UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL

PARALELIZAÇÃO DE CÁLCULOS DE ELEMENTOS FINITOS UTILIZANDO PROGRAMAÇÃO ORIENTADA A OBJETOS

Érico Correia da Silva

Orientador: **Prof. Dr. Philippe R. B. Devloo**

Dissertação de mestrado apresentada à Faculdade de Engenharia Civil como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Civil.

Área de Concentração: Estruturas

Campinas - SP, Brasil
1997

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

Si38p

Silva, Érico Correia da
Paralelização de cálculos de elementos finitos utilizando programação orientada a objetos / Érico Correia da Silva.--Campinas, SP: [s.n.], 1997.

Orientador: Philippe R. B. Devloo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Civil.

1. Métodos dos elementos finitos. 2. Programação orientada a objetos. 3. Processamento paralelo. 4. Métodos do gradiente conjugado. I. Devloo, Philippe R. B. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil. III. Título.

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL

PARALELIZAÇÃO DE CÁLCULOS DE ELEMENTOS FINITOS UTILIZANDO PROGRAMAÇÃO ORIENTADA A OBJETOS

Érico Correia da Silva
Orientador: **Prof. Dr. Philippe R. B. Devloo**
DISSERTAÇÃO DE MESTRADO

Campinas - SP, Brasil
1997

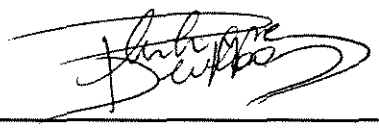
UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA CIