estrutura de dados associada servindo para encapsular as funções de análise do padrão de refinamento de um elemento.

O problema aqui analisado é o de definir qual padrão *hp* de refinamento para uma dada aresta resulta na solução mais próxima, em termos de semi norma $H1$, da solução nessa mesma aresta obtida da malha uniformemente refinada. O padrão de refinamento é armazenada em objetos *TPZRefineData* os quais são descritos posteriormente.

Dois métodos são implementados nessa classe:

- *void AnalyseElement(TPZInterpolatedElement *cint, TPZRefineDataUniform &refData)*: aqui são definidos os padrões de refinamento *hp* ótimos para cada aresta do elemento analisado. O algoritmo utilizado foi reproduzido de [49];

- *void DeduceRefPattern(TPZVec<SPZRefPattern> &refpat, TPZVec<int> &cornerids, TPZVec<int> &porders, int originalp)*: Analisa-se o conjunto de padrões ótimos para cada aresta para determinar o padrão de refinamento para o elemento. A descrição do processo de definição do padrão de refinamento do elemento foi baseado em [16].

TPZRefineData

Essa classe é uma classe *virtual* que serve para armazenar as informações básicas para a adaptação de um elemento, a saber: índice do elemento analisado, *flag* indicando a necessidade ou não de refinamento h. Os métodos e funções implementadas servem para gerenciar o acesso a esses dados.

Atributos

Os dois atributos da classe são:

- *bool fDivide*: booleano que indica ou não a necessidade de refinamento h do elemento e

- *int fElementIndex*: índice do elemento analisado.

Principais métodos e interface pública

Além dos métodos de acesso e definição da estrutura de dados, essa classe implementa a interface da função *virtual* em que, se passando a malha relativa ao elemento analisado, realiza-se o processo de adaptação do elemento. Esse processo é distinto em caso de refinamento uniforme e direcional.

Os outros métodos da classe são os seguintes:

- *int getIndex()*: retorna o índice do elemento analisado;
- *void setIndex(int Index)*: define o índice do elemento analisado;
- *bool getDivide()*: retorna o booleano indicador de necessidade de refinamento h;
- *void setDivide(bool Divide)*: define o booleano indicador de necessidade de refinamento h;
- *void Print(ostream & out)*: imprime a estrutura de dados da classe;
- *virtual void Write(TPZStream &buf, int withclassid)*: serializa a estrutura de dados para um vetor de dados;
- *void Read(TPZStream &buf, void *context)*: restaura a estrutura de dados a partir de um vetor de dados.

Classe TPZRefineDataUniform

Implementa a derivação de *TPZRefineData* que trata de refinamentos uniformes. Para tal há o acréscimo de um vetor com as ordens p dos subelementos em caso de refinamento h. Caso não ocorra refinamento h, o vetor tem apenas um dado, correspondendo à ordem p do elemento.

Atributos

O único atributo aqui definido é o vetor de inteiros: *TPZVec<int> fPOrder*, que contém a ordem p dos subelementos.

Principais métodos e interface pública

O principal método da classe é o método: *void Divide(TPZAutoPointer<TPZCompMesh> mesh)*, o qual implementa a divisão do elemento ao qual o objeto se refere. O Algoritmo 2 mostra o processo de adaptação do elemento.

Algorithm 2 Adaptação de um elemento utilizando padrões de refinamento uniformes

1. Verificar se o elemento precisa ser dividido (refinamento h): Tal informação é armazenada no *flag* de divisão da classe pai *TPZRefineData*

2. Em caso afirmativo:

   (a) Utilizar o método *Divide*, sobre o elemento e armazenar os índices dos elementos filhos

   (b) Para cada elemento filho:

      i. Obter a ordem p correspondente ao filho (vetor de ordens p da classe)

      ii. Mudar a ordem p do elemento para ordem estabelecida

3. Em caso negativo: alterar a ordem p do elemento para a ordem estabelecida no vetor de ordens p da classe

4.3 Paralelização utilizando o OOPar

Na descrição do ambiente OOPar já foi colocado que um dos objetivos desse ambiente é o de prover uma ferramenta que possibilite a paralelização de um código serial com impacto mínimo sobre esse.

A implementação do estimador de erros baseado em *patches* de elementos é naturalmente paralelizável, uma vez que cada sub-malha pode ser processada independentemente das demais. Tal abordagem, de paralelizar no nível de processamento de cada sub-malha, não é baseada em análises de granularidade e sim pela facilidade de implementação, uma vez que o método foi assim concebido.

A paralelização do código foi feita em dois pontos básicos:

1. Cálculo da estimativa de erro (diferença entre a solução uniformemente refinada e a solução original de cada elemento): onde a chamada para a função serial foi substituída pela criação de uma tarefa;

2. Definição do padrão de refinamento de cada elemento: onde a chamada serial foi substituída pela criação de tarefas de cálculo distribuído.

Essas alterações na classe que gerencia o processo auto-adaptativo, *TPZAdaptiveProcess*, foram feitas através da inserção de diretivas de compilação - *#ifdef*, onde caso o OOPar esteja definido para uso o código paralelo é utilizado. Originalmente, a implementação dessa classe foi feita em pouco mais de 720 linhas de código. As alterações consumiram aproximadamente 120 linhas de código. Ou seja, foi alterado pouco mais de 15% do código existente para a implementação do algoritmo adaptativo.

Para a paralelização do código foram criados novos atributos para a classe *TPZAdaptiveProcess*, relativos aos dados distribuídos que são objeto de cálculo, bem como as variáveis de controle de

versão associadas. Foram definidas tarefas para cálculo e atualização de valores de erro e de parâmetros de refinamento. Como as tarefas dizem respeito à operações sobre as malhas *patch*, as classes *TPZPatchMesh* e *TPZPatchErrorEstimator* necessitaram ser serializadas.

4.3.1 Alteração da Estrutura de Dados da Classe TPZAdaptiveProcess

Para implementar o algoritmo paralelo, foi necessário acrescer alguns atributos a classe TPZAdaptiveProcess:

- *vector<OOPObjectId> fPatchErrorsId*: vetor de identificadores dos objetos distribuídos correspondentes aos objetos de estimação de erro em malhas *patch*;
- *OOPObjectId fErrorToIndexId*: identificador de dado distribuído correspondendo a um vetor de erros global. Após o cálculo do erro em todas as malhas *patch*, os dados desse vetor são inseridos no mapa de erro para elemento, de modo a possibilitar o cálculo do erro limite para refinamento;
- *OOPObjectId fRefinementsId*: identificador de dado distribuído relativo a uma lista de objetos de padrão de refinamento.

O vetor de identificadores de objetos distribuídos de cálculo de erro substitui o vetor de objetos TPZPatchErrorEstimator, sendo esses enviados aos processadores remotos. Já os outros dois dados dizem respeito às variáveis globais que irão receber o cálculo do erro distribuído e também o vetor de padrões de refinamento calculado remotamente.

4.3.2 Serialização da estrutura de dados

Tipicamente são transmitidos quatro tipos de dados durante o processo adaptativo:

1. Objetos do tipo TPZPatchErrorEstimator, os quais levam consigo as malhas patch - TPZPatchMesh;
2. Vetores contendo o erro calculado;
3. Valor do erro limite para que um elemento seja adaptado e
4. Listas de padrões de refinamento.

A serialização das malhas *patch* foi um processo custoso, uma vez que uma malha *patch* contém uma malha computacional e, para efeitos de cálculo, é necessário transmitir também a malha geométrica associada, o que implicou na necessidade de serialização de boa parte das classes do ambiente PZ dos módulos de malha e de definição de formulações variacionais, destacando-se os

elementos (geométricos e computacionais) e os materiais que foram utilizados durante as etapas de testes e validação. A serialização das malhas do PZ foi iniciado no trabalho de mestrado de Santos [50], o qual a serialização para escrita e leitura de disco foi implementada.

A serialização de valores e vetores está implementada no ambiente OOPar e assim a não foi necessária implementar métodos de serialização para o vetor de erros e o valor do erro limite.

Para a transmissão da lista de padrões de refinamento foi necessário serializar as classes *TPZRefineData* e *TPZRefineDataUniform*. Como a lista de padrões de refinamento de global é considerado um dado distribuído do ambiente OOPar, houve a necessidade de se implementar uma classe container, cujo atributo é uma lista de ponteiros para padrões de refinamento, de modo que essa classe pudesse ser derivada da classe *TPZSaveable*, que implementa a interface de dados distribuídos no OOPar. Essa classe *TPZListRefinementPattern* é descrita sucintamente abaixo.

Alterações na Classe TPZAdaptiveProcess

Classe *TPZListRefinementPattern*

Essa classe foi definida de modo a implementar um dado distribuído do processo de adaptação de malhas, sendo a lista global de padrões de refinamento um dado distribuído. Destaca-se que a lista para uma malha patch poderia ser serializada simplesmente como um vetor de objetos, ou seja, primeiramente se transmite o tamanho da lista e na seqüência os elementos.

Entretanto, a lista global de padrões é uma lista que será acessada por todas as tarefas de cálculo, para inserir nessa os padrões calculados localmente. Há a necessidade de que o acesso a tal dado seja controlado, de modo a evitar acesso simultâneo à escrita. Tal gerenciamento é feito sobre os dados distribuídos do OOPar, entretanto, para tal há a necessidade de que esse dado implemente a interface determinada pela classe *TPZSaveable*.

Atributos

O único atributo da classe é uma lista para ponteiros de objetos do tipo TPZRefineData: *list< TPZAutoPointer<TPZRefineData> > fRefineDataList*. Note que o ponteiro é utilizado pelo fato de que o padrão de refinamento, em princípio, pode ser do tipo uniforme ou direcional.

Principais métodos e interface pública

Os principais métodos implementados são os métodos virtuais da classe *TPZSaveable*:

- *virtual void Read(TPZStream &buf, void *context)*: restaura um objeto a partir de um vetor de dados;
- *virtual void Write(TPZStream &buf, int withclassid)*: serializa os dados da classe para um vetor de dados;

- *virtual int ClassId() const*: retorna o identificador, que deve ser único, da classe.

Além desses métodos, também são implementados métodos para gerenciamento e controle de acesso à lista de padrões de refinamento:

- *list< TPZAutoPointer<TPZRefineData> > &getList()*: retorna a lista de padrões de refinamento;
- *void Print(ostream &out)*: imprime a estrutura de dados no dispositivo indicado por *out*;
- *void insert(TPZListRefinementPattern &listToApp)*: insere um padrão de refinamento na lista de padrões;
- *void insert(list< TPZAutoPointer<TPZRefineData> > &listToApp)*: insere uma lista de padrões de refinamento na lista do objeto.

4.3.3 *Tarefas*

Conforme mencionado na descrição do ambiente OOPar, a idéia de paralelização de um código serial através do OOPar é definir tarefas que façam chamadas para funções já implementadas e validadas no código serial. A idéia aqui implementada foi exatamente essa, onde definiu-se uma tarefa cujo atributo é um objeto do tipo *TPZPatchErrorEstimator* e o que a tarefa faz é realizar as mesmas chamadas feitas sobre esse objeto para obter o erro para os elementos de referência da malha *patch* associada e, posteriormente, tendo o valor do erro limite para adaptação realizar a chamada para o método que analisa os padrões de refinamento para cada elemento de referência.

Foram implementadas quatro tarefas, duas de cálculo sendo uma para o cálculo do erro e outra para a análise dos padrões de refinamento e duas tarefas para inserção dos dados calculados na malha global (*assemble*).

As tarefas de cálculo foram criadas nos pontos do código da classe *TPZAdaptiveProcess* que desempenhavam a mesma função no código serial, sendo a definição de qual código utilizar, serial ou paralelo, definido durante a compilação, em função da definição ou não da variável OOPAR.

Após a criação das tarefas de cálculo foram criados pontos de sincronismo, por meio de tarefas do tipo *OOPWaitTask*, para garantir que todas malhas *patch* distribuídas tenham contribuído no objeto global correspondente (mapa de erros ou lista de padrões de refinamento).

Assim, a seqüência de processamento paralelo é:

1. *TPZAdaptiveProcess*: cria as tarefas de cálculo
2. Tarefa de cálculo: tendo acesso aos dados de dependência processa a malha *patch* e ao final gera uma tarefa de "assemblagem";

3. Todas as tarefas de assemblagem acessam o dado global distribuído para inserir seus dados calculados localmente. Esse acesso segue a ordem de chegada, ou seja uma fila do tipo "*fifo*".

Na seqüência são descritas as tarefas, na seqüência que essas são processadas. Destaca-se que, na descrição de uma classe que implementa uma tarefa, além da descrição dos atributos da classe também há a necessidade de descrição das dependências da tarefa, uma vez que a definição do acesso a um dado em determinada versão é que controla o fluxo em que as tarefas são executadas no OOPar.

Classe *OOPTaskErrorEstimation*

Essa classe implementa uma tarefa OOPar cuja função é realizar os cálculos sobre uma malha clone de um *patch*.

A implementação de uma tarefa OOPar implica que os objetos dessa classe devem ser serializados e, por conseqüência, todos seus atributos também devem ser.

Atributos

Essa classe tem apenas um atributo que é o identificador do dado distribuído relativo ao vetor de erros global:

- *OOPIdVersion fGlobalErrorIdVersion*: Armazena o identificador do vetor de erros global.

Dependências

Para o cálculo da estimativa de erro, a única dependência é o acesso à escrita ao objeto *TPZPatchErrorEstimator* correspondente à malha *patch* a analisar. A versão inicial do dado é a requerida, uma vez que o processamento será realizado durante a execução dessa tarefa. A linha de obtenção da dependência é mostrada abaixo:

- *TPZPatchErrorEstimator *patchError = dynamic_cast<TPZPatchErrorEstimator *>(fDependRequest.ObjectPtr(0));*

Principais métodos e interface pública

A interface a ser descrita em qualquer tarefa é aquela implementada no método Execute, que é o método chamado pelo gerenciador de tarefa do OOPar quando todas as dependências para a execução da tarefa estão satisfeitas. O algoritmo implementado no método Execute é mostrado no Algoritmo 3.

Algorithm 3 *OOPReturnType OOPTaskErrorEstimation::Execute()*

1. *TPZPatchErrorEstimator *patchError = dynamic_cast< TPZPatchErrorEstimator * >(fDependRequest.ObjectPtr(0))*;
   Obtém o dado de dependência relativo ao objeto da classe *TPZPatchErrorEstimator*. Esse objeto tem como um de seus atributos a malha *patch*;

2. *if(!patchError) return EFail;*
   verifica a integridade do dado de dependência e, em caso de falha, retorna o indicador de falha na execução da tarefa

3. *patchError->evalError*();
   faz a chamada para o método que processa o cálculo da estimativa de erro para a malha *patch*

4. *multimap< double,int >errorToGlIdx;*
   patchError->getGlobalError(errorToGlIdx);
   Obtém o mapa de erros estimado para cada índice de elemento da malha original

5. *OOPTaskErrorAssemble *assemble = new OOPTaskErrorAssemble(0)*;
   Cria uma tarefa de assemblagem do erro a ser executada no processador 0

6. *TPZVec<REAL> &errorVec = assemble->ErrorVec();*
   TPZVec<int> &indexVec = assemble->GlobalIndexesVec();
   Obtém os vetores da tarefa de assemblagem que definem os índices de elementos e o vetor de erros associado a cada um desses índices. Transfere-se os dados de erro estimado do mapa obtido em 4 para esses vetores

7. *OOPAccessTag globalErrorVecDep(fGlobalErrorIdVersion.Id(), EWriteAccess, fGlobalErrorIdVersion.Version(), 0);*
   assemble->AddDependentData(globalErrorVecDep);
   Cria a dependência para a tarefa de assemblagem de acesso ao vetor de erros global (*fGlobalErrorIdVersion.Id()*) na versão indicada (*fGlobalErrorIdVersion.Version()*)

8. *assemble->Submit();*
   Submete a tarefa ao gerenciador de tarefas do OOPar

9. *OOPTask::Execute();*
   Método que incrementa a versão dos dados dependentes com acesso à escrita

10. *return ESuccess;*
    retorna o indicador de que a tarefa foi processada corretamente

Classe *OOPTaskErrorAssemble*

Essa classe implementa uma tarefa OOPar cuja função é realizar a inserção de valores de erro estimado em uma malha *patch* em um vetor de erros estimados global.

A vantagem da abordagem proposta é que essa tarefa só é criada após o cálculo do erro ter sido realizado. Como a tarefa requer acesso a escrita sobre o vetor de erros global, o fato dessa solicitação ser feita quando os valores calculados já estão disponíveis reduz a possibilidade de atrasos devido a espera por um dado ainda não disponível.

Atributos

Os atributos dessa classe corresponde a dois vetores que representam o mapa correlacionando erro e índice de elemento, conforme descrito abaixo:

- *TPZVec <int> fGlobalIndex*: Vetor com os índices dos elementos onde os valores devem ser inseridos.
- *TPZVec <REAL> fError*: Vetor com os valores de erro calculados para ser inseridos no vetor global.

Dependências

A dependência dessa tarefa é o acesso a escrita do vetor de erros global em qualquer versão: *OOPVector<REAL> *globalError = dynamic_cast< OOPVector<REAL> * >(fDependRequest.ObjectPtr(0));*

Principais métodos e interface pública

A interface pública dessa tarefa implementa:

- métodos para serialização: conforme já descrito para a tarefa anterior:
  - *virtual void Read(TPZStream &buf, void *context);*
  - *virtual void Write(TPZStream &buf, int withclassid);*
  - *virtual int ClassId() const;*
- o acesso aos seus atributos:
  - *TPZVec<REAL> &ErrorVec()*: acesso ao vetor contendo os valores de erros estimados;
  - *TPZVec<int> & GlobalIndexesVec()*: acesso ao vetor contendo os índices dos elementos aos quais correspondem os erros estimados;
  - *void Print(ostream &out)*: imprime a estrutura de dados no dispositivo indicado por *out*;

Algorithm 4 *OOPTaskErrorAssemble::Execute()*

1. *OOPVector<REAL> *globalError = dynamic_ cast< OOPVector<REAL> * >(fDependRequest.ObjectPtr(0));*
   Obtém o dado de dependência, correspondendo a um vetor global de erros, cuja dimensão é o número de elementos da malha
2. Percorre-se os vetores locais da tarefa e para cada posição dos vetores:
   (a) Obtém-se o índice do elemento à contribuir;
   (b) Obtém-se o valor a contribuir;
   (c) Define-se no vetor global para o índice obtido acima o valor do erro para essa posição
3. *OOPTask::Execute();*
   Método que incrementa a versão dos dados dependentes com acesso à escrita
4. *return ESuccess;*
   retorna o indicador de que a tarefa foi processada corretamente

- OOPMReturnType Execute(): chamada realizada pelo gerenciador de tarefas do OOPar quando todas as dependências da tarefa estão satisfeitas. O algoritmo 3 mostra a seqüência de operações aqui realizada.

Classe *OOPTaskBuildRefinement*

Essa tarefa implementa a obtenção dos padrões de refinamento para os elementos de referência de uma determinada malha *patch*. Essas tarefas são criadas após a determinação do valor de erro limite para que um elemento seja adaptado.

Da mesma forma que para as tarefas relacionadas à estimativa do erro, aqui se analisam os padrões de refinamento, gerando-se uma lista local de padrões de refinamento. A inserção dessa lista local em uma lista global é feita por outra tarefa de assemblagem, a qual é descrita na seqüência.

Atributos

Os atributos dessa classe corresponde a dois vetores que representam o mapa correlacionando erro e índice de elemento, conforme descrito abaixo:

- *OOPIdVersion fGlobalRefinementIdVersion*: identificador da lista de padrões de refinamento global, na qual a lista aqui gerada deverá ser "assemblada";
- *REAL fThreshold*: valor do erro limite para que um elemento tenha seu padrão de refinamento analisado.

Dependências

A dependência dessa tarefa é o acesso a escrita ao objeto *TPZPatchErrorEstimator* correspondente à malha *patch* a analisar: *TPZPatchErrorEstimator *patchError = dynamic_ cast< TPZPatchErrorEstimator * >(fDependRequest.ObjectPtr(0));*

A versão requerida é a segunda versão do dado, uma vez que após o cálculo da estimativa do erro a versão desse dado é incrementada.

Principais métodos e interface pública

A interface pública dessa tarefa implementa:

- métodos para serialização: conforme já descrito para a tarefa anterior:
  - *virtual void Read(TPZStream &buf, void *context);*
  - *virtual void Write(TPZStream &buf, int withclassid);*
  - *virtual int ClassId() const;*
- *OOPMReturnType Execute()*: chamada realizada pelo gerenciador de tarefas do OOPar quando todas as dependências da tarefa estão satisfeitas. O Algoritmo 5 mostra a seqüência de operações aqui realizada.

Classe *OOPTaskRefinementAssemble*

Essa classe implementa uma tarefa OOPar cuja função é realizar a inserção da lista de objetos de padrão de refinamento obtidos em uma malha *patch* em uma lista global.

Conforme descrito para a tarefa de assemblagem do erro, a vantagem da abordagem proposta é que essa tarefa só é criada após o cálculo de uma lista local ter sido realizado.

Atributos

Os atributos dessa classe corresponde a um objeto lista de padrões de refinamento, conforme descrito abaixo:

- *TPZListRefinementPattern fLocalRefinements*: lista de objetos padrão de refinamento de uma malha *patch.*

Dependências

A dependência dessa tarefa é o acesso a escrita na lista de objetos de padrão de refinamento global em qualquer versão: *TPZListRefinementPattern *globalList = dynamic_ cast<TPZListRefinementPattern *>(fDependRequest.ObjectPtr(0));*

Algorithm 5 *OOPTaskBuildRefinement::Execute()*

1. *TPZPatchErrorEstimator *patchError = dynamic_cast< TPZPatchErrorEstimator * >(fDependRequest.ObjectPtr(0))*;
   Obtém o dado de dependência correspondendo à malha *patch* a analisar

2. *if(!patchError) return EFail;*
   Verifica a consistência do dado, retornando estado de falha caso o dado não tenha sido corretamente recebido

3. *list< TPZAutoPointer <pzadapt::TPZRefineData> > patchRefData;*
   Cria uma lista de objetos de padrão de refinamento local

4. *patchError->BuildRefinementPatterns(fThreshold, patchRefData);*
   realiza a chamada para o método da classe *TPZPatchErrorEstimator* que analisa os padrões de refinamento para os elementos de referência da malha *patch* associada

5. *OOPTaskRefinementAssemble *assemble = new OOPTaskRefinementAssemble(0);*
   Cria uma tarefa de "assemblagem" da lista de padrões de refinamento local na lista global

6. *TPZListRefinementPattern &refList = assemble->RefinementList();*
   refList.insert(patchRefData);
   Insere a lista local de objetos de padrão de refinamento no atributo da tarefa

7. O*OPAccessTag globalRefinVecDep(fGlobalRefinementIdVersion.Id(), EWriteAccess, fGlobalRefinementIdVersion.Version(), 0);*
   assemble->AddDependentData(globalRefinVecDep);
   Define a dependência da tarefa como sendo o acesso a escrita na lista de padrões de refinamento global (*fGlobalRefinementIdVersion.Id()*) na versão inicial dessa (*fGlobalRefinementIdVersion.Version()*)

8. *assemble->Submit();*
   Submete a tarefa ao gerenciador de tarefas

9. *OOPTask::Execute();*
   Método que incrementa a versão dos dados dependentes com acesso à escrita

10. *return ESuccess;*
    retorna o indicador de que a tarefa foi processada corretamente

Algorithm 6 *OOPTaskRefinementAssemble::Execute()*

1. *TPZListRefinementPattern *globalList = dynamic_cast< TPZListRefinementPattern *>(fDependRequest.ObjectPtr(0))*
   Obtém o dado de dependência, correspondendo a lista global de objetos padrão de de refinamento
2. *globalList->insert(fLocalRefinements);*
   Insere os objetos da lista local da tarefa na lista global
3. *OOPTask::Execute();*
   Método que incrementa a versão dos dados dependentes com acesso à escrita
4. *return ESuccess;*
   retorna o indicador de que a tarefa foi processada corretamente

Principais métodos e interface pública

A interface pública dessa tarefa implementa:

- métodos para serialização: conforme já descrito para a tarefa anterior:
  - *virtual void Read(TPZStream &buf, void *context);*
  - *virtual void Write(TPZStream &buf, int withclassid);*
  - *virtual int ClassId() const;*
- o acesso aos seus atributos:
  - *TPZListRefinementPattern & RefinementList()*: acesso à lista de objetos padrão de refinamento da tarefa;
  - *void Print(ostream &out)*: imprime a estrutura de dados no dispositivo indicado por *out*;
- *OOPMReturnType Execute()*: chamada realizada pelo gerenciador de tarefas do OOPar quando todas as dependências da tarefa estão satisfeitas. O algoritmo 6 mostra a seqüência de operações aqui realizada.

Capítulo 5

Casos de validação e testes

Para a validação do sistema implementado foram desenvolvidos testes de validação para qualificar os resultados gerados. Os testes são feitos para dois tipos de formulações: equação de elasticidade e equação de Laplace com uma variável de estado.

Os problemas bi-dimensionais são validados a partir de duas malhas iniciais cada: uma malha composta exclusivamente por elementos quadrilaterais adulterais outra composta exclusivamente por elementos triangulares.

Primeiramente, procura-se mostrar a efetividade do estimador de erros. Para tal são utilizados problemas com solução analítica conhecida. Desse modo torna-se possível a obtenção do índice de efetividade do estimador de erros para os problemas selecionados.

O passo seguinte visa qualificar a implementação paralela. Para tal, analisa-se o tempo de resolução de problemas selecionados em diversas configurações, tais como: um único processador, multithreading variando o número de threads e memória distribuída, variando o número de processadores.

5.1 Testes internos para certificação dos procedimentos

Durante cada etapa de desenvolvimento, foram definidos testes internos para certificar os resultados obtidos. Para tal foram inseridos logs de testes ao longo de todo o código. Resumidamente, quando o código é executado com as diretivas de depuração ativadas, são executados os seguintes testes:

1. Verificação da tranferência correta da solução da malha original para a malha patch: isso é feito por meio de dois procedimentos:

   (a) cálculo da norma da diferença entre a solução da malha patch e a solução da malha original. Tal teste garante que a cópia foi feita corretamente;

   (b) comparação da solução proveniente da malha original com uma solução proveniente da execução de uma análise com malha patch. Caso algum dado da malha não esteja

correto, tal como a transferência de uma condição de contorno, a solução obtida do processamento será diferente daquela definida pela malha original;

2. Verificação da simetria dos erros estimados e padrões de refinamento calculados para problemas simétricos;

3. Comparação dos resultados obtidos, erros e padrões de refinamento, entre problemas seriais e paralelos.

Tais verificações mostraram-se muito eficazes na validação do código.

5.2 Avaliação do estimador de erros: convergência e efetividade

5.2.1 Equação de Laplace

Problema de Laplace aplicado em Placa em L bidimensional

Como primeiro teste de validação foi utilizado o modelo apresentado na Figura 5.1, reproduzida de [14]. O problema é dado por:

$$\Delta u = 0 \; x \in \Omega \tag{5.1}$$

As condições de contorno aplicadas são tais que $\frac{\partial u}{\partial n}$ corresponde a solução exata, dada em termos de coordenadas polares (r, θ):

$$u(r,\theta) = \sqrt[3]{r} \sin\left(\frac{1}{3}\left(\theta + \frac{\pi}{2}\right)\right) \tag{5.2}$$

Esta solução analítica possibilita o cálculo do erro real em cada elemento e desta forma verificar o índice de efetividade do estimador de erros implementado.

Convergência e índice de efetividade do estimador de erros

O comportamento dos gráficos de convergência do estimador de erros, bem como do erro real para as duas malhas analisadas foi similar, conforme pode ser visto nas figuras representativas de cada caso. Podem ser destacadas três seções típicas: a primeira com taxa de convergência exponencial, conforme esperado para um método *hp* adaptativo. Na seqüência, tem-se um trecho linear. Aqui, foi observado que na região de singularidade da solução, atingiu-se a máxima ordem p disponível, passando então o método a convergir em uma taxa linear, condizente com o refinamento h. No terceiro trecho, a taxa de convergência começa a divergir. Possivelmente isso ocorre por erros de

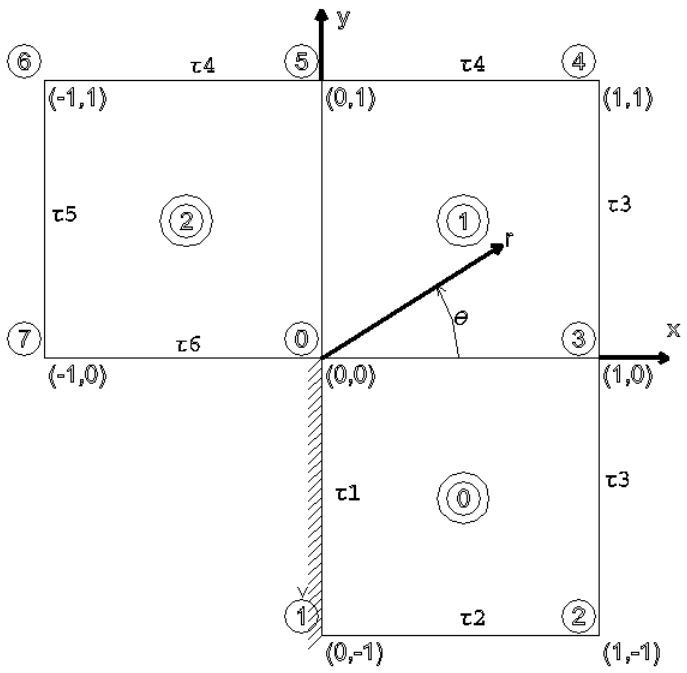


Figura 5.1: Problema modelo - domínio em L

truncamento, uma vez que a norma de erro correspondente a 10^{-4}, implica que o somatório das contribuições para o erro estão na ordem de no máximo 10^{-8}.

Malha de elementos quadrilateral

A Figura 5.2 apresenta o gráfico de convergência para o problema modelo, utilizando elementos quadrilaterais. Os três trechos citados anteriormente podem ser delimitados por volta de 1000 equações e 7500 equações.

O índice de efetividade é apresentado na Figura 5.3. O comportamento do índice de efetividade fica em torno de 80% até o número de equações atingir 7500 aproximadamente. Quando da mesma forma que para o caso acima, ocorre uma anomalia no índice de efetividade.

A malha inicial é apresentada na Figura 5.4(a), sendo na seqüência mostradas as malhas geradas para os passos de refinamento indicados. A Figura 5.5 mostra uma seqüência de detalhes do canto do L.

Malha de elementos triangulares

A Figura 5.6 apresenta o gráfico de convergência para o problema modelo, utilizando elementos quadrilaterais. Os três trechos citados anteriormente podem ser delimitados por volta de 1000 equações e 7500 equações.

O índice de efetividade é apresentado na Figura 5.7. O comportamento do índice de efetividade fica em torno de 90% até o número de equações atingir 7500 aproximadamente.

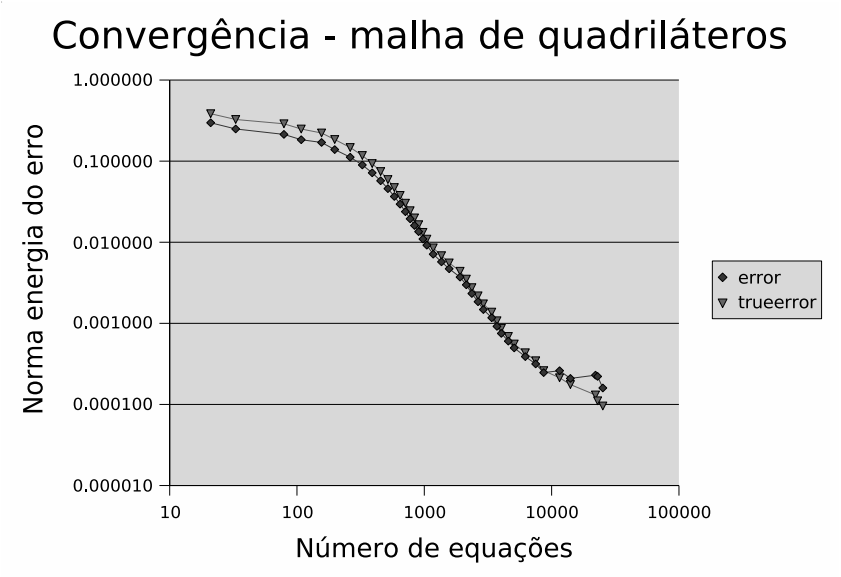


Figura 5.2: Convergência para a malha de quadriláteros

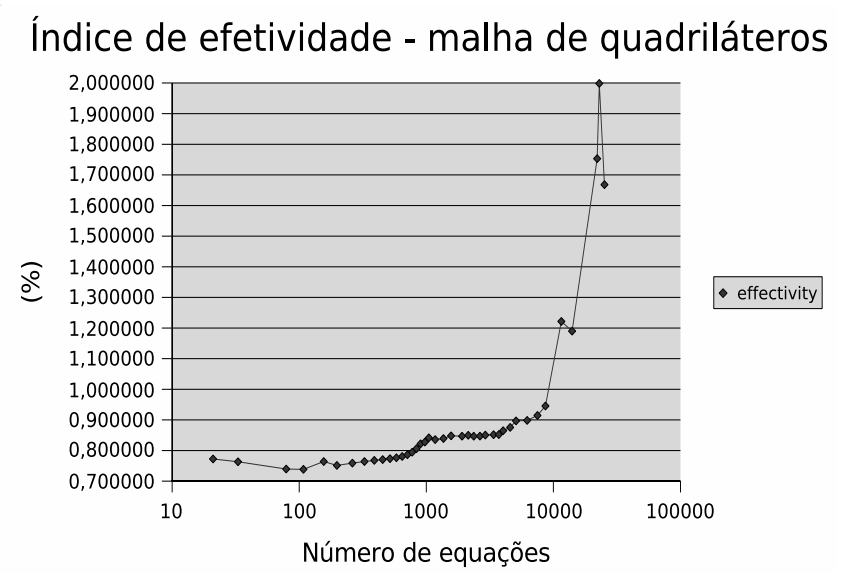


Figura 5.3: Índice de efetividade para o estimador de erros para a malha de quadriláteros

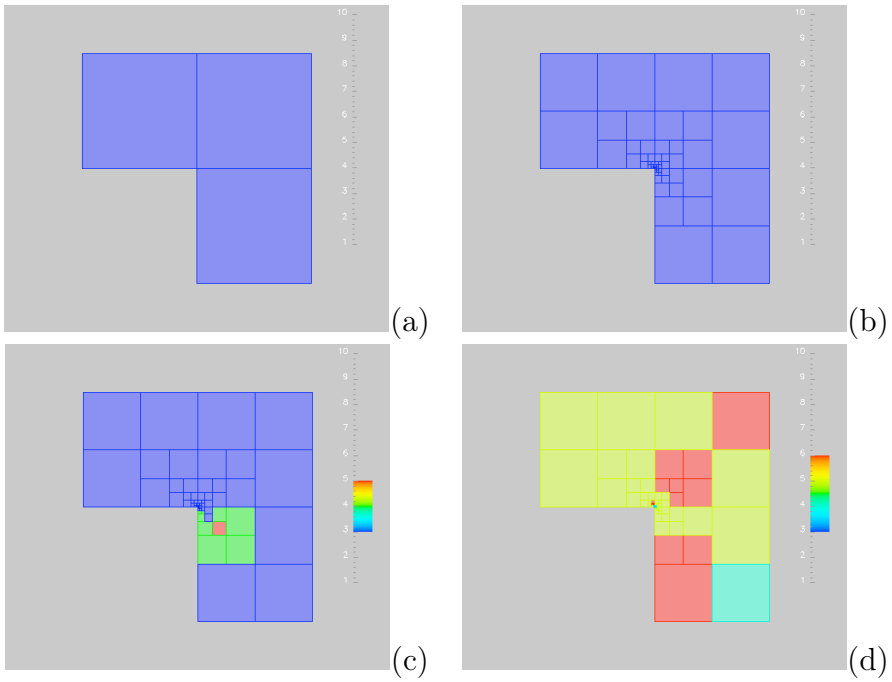


Figura 5.4: Malha quadrilateral - seqüência de malhas geradas: (a) malha original, (b) 10 passos de refinamento (d) 20 passos de refinamento (d) 30 passos de refinamento

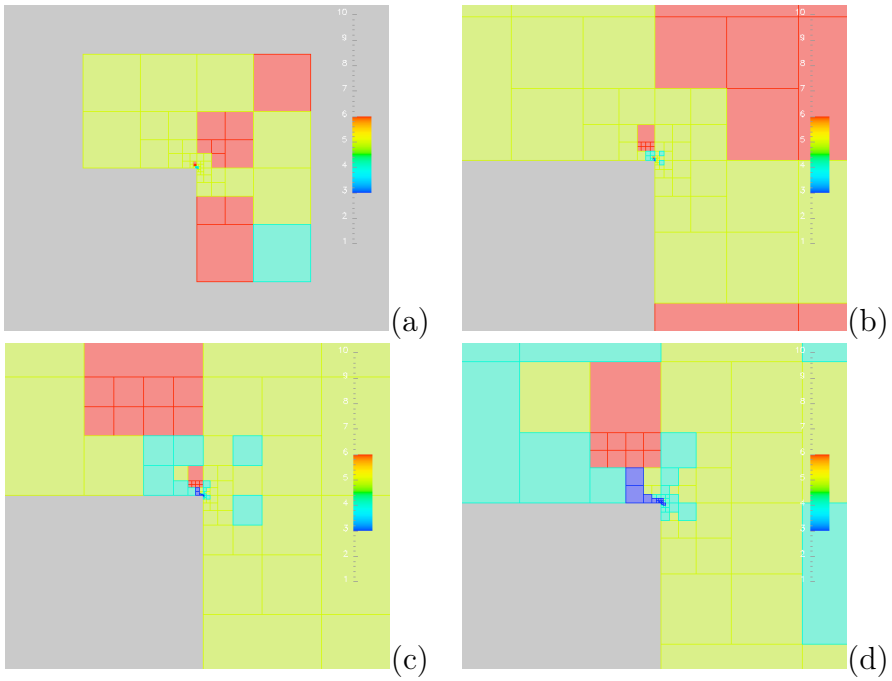


Figura 5.5: Malha quadrilateral - seqüência de detalhes do canto do L

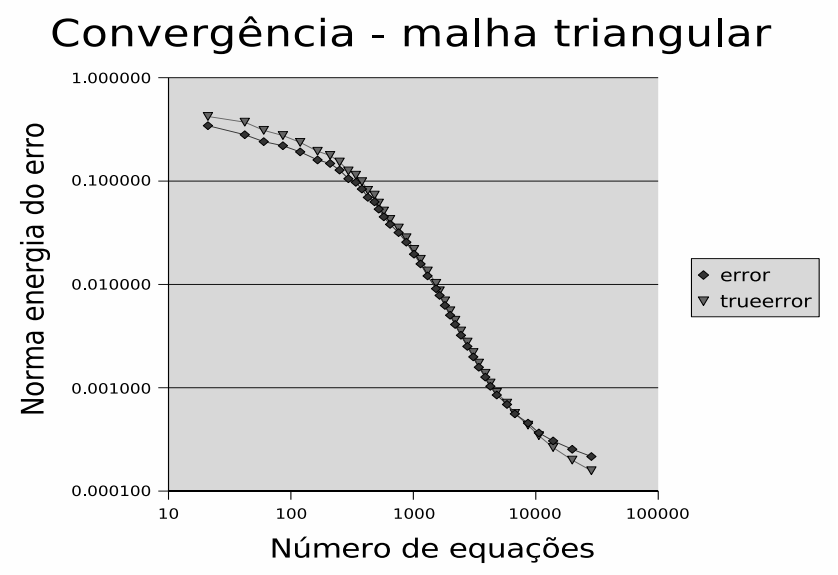


Figura 5.6: Convergência para a malha de quadriláteros

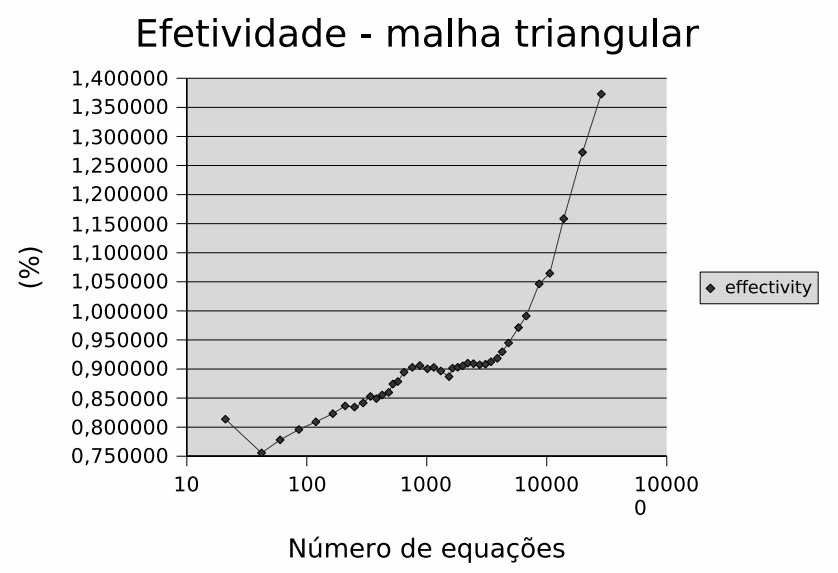


Figura 5.7: Índice de efetividade para o estimador de erros para a malha de quadriláteros

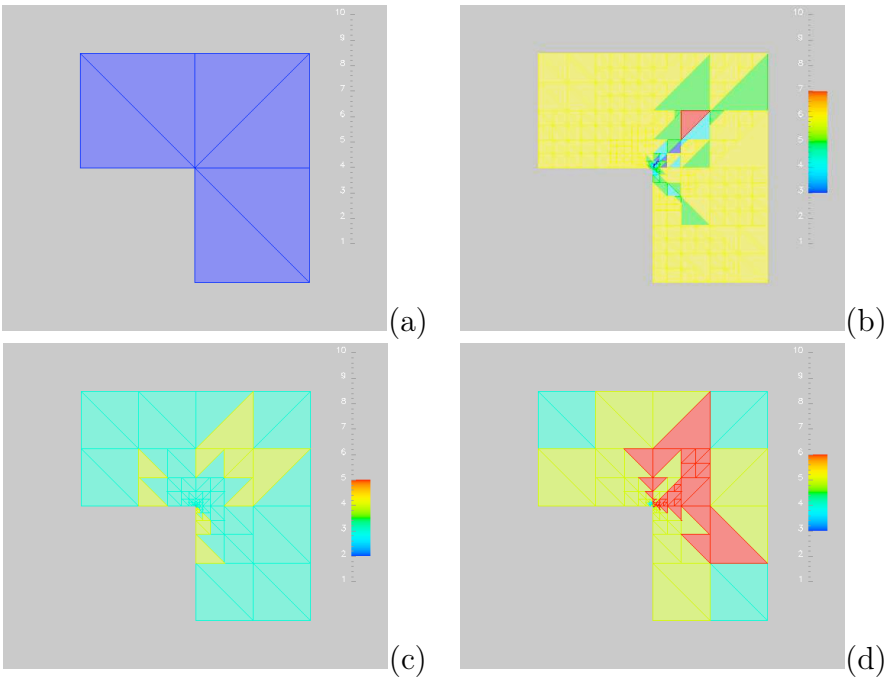


Figura 5.8: Malha quadrilateral - seqüência de malhas geradas: (a) malha original, (b) 10 passos de refinamento (d) 20 passos de refinamento (d) 30 passos de refinamento

A malha inicial é apresentada na Figura 5.8(a), sendo na seqüência mostradas as malhas geradas para os passos de refinamento indicados. A Figura 5.9 mostra uma seqüência de detalhes do canto do L.

5.3 Implementação paralela

Os teste de avaliação de aceleração devido à utilização de processamento paralelo com memória distribuída foram realizados em um cluster composto por nós biprocessados, processador AMD-Opteron 242, 1.6GHz e 2GB RAM, nós interconectados através de uma rede Gigabit Ethernet.

A biblioteca MPI utilizada é a mpich-1.0.4p1, compilada com gcc versão 3.4.2.

O acesso aos nós é controlado por um sistema de filas, o que garante exclusividade para o processo em execução.

Os testes foram feitos para as seguintes configurações:

- dois, quatro e oito nós com um thread por nó e
- dois, quatro e oito nós com dois threads por nó.
- dois, quatro e oito nós com quatro threads por nó.

Inicialmente, foi garantido que a solução obtida por meio de processamento paralelo fosse igual àquela obtida por meio de processamento serial. Tal validação consistiu nos seguintes passos:

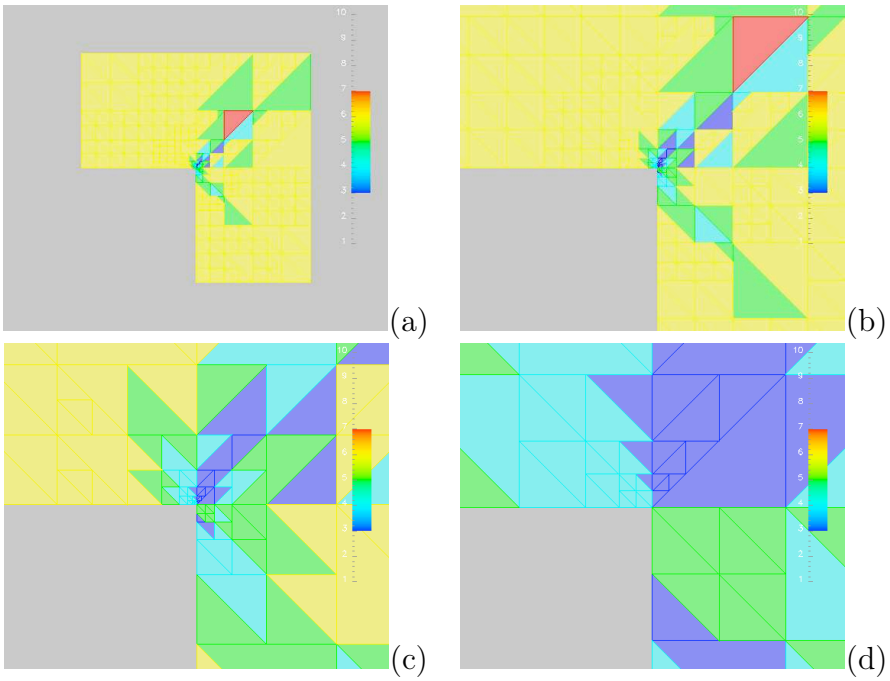


Figura 5.9: Malha quadrilateral - seqüência de detalhes do canto do L

1. Garantir que a estrutura de dados das malhas patch distribuídas correspondiam à estrutura de dados dessa mesma malha quando processada em modo serial;

2. Garantir que o erro calculado remotamente fosse igual ao erro calculado em modo serial;

3. Garantir que os elementos selecionados para divisão correspondessem a aqueles selecionados em modo serial;

4. Comparação dos padrões de refinamento nos processos serial e paralelo

Essa validação foi feita utilizando multiplos processos na mesma máquina, um computador similar aos nós descritos acima, más quad-processado e com 8GB de memória RAM.

Após os testes de validação foram gerados os scripts para submissão dos processos no sistema de filas descritos acima.

O primeiro teste feito foi para o problema de validação laplaciano utilizando malhas quadrilaterais. A configuração utilizada foi de apenas um thread por processo MPI, ou seja um thread para cada computador remoto. Os resultados são apresentados na tabela abaixo:

Tabela 5.1: Tempo (ms) de processamento da malha LShape formada por quadriláteros para dadas configurações

N. processadores	Threads/processador	10 iter.	20 iter.	30 iter.
1	1	24010	154250	710580
2	2	14360	89360	396570
4	2	14250	88870	390660
8	2	14280	88280	389490
2	1	14760	94480	415170
4	1	14220	89590	395720
8	1	14380	89710	401650

Os resultados mostram que há um ganho de velocidade quando da utilização de processamento paralelo, mas não há aceleração em função do número de processadores envolvidos. Tal observação se faz para processos remotos utilizando apenas um thread, ou dois threads.

Esse comportamento não era o esperado, o que motivou a necessidade de depuração e avaliação do procedimento paralelo, de modo a verificar se havia um problema observável, tal como gargalo de comunicação, ou ainda pontos de sincronismo mal locados.

A avaliação aqui feita utilizou-se logs e também da biblioteca mpe, disponibilizada juntamente com a biblioteca mpich2. Tal biblioteca gera logs interpretáveis pelo programa jumpshot também obtido junto com a biblioteca MPI.

A Figura 5.10 mostra a tela inicial do jumpshot para o caso de utilização de 8 processadores com um thread por processador.

Os blocos representam tarefas sendo executadas, sendo o início da tarefa demarcado por um círculos sobre barras. Cada conjunto vertical de blocos corresponde a um passo de adaptação. Como os blocos estão sempre alinhados do lado esquerdo, já se pode descartar a possibilidade de alguma demora na submissão das tarefas.

O problema encontrado pode ser visualizado melhor em um detalhe de um passo de adaptação, que é representativo, mostrado na Figura 5.11. Note que o tempo está sendo limitado pelo tempo de execução das tarefas do quarto processador. Outro ponto a ressaltar é o tamanho da tarefa do sexto processador.

Apenas a título de averiguação, foi realizada uma nova análise, com apenas dois processadores, más utilizando dois threads por processo MPI. Por cada nó ser bi-processado, teríamos uma tarefa por processador. Os resultados obtidos são mostrados na Figura 5.12.

Análise dos resultados

Os resultados encontrados não foram aqueles esperados e as seguintes conclusões podem ser obtidas dos resultados acima mostrados:

1. A carga de processamento está desbalanceada;

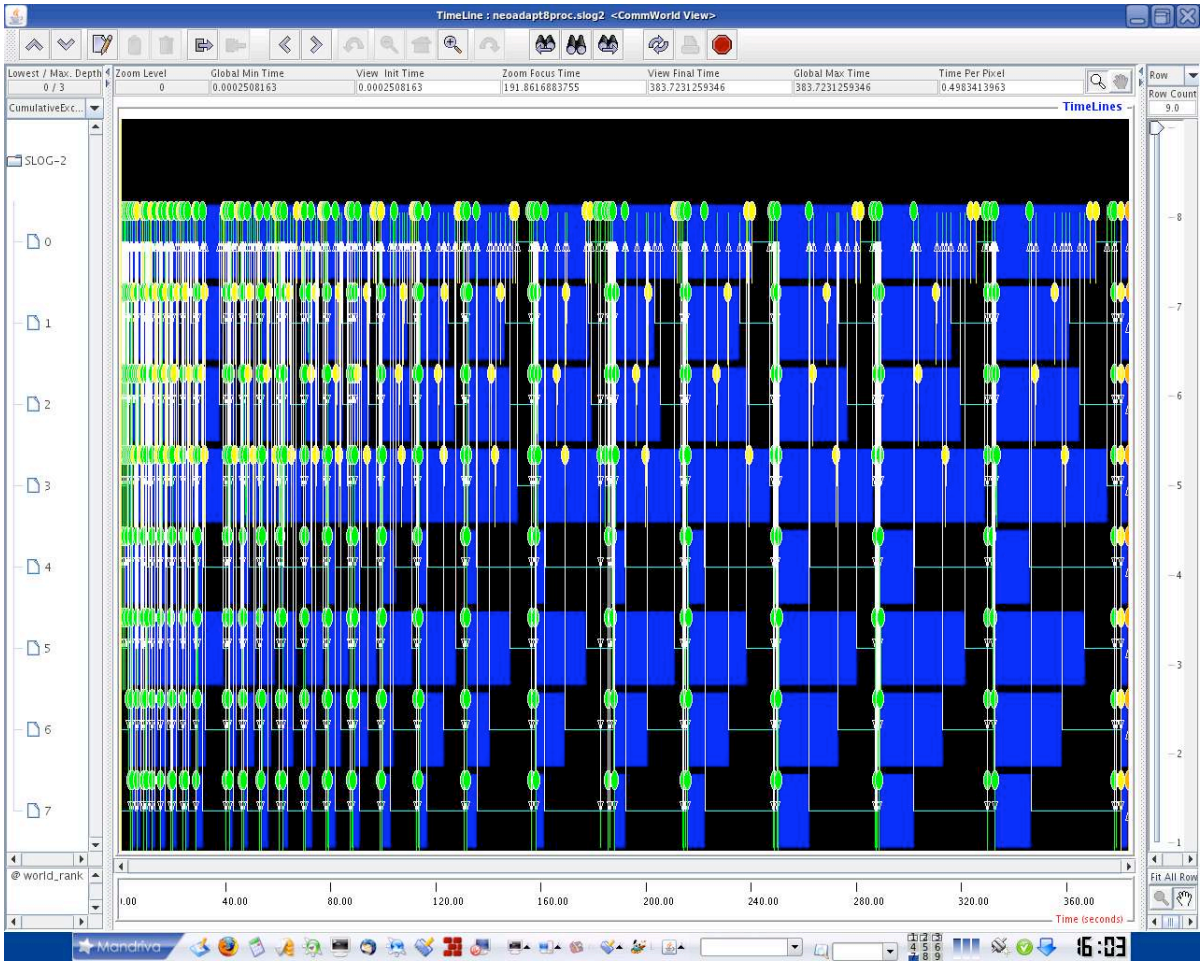


Figura 5.10: Análise da implementação paralela - 8 processos MPI com 1 thread por processo. Visão geral

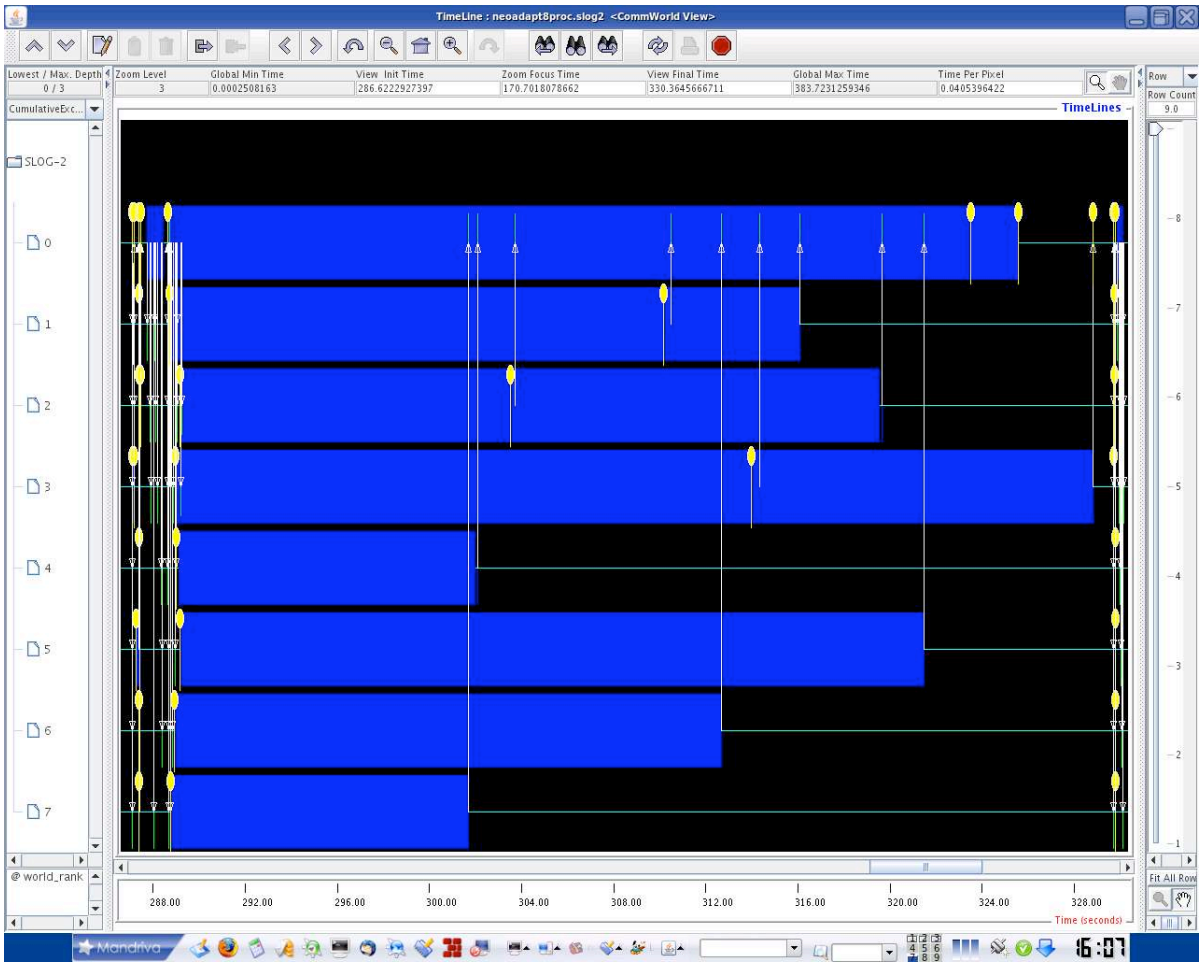


Figura 5.11: Análise de um passo de adaptação - 8 processos MPI com 1 thread por processo. Detalhe

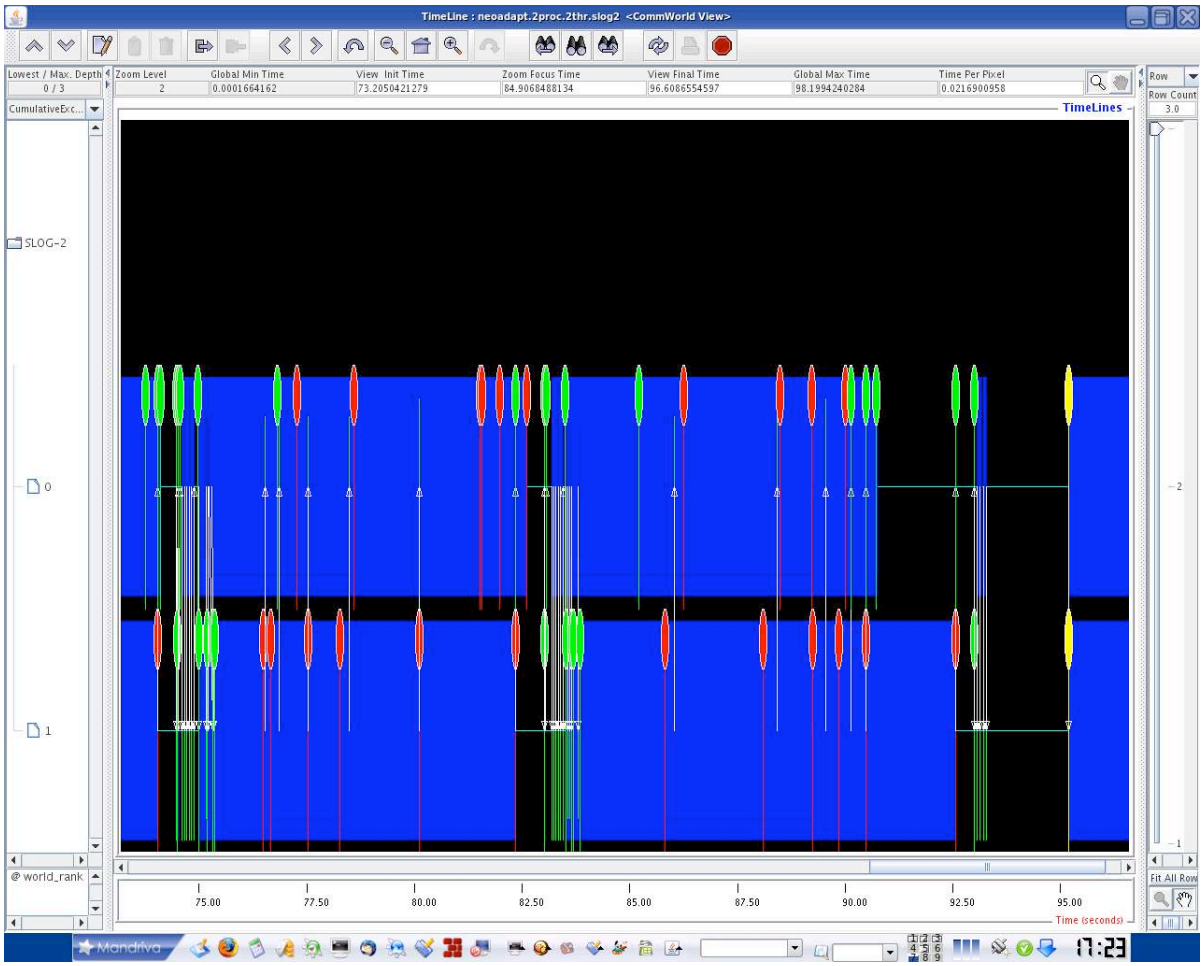


Figura 5.12: Análise de um passo de adaptação - 2 processos MPI com 2 threads por processo. Detalhe

2. A melhoria do balanceamento de carga não aumetará de modo sensível a aceleração do tempo de execução em paralelo uma vez que esse será restrito pelo tempo de processamento das maiores malhas patch.

A hipótese que temos então é de que o tamanho das malhas patch influencia as taxas de aceleração.

Para verificar tal hipótese foi implementada uma restrição na quantidade de níveis de elementos utilizados para construir uma malha patch. O teste foi feito utilizando as seguintes configurações paralelas: 1, 2 e 4 processos MPI, com 2 threads por processo. Os tempos obtidos são mostrados na tabela a seguir:

Tabela 5.2: Comparação de tempos variando-se a restrição das malhas patch. Tempo (minutos e segundos) para execução de 20 passos de adaptação, utilizando malha de quadriláteros

	serial	2 processadores	4 processadores
sem restrição de nível	02'17",260	01'17",800	01'17",610
4 níveis	03'09",780	01'47",120	01'47",060
2 níveis	03'18",150	01'51",840	01'41"920

O aumento do tempo geral em relação a análise sem restrições é algo esperado, uma vez que o número de malhas patch a ser processadas foi aumentado. É interessante notar que a restrição de 2 níveis na malha patch apresenta ainda alguma aceleração em relação aos casos de 4 níveis de restrição e sem restrições.

Em contrapartida a manutenção de taxas de aceleração, tem-se dois aspectos a considerar: primeiro é que o aumento do número de malhas patch aumenta a quantidade de dados comunicados. Isso pode ser entendido pelo fato de que aumentamos não apenas a quantidade de elementos de referência, más também a quantidade de elementos vizinhos.

O segundo aspecto é que a restrição no tamanho do patch influencia diretamente o índice de efetividade do estimador de erros, conforme descrito em [1] (seção 3.4). Nos casos mostrados na tabela acima observou-se que o índice de efetividade original variou da seguinte forma:

- sem restrições: 83,58%;
- com 4 níveis de restrição: 83,40%;
- com 2 níveis de restrição: 82,99%.

Capítulo 6

Conclusões e Propostas de Extensões

Foi desenvolvida uma metodologia para a implementação de um algoritmo auto-adaptativo em paralelo, utilizando cálculo de erro com base em *patches* de elementos. Tal abordagem facilitou a implementação dos algoritmos de estimativa do erro e da escolha do padrão de refinamento utilizando processamento paralelo.

O estimador de erros aqui utilizado é baseado na diferença entre dois espaços de aproximação hierárquicos, o que o torna independente da formulação fraca, tipo de elemento, tipo de método de resolução etc. Outros estimadores de erro podem ser acoplados ao sistema implementado, seguindo a interface aqui definida.

Com relação ao método para definição dos padrões de refinamento, a escolha baseia-se na definição dos padrões ótimos de refinamento para cada aresta do elemento em análise. Tal abordagem produziu bons resultados em termos de convergência. Essa análise foi feita para os padrões de refinamento h uniforme podendo ser utilizada para a definição de um algoritmo de escolha de padrões de refinamento h direcionais.

A implementação dos padrões de refinamento h direcionais, baseados em malhas exemplo, é um desenvolvimento da adaptatividade h. A escolha do "melhor padrão" h para esse tipo de refinamento deve levar em consideração a compatibilidade entre dois padrões adjacentes, o que insere uma complexidade extra na definição do padrão de refinamento. Nesse caso é necessário o conhecimento de toda a vizinhança do elemento para se definir o conjunto de padrões de refinamento compatíveis, dentro do qual se deve buscar aquele com precisão ótima.

Com relação aos resultados de performance do código paralelo, esses foram aquém das espectativas iniciais, tendo sido identificado o problema de falta de aceleração do processamento paralelo em função da dimensão dos *patches* de elementos. Foi observado que a taxa de aceleração é dependente do tempo de processamento da maior malha *patch*.

Formas alternativas de definição da malha *patch* podem ser investigados em trabalhos futuros. Uma limitação da quantidade de níveis de elementos é uma abordagem possível, devendo ser analisada em conjunto com a variação do índice de efetividade do estimador de erros. Outra

possibilidade é a utilização de padrões de refinamento que não gerem restrições. Tal abordagem é uma forma natural de restrição da quantidade de elementos em uma malha patch. Ambas abordagens levam a um aumento do número de malhas *patch* em contrapartida a redução do tamanho dessas. É provavel, com base em indícios mostrados pelos testes realizados, que tal abordagem leve a resultados de aceleração do cálculo paralelo melhores do que os aqui obtidos.

Definições de estratégias de uso de bibliotecas de padrões de refinamento também são um campo de pesquisa muito vasto. Nesse projeto foram mostrados exemplos efetivos de uso na geração de malhas de camada limite, o que motiva o desenvolvimento dessa linha de pesquisa.

Apêndices

Os textos abaixo foram reproduzidos de [49], por serem considerados importantes para a compreensão do trabalho.

Definição de malhas patch

Estimador de Erros Baseado em Subespaços

Quando se considera a utilização de subdomínios para obtenção de um estimador de erros, o primeiro fato que deve ser atentado é que não estará sendo levado em consideração o erro devido ao restante da malha, ou seja, estará se desprezando o erro por acoplamento. A parte desprezada do erro será maior quanto menor for o subespaço utilizado em relação ao espaço do domínio do problema.

De fato, no caso de utilização do menor subespaço possível, que seria a análise elemento a elemento, não há garantias teóricas para que o estimador venha a convergir para o erro real, ver [2].

Intuitivamente, quanto maior o subdomínio considerado, mais o resultado se aproximará do resultado apresentado na equação (2.25), onde a convergência é demonstrada.

Assim, um problema a ser estudado futuramente será a definição de qual dimensão de subespaço utilizar, de modo a compatibilizar a convergência do método com o custo computacional gerado pela resolução de um subespaço refinado.

Subespaço proposto - Definição do *patch* de elementos

Diversas metodologias podem ser adotadas na escolha, sendo aqui definido como critério básico a necessidade dos elementos de referência para a formação dos *patches* formarem uma partição da malha conforme já mencionado.

De maneira a facilitar o entendimento deste estudo, faz-se necessária a definição de alguns termos que serão utilizados de agora em diante:

- Elemento de Referência do *Patch*: consiste de um elemento geométrico da malha original, tendo como característica especial o fato de ser o maior elemento ancestral a ter vizinhos. O elemento de referência pode ou não ter sido refinado, tendo subelementos[1]. A forma de escolha dos elementos de referência será descrita adiante com maiores detalhes;

- *Patch*: é um conjunto de elementos formado pelo elemento de referência do *patch* ao centro, e por todos os elementos vizinhos ligados a ele diretamente, ou cujos nós sofram contribuição do elemento de referência ou de um de seus subelementos.

A forma de obtenção dos elementos de referência do *patch*, aqui adotada, é a seguinte:

[1]Os elementos no ambiente PZ tem duas representações: uma geométrica e outra computacional. Os elementos geométricos guardam referências para seus subelementos e também para o seu elemento pai. Estas características tornam possível a identificação de toda a "genealogia" de um elemento necessária às operações descritas. Maiores detalhes podem ser encontrados em [19].

Algorithm 7 Obtenção dos Elementos de Referência dos *Patches*

1. Definir uma lista vazia de Elementos de Referência do *Patch* - *Stack <Elementos> ElRef-Patch*
2. Para todos os elementos da malha original (malha a ser adaptada)
   (a) Criar uma lista de elementos ancestrais ao elemento, inserindo o elemento analisado na lista
   (b) Inserir todos os ancestrais do elemento na lista de ancestrais, de tal forma que o elemento analisado seja o primeiro da lista e o elemento ancestral mais antigo, proveniente da malha original, seja o último
   (c) Percorrer a lista de elementos ancestrais do final para o início e caso o elemento analisado tenha vizinhos com nível de refinamento igual ao seu, este elemento será definido como elemento de referência do *patch*, para o elemento analisado
   (d) Verificar se o elemento definido anteriormente já está na lista de elementos de referência do patch, em caso afirmativo passar para o próximo elemento da malha original, em caso negativo, inserir o elemento na lista

Tendo-se os elementos de referência para os *patches*, o passo seguinte é, para cada *patch*, identificar os elementos vizinhos ao seu elemento de referência, ou cujos nós contribuam para o elemento de referência do *patch* ou de algum de seus subelementos.

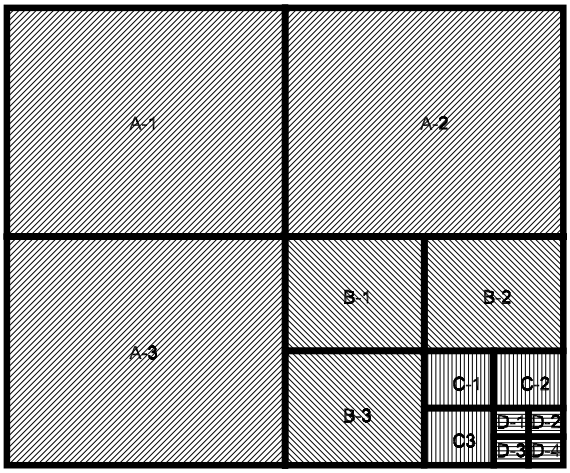
Como exemplo, analisando a malha mostrada na Figura (1), temos:

Nível	Denominação
0	A: $A_1, A_2, A_3, (A_4)$
1	$B : B_1, B_2, B_3, (B_4)$
2	$C : C_1, C_2, C_3, (C_4)$
3	$D : D_1, D_2, D_3, D_4$

Os elementos de nível "0" são os elementos da malha original, sendo um desses elementos A_4 refinado, dando origem aos elementos de nível "1" - elementos B_i, dentre os quais o elemento B_4 foi refinado , e assim sucessivamente. Caso fôssemos montar o "*patch*" de alguns elementos selecionados, teríamos:

Elemento analisado	Elemento de referência do *patch*	Componentes do *patch*
A_1	A_1	Todos elementos
B_2	A_4	Todos elementos
C_3	B_4	Subelementos de A_4
D_4	D_4	Subelementos de C_4

Figura 1: Ordem de Refinamento



Análise do padrão de refinamento baseado em análise de arestas

Classe TPZOneDRef

Essa classe tem por objetivo definir o melhor refinamento hp em um elemento linear. Esse elemento linear, no caso do ambiente PZ, inclui arestas de elementos bi-dimensionais e tri-dimensionais.

O processo consiste em: dada a solução em elementos uniformemente refinados hp, de ordem $n+1$, verificar qual o refinamento do elemento pai, de ordem n, que melhor aproxima a solução uniformemente refinada.

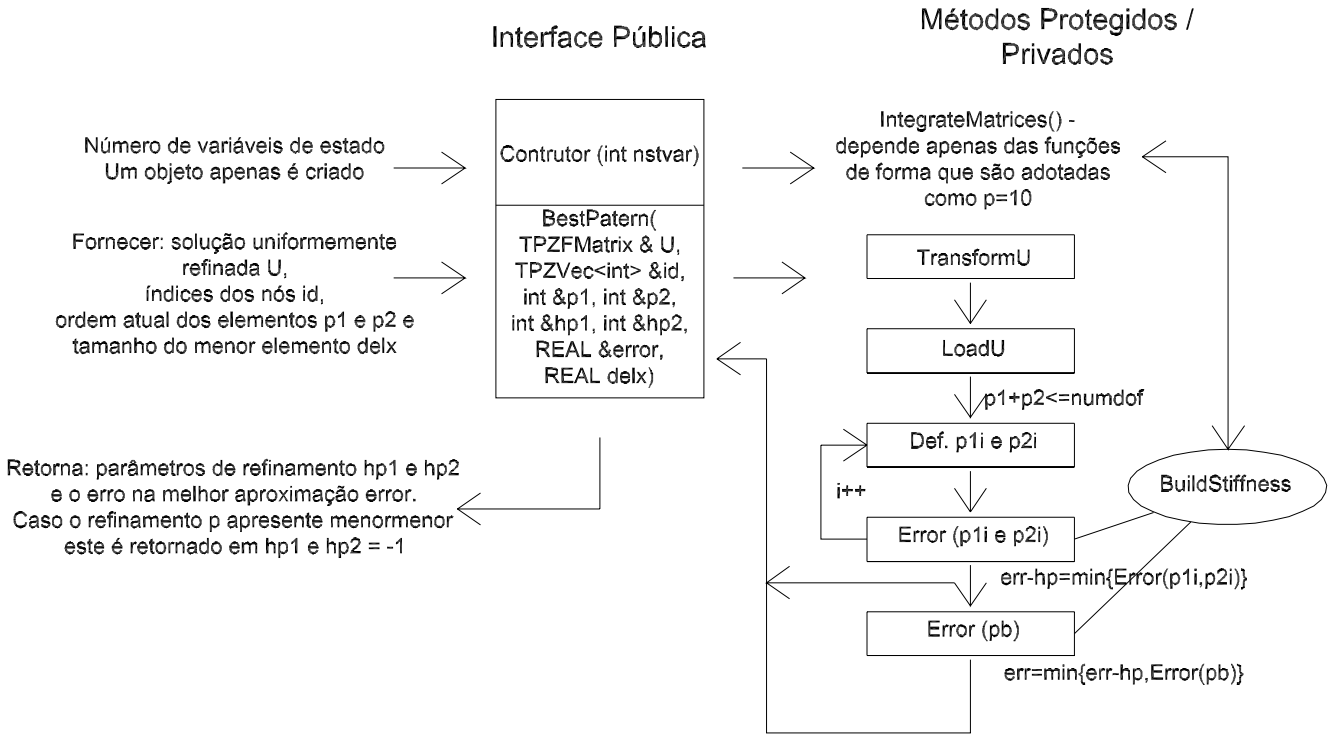
Para tal, os parâmetros que devem ser fornecidos à classe são:

- Solução U nos elementos refinados, provenientes do elemento em análise, após este sofrer um refinamento uniforme hp;
- Ordem das funções de forma dos elementos pequenos p_1 e p_2;
- Dimensão do elemento refinado Δ_x, com a qual será possível calcular a transformação entre elementos refinados e não refinados;

Com estas informações é possível determinar as funções de forma dos elementos, bem como as transformações entre elementos refinados e o elemento não refinado, sendo possível a transferência da solução fornecida nos subelementos para o elemento pai.

A Figura (2) ilustra a forma de utilização desta classe, sendo seus métodos descritos na seqüência.

Figura 2: Interface da Classe TPZOneDRef



Construtor: TPZOneDRef (int nstate)

No construtor são criadas todas as matrizes K (matriz de rigidez sem considerar os nós com restrições), F (vetor de carga), U (solução procurada) e M (matriz auxiliar), dos elementos refinados e dos elementos grandes, considerando um tamanho fixo, sendo definida como ordem máxima de interpolação 10. Esse valor foi escolhido com base na prática, uma vez que ordens de interpolação muito elevadas podem causar grandes oscilações entre os nós, afetando a estabilidade numérica do código.

A matriz M tem a função de armazenar os valores relativos à solução, incluindo os nós com restrição.

O parâmetro *nstate* indica o número de variáveis de estado que terá o vetor solução.

Feito o dimensionamento desses vetores, estes já são inicializados pelo método *IntegrateMatrices()*.

void IntegrateMatrices()

No ambiente PZ, os aspectos geométricos e computacionais de malhas e elementos são separados. Desta forma, dada a ordem de refinamento de cada elemento, que no caso é fixo, e também as funções de forma, que são calculadas com respeito ao elemento mestre em termos de bases hierárquicas, é possível a montagem das matrizes de rigidez dos elementos refinados e do elemento não refinado.

Assim, são feitos os seguintes cálculos:

$$fMS_1S_1 = \int_{\widetilde{\Omega_1}} \psi_{s_i}.\psi_{s_j}.d\widetilde{\Omega}_1 \tag{1}$$

$$fMS_1B = \int_{\widetilde{\Omega_1}} \psi_{s_i}.\psi_{b_j}.d\widetilde{\Omega}_1 \tag{2}$$

$$fMS_2B = \int_{\widetilde{\Omega_2}} \psi_{s_i}.\psi_{b_j}.d\widetilde{\Omega}_2 \tag{3}$$

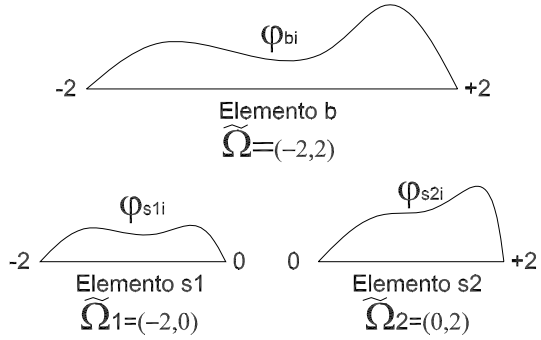
$$fMBB = \int_{\widetilde{\Omega}_1+\widetilde{\Omega}_2} \psi_{s_i}.\psi_{s_j}.d\widetilde{\Omega} \tag{4}$$

$$fKS_1S_1 = \int_{\widetilde{\Omega_1}} \psi'_{s_i}.\psi'_{s_j}.d\widetilde{\Omega}_1 \tag{5}$$

$$fKS_1B = \int_{\widetilde{\Omega_1}} \psi'_{s_i}.\psi'_{b_j}.d\widetilde{\Omega}_1 \tag{6}$$

$$fKS_2B = \int_{\widetilde{\Omega_2}} \psi'_{s_i}.\psi'_{b_j}.d\widetilde{\Omega}_2 \tag{7}$$

Figura 3: Notação para as funções de forma



$$fKBB = \int_{\widetilde{\Omega}_1+\widetilde{\Omega}_2} \psi'_{b_i}.\psi'_{b_j}.d\widetilde{\Omega} \tag{8}$$

As funções ψ_{s_i} e ψ_{b_i}são as funções de forma dos elementos refinados e do elemento não refinado, respectivamente, sendo seu cálculo realizado através do método *TPZShapeLinear::Shape1D(int ptx, int order, TPZMatrix phi, TPZMatrix dphi, int id)*, onde *ptx* é o ponto onde se quer calcular a função, *order* indica a ordem com a qual devem ser criadas as funções de forma, *phi* é a matriz onde serão armazenados os valores da função *phi* e *dphi* será a matriz onde serão armazenados os valores calculados para a derivada de *phi*. A Figura (3) ilustra a notação utilizada.

REAL BestPattern (TPZFMatrix &U, TPZVec<int> &id, int &p1, int &p2, int &hp1, int &hp2, REAL &hperror, REAL delx)

Este método é o único método público, além do construtor, da classe e faz a chamada de todas as funções necessárias à obtenção dos parâmetros $h-p$ ótimos.

Os parâmetros de entrada são:

- U: matriz com os resultados obtidos através do refinamento uniforme $h-p$;
- *id*: vetor com os identificadores dos nós, utilizado para a correção das funções de forma. Isso é necessário em função da implementação atual considerar que uma função inicia-se no nó de identificador menor e termina no nó de identificador maior;
- *p1* e *p2*: ordens dos polinômios dos subelementos 1 e 2, obtidos após o refinamento uniforme;
- *hp1* e *hp2*: ordens dos polinômios para os subelementos 1 e 2 os quais minimizam o erro e o número de graus de liberdade;
- *hperror*: retornará o erro correspondente aos parâmetros $h-p$ ótimos calculados;

- *delx*: dimensão geométrica dos subelementos. Esse valor é necessário para o cálculo das transformações entre subelementos e elemento não refinado (cálculo do Jacobiano da transformação linear).

Com estes dados e com os valores das matrizes calculadas quando da criação do objeto, serão realizadas as seguintes operações:

- Verificação das funções de forma dos elementos - método *TransformU;*
- Transporte da solução dos subelementos para o elemento não refinado - método $LoadU$;
- Nova verificação das funções de forma - *TransformU;*
- Cálculo do número de graus de liberdade $numdof$, sendo adotado o valor do elemento de maior número de graus de liberdade decrescido de um, pois buscamos uma solução com um número de graus de liberdade menor do que a solução uniformemente refinada;
- Define-se ordem $P_1 = 1$ e $P_2 = numdof + 1 - P_1$;
- Calcula-se o erro desta primeira aproximação - $besterror$;
- Varia-se $p_1$de 2 até $numdof - 1$, com $p_2 = numdof + 1 - P_1$ e para cada combinação calcula-se o erro - método *Error(int p1, int p2).* Caso o erro obtido em uma determinada iteração seja menor que $besterror$, esse é redefinido com o novo valor e $P_1 = p_1$ e $P_2 = p_2$. Ao final de todas as iterações ter-se-á a melhor combinação $h - p$ e o menor erro possível;
- A última verificação que é feita é a não utilização do refinamento h, e o cálculo do erro - método *Error(int pb).* Caso o erro obtido seja menor, a ordem p_b calculada é definida como ótima e retornada em hp_1 sendo hp_2 definido como -1, de modo a identificar a não utilização do refinamento h;

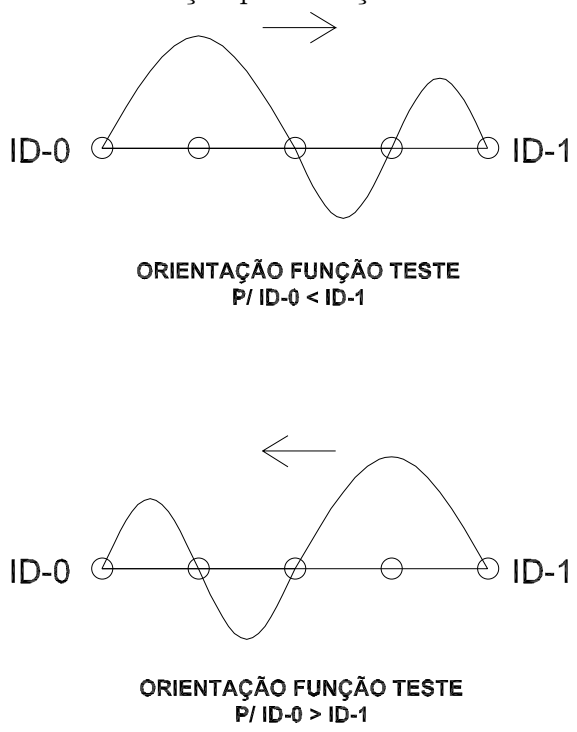
Os métodos citados acima são descritos na seqüência.

TransformU(TPZFMatrix &U, TPZVec<int> &id, int p1, int p2)

O método *TPZOneDRef::TransformU(TPZFMatrix &U, TPZVec<int> &id, int p1, int p2)* realiza uma compatibilização das funções de forma de ordem ímpar. Isso acontece pelo fato destas funções poderem satisfazer as condições necessárias para uma função de forma de duas maneiras distintas, ou sendo côncava ou convexa no trecho entre o nó inicial e o primeiro nó intermediário.

A convenção adotada no PZ é que a função sempre deve ser orientada do nó de menor identificador para o nó de maior identificador, conforme mostrado na Figura (4).

Figura 4: Convenção para funções de ordem ímpar



LoadU (TPZFMatrix &U, int p1, int p2, REAL delx)

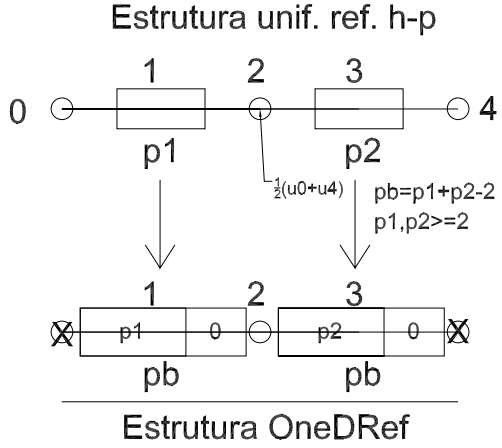
Neste método, a solução U, proveniente do refinamento uniforme $h-p$, é passada para os elementos refinados da classe, considerando que os elementos refinados terão ordem de refinamento $p_b = p_1 + p_2 - 2$, onde p_1 e p_2 são as ordens das funções de forma dos elementos provenientes do refinamento uniforme. O parâmetro *delx* representa a menor dimensão dos elementos refinados, sendo utilizada para o cálculo da transformação entre os elementos refinados e o elemento não refinado.

Com a transformação calculada, aplica-se o Jacobiano da transformação às matrizes de rigidez já calculadas e tem-se a matriz de rigidez dos elementos refinados em termos do espaço do elemento não refinado.

Através de um pós-processamento normal, dada a solução no elemento refinado, é possível encontrar-se a solução no elemento não refinado.

Outro procedimento feito é tornar nula a solução nos nós extremos (nó 0 e nó 2), de modo a possibilitar a solução de um problema tendo como malha os dois elementos refinados. Desta forma, considera-se que o erro nesses pontos será nulo, sendo considerado o erro relativo ao restante do domínio. Os procedimentos são ilustrados na Figura (5).

Figura 5: LoadU - transferência de blocos



void BuildStiffness (int p1, int p2, TPZFMatrix &stiff)

Esse método calcula a matriz de rigidez, com base nas ordens de refinamento p_1 e p_2, retornando a matriz gerada em *stiff.*

Os valores da matriz são aqueles calculados no método *IntegrateMatrices()*, entretanto, por motivo de consideração de condições de contorno Dirichlet homogêneas, os nós de extremidade da malha, onde a resposta é fixada em zero, não são considerados quando da alocação da matriz *stiff.*

REAL Error (int p1, int p2)

Este método retorna o erro mínimo que pode ser obtido utilizando-se ordens de refinamento p_1 e p_2 nos subelementos.

Para tal é realizada a seguinte seqüência de operações:

1. Gera-se uma matriz de rigidez *stiff* - método *BuildStiffness(p1,p2,stiff)*, tendo como parâmetros p_1 e p_2;

2. Gera-se o vetor de resíduos *rhs*, tendo o cuidado de utilizar a mesma estrutura de alocação de blocos utilizado na geração da matriz de rigidez;

3. Calcula-se a solução do problema $stiff * u_{ref} = rhs$. Para tal utiliza-se um *solver* do tipo LDLT, sendo o resultado retornado na matriz *rhs*;

4. É calculada a diferença entre a solução inicial e a solução aqui calculada $\Delta u = u - u_{ref}$;

5. O erro é então calculado através de (4.11), substituindo Δu e lembrando que : $K_{ij} = \sum_i \sum_j \psi_i \psi_j$, corresponde a matriz de rigidez global, temos:

$$erro = \sum_i \sum_j \Delta u.K.\Delta u$$

REAL Error(int pb)

Este método calcula o menor erro que pode ser obtido sem a utilização de refinamento h, com um refinamento geral p_b, sendo realizadas as seguintes operações:

1. Gera-se um vetor de resíduos rhs_b de ordem $p_b - 1$, com base no vetor de resíduos original (obtido do elemento uniformemente refinado $h - p$);

2. Gera-se uma matriz de rigidez para o elemento não refinado. Como já existe uma matriz de rigidez calculada para os elementos refinados, esta é aproveitada, sendo sob ela aplicada a transformação adequada (aplicação do Jacobiano)

$$stiff = \frac{1}{\Delta x} * fKSS$$

3. Calcula-se a projeção da solução do elemento refinado no elemento não refinado, sendo essa considerada sua solução para o trecho de sobreposição com o elemento refinado.

4. Resolve-se o sistema gerado $stiff * u_{res} = rhs_b$;

5. Ajustam-se os blocos de matrizes, de tal forma a se chegar ao padrão utilizado pelo ambiente PZ, onde os blocos dos nós com restrição não são considerados durante esse cálculo, pois seus resultados são conhecidos e nulos;

6. Calcula-se $\Delta u = u - u_{res}$;

7. Da mesma forma que descrito no método anterior, o erro é dado por:

$$erro = \sum_i \sum_j \Delta u.K.\Delta u$$

struct TPZRefPattern { int fId[3]; int fp[2]; int fh[2]; REAL fhError; REAL fError; }

Esta é uma estrutura de armazenamento de informações de refinamento unidimensional, sendo considerado sempre que a estrutura consiste de um elemento refinado uniformemente em $h - p$, tendo assim dois subelementos gerados. Seus objetos armazenam as seguintes informações:

- *fId[3]*: armazena os identificadores dos nós iniciais e finais dos subelementos;
- *fp[2]*: armazena a ordem de refinamento p para cada um dos subelementos;
- *fh[2]*: armazena a ordem de refinamento h para cada um dos subelementos;
- *fhError*: armazena o menor erro obtido através de refinamento h podendo ter ou não refinamento p:
- *fError*: menor erro obtido entre *fhError* e o erro obtido por refinamento p simples.

Referências Bibliográficas

[1] M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Pure and Applied Mathematics. First edition edition, 2000.

[2] M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Pure and Applied Mathematics. First edition edition, 2000.

[3] A. E. Assan. *Método dos Elementos Finitos: primeiros passos*. Coleção Livro Texto. Editora da Unicamp, segunda edição edition, 2003.

[4] I. Babuska and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *International Journal of Numerical Methods in Engineering*, Vol. 12:pag. 1597–1615, 1978.

[5] I. Babuska, T. Strouboulis, and K. Copps. H-p optimization of finite elements aproximations: Analysis of the optimal mesh sequences in one dimension. *Computer Methods in Applied Mechanics and Engineering*, Vol. 150:89–108, 1997.

[6] E.B. Becker, J. Tinsley Oden, and Graham F. Carey. *FINITE ELEMENTS: An Introduction*, volume I. Prentice-Hall, 1983.

[7] Cedric M. A. A. Bravo. *Sobre a implementação da técnica hp-adaptativa tri-dimensional para elementos finitos*. PhD thesis, Faculdade de Engenharia Mecânica, UNICAMP, 2000.

[8] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Number ISBN 0201379503. Pearson Education, 1 edition, 1999.

[9] CENAPAD - SP. *Introdução ao MPI*.

[10] E.C. Correia da Silva, P.R.B. Devloo, L. Slhessarenko, and F.A.M. Menezes. An object oriented environment for the development of parallel finite element applications. In E. Dvorkin S. Idelsohn, E. Oñate, editor, *Computational Mechanics, New Trends and Applications*. Fourth World Congress on Computational Mechanics (IV WCCM), Buenos Aires, Argentina, june 1998.

[11] E.C. Correio Da Silva and P. R. B. Devloo. Paralelização de elementos finitos utlizando programação orientada para objetos. In *Proceedings XVIII CILAMCE - Congresso Ibero Latino Americano de Métodos Computacionais Em Engenharia*, Brasília, Brasil, 1997.

[12] R. Courant. Variational methods for the solution of problems of equilibrium and vibration. *Bull. Am. Mathem. Soc.*, 49:1–23, 1943.

[13] B. F. de Veubeke. Displacement and equilibrium models in the finite element method, 1965.

[14] L. Demkowicz. 2d hp-adaptive finite element package (2dhp90) version 2.0. TICAM - Report 02-06, TICAM - Texas Institute for Computational and Applied Mathematics - The University of Texas at Austin, Austin, TX 78712, 2002.

[15] L. Demkowicz, D. Pardo, and W. Rachowicz. 3d hp-adaptive finite element package (3dhp90) version 2.0. TICAM - Report 02-24, TICAM - Texas Institute for Computational and Applied Mathematics - The University of Texas at Austin, Austin, TX 78712, 2002.

[16] L. Demkowicz, W. Rachowicz, and Ph. Devloo. A fully automatic hp-adaptivity. TICAM Report 01-28, TICAM - University of Texas at Austin, 2001.

[17] P. R. B. Devloo. History of the finite element method. Report on Scientif Methodology, April 1984.

[18] P. R. B. Devloo. Pz : An object oriented environment for scientific programming. (150):133–153, 1997.

[19] P. R. B. Devloo. {PZ} : An object oriented environment for scientific programming. *Computer Methods in Applied Mechanics and Engineering*, 150:133–153, 1997.

[20] Philippe R. B. Devloo. An object oriented framework for flexible mechanism simulation. In *European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS*, pages 1–13, 2000.

[21] Philippe R. B. Devloo and Cedric M. A. BRAVO. An object oriented approach to adaptive finite element techniques. In *European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS*, pages 1–21, 2000.

[22] Philippe R. B. Devloo and Cedric M. A. BRAVO. Sobre o refinamento uni-, bi- e tridimensional h-p adaptativo de elementos finitos. In *IV SIMMEC Simpósio Mineiro de Mecânica Computacional*, pages 125–139, 2000.

[23] Philippe Remy Bernard Devloo. *An H-P Adaptive Finite Element Method for Steady Compressible Flow*. PhD thesis, The University of Texas at Austin, August, 1987.

[24] P.R.B. Devloo. Object oriented programming applied to the development of scientific software. In E. Dvorkin S. Idelsohn, E. Oñate, editor, *Computational Mechanics, New Trends and Applications*, Edificio C-1 Campus Nord UPC, Gran Capità, s/n, 08034 Barcelona, Spain, june 1998. Fourth World Congress on Computational Mechanics (IV WCCM), CIMNE.

[25] Kevin Dowd and Charles Severance. *High Performance Computing*. O'Reilly & Associates, second edition, July 1998.

[26] J.E. Flaherty, R.M. Loy, M.S. Shephard, and J.D. Teresco. Software for the parallel adaptive solution of conservation laws by discontinuous galerkin methods. In B. Cockburn, G.E. Karniadakis, and S.-W. Shu, editors, *Discontinuous Galerkin Methods Theory, Computation and Applications*, pages 113–124, Berlin, 2000. Springer.

[27] Joseph E. Flaherty and James D. Teresco. Software for parallel adaptive computation. In Michel Deville and Robert Owens, editors, *Proc. 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, 2000. IMACS. Paper 174–6.

[28] José Davi Furlan. *Modelagem de Objetos Através da UML - The Unified Modeling Language*. Makron Books do Brasil Ltda., 1998.

[29] Adele Goldberg. *Smalltalk-80 : The Interactive Programming Environmen*. Addison-Wesley Publishing, 1983.

[30] William Gropp and Ewing Lusk. *Installation Guide to mpich, a Portable Implementation of MPI*. Argone National Laboratory, University of Chicago, version 1.2.2 edition, May 1996. Mathematics and Computer Science Division.

[31] Ceki Gülcü. Short introduction to log4j. Documentation, The Apache Software Foundation, March 2002. http://logging.apache.org/log4j/docs/manual.html.

[32] B. Tourancheau J. J. Dongarra. *Environments and Tools for Parallel Scientific Computing*. Philadelphia : SIAM, 1994.

[33] Frank Williamson Jr. An historical note on the finite element method. *International Journal for Numerical Methods in Engineering*, 15:930–934, 1980.

[34] Ted Kaehler and Dave Patterson. *A Taste of Smalltalk*. WW Norton & Co, May 1986.

[35] Glen Krasner. *Smalltalk-80 : Bits of History, Words of Advice*. Addison-Wesley Publishing, 1983.

[36] Erwin Kreyszig. *Introductory Functional Analysis with Applications*. John Wiley & Sons, 1978.

[37] P. Ladeveze and M. Zlamal. Advances in adaptive computational methods in mechanics. *SIAM J. Numer. Anal.*, Vol. 20:pag. 485–509, 1983.

[38] Gustavo C. Longhin and Philippe R. B. Devloo. Parallelization of a scientific code using oopar. In *IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING*, volume XXIV. ABMEC, 2003.

[39] H. C. Martin M. J. Thurner, R. W. Clough and L. J. Topp. Stiffness and deflection analysis of comples structures. *J. Aer. Sci.*, page 805, Sept. 1956.

[40] Punet Narula. An adaptive mesh refinement (amr) library using charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2001.

[41] NewAuthor0, James Gosling, and David Holmes. *The Java Programming Language*. The Java Series. Pearson Education, third edition edition, 2001.

[42] J. T. Oden, L. Demkowicz, W. Rachowicz, and T. A. Westermann. Toward a universal h-p adaptative element strategy - part 2 a posteriori error estimation. *Computer Methods in Applied Mechanics and Engineering*, Vol. 77:pag. 113–180, 1989.

[43] John Tinsley Oden, F. Carey, Graham, and E. B. Becker. *Finite Elements - An Introdution*, volume Vol. 1. Prentice Hall Inc., New Jersey - USA, 1981.

[44] D Pardo and L. Demkowicz. Integration of hp adaptivity multigrid. TICAM - Report 02-33, TICAM - University of Texas at Austin, 2002.

[45] M. Paszynski and L. Demkowicz. Parallel, fully automatic hp-adaptive 3d finite element package. ICES - Report 05-33, ICES - The Institute for Computational Engineering and Sciences - The University of Texas at Austin, Austin, TX 78712, 2005.

[46] Abani K. Patra, Jingping Long, and A. Laszloffy. Efficient parallel adaptive finite element methods using self-scheduling data and computations. In *HiPC 1999*, pages 359–363, 1999.

[47] K. Friedrichs R. Courant and H. Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Math. Ann.*, pages 34–74, 1928.

[48] Jean-Francois Remacle, Klaas Ottmar, Joseph E. Flaherty, and Shephard Mark S. Parallel algorithm oriented mesh database. Technical report, Sientific Computational Research Center, Rensselaer Polytechnic Institute, Troy, New York - USA.

[49] E. C. Rylo and P. R. B. Devlo. Adaptatividade hp aplicada em malhas de elementos finitos. Master's thesis, Faculdade de Engenharia Civil - Departamento de Estruturas - Universidade Estadual de Campinas, Julho 2002.

[50] Erick Slis R. Santos. Desenvolvimento de método implícito para simulador numérico tridimensional de escoamentos compressíveis invíscidos. Master's thesis, Faculdade de Engenharia Civil - Departamento de Estruturas - Universidade Estadual de Campinas, 2004.

[51] M. Snir. *MPI: The complete reference*. Unknown, 1996.

[52] H. L. Soriano. *Método de Elementos Finitos em Análise de Estruturas*. Edusp - Editora da Univesidade de São Paulo, 2003.

[53] B. Szabó and I. Babuska. *Finite Element Analysis*. A Wiley-interscience publication, 1991.

[54] Michael K. Wong and James S. Peery. Modern industrial simulation tools. Internal Report SAND96-2951, Sandia National Laboratories, Albuquerque, NM 87185-0819, February 1997.

[55] O. C. Zienkiewicz. The background of error estimation and adaptivity in finite element computations. *Computer methods in applied mechanics and engineering*, (195):207–213, 2006.

[56] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*, volume 2. Butterworth & Heinemann, fifth edition, 2000.

[57] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptative for pratical engineering analysis. *International Journal for Numerical Methods in Engineering*, Vol. 24:pag. 337–357, 1987.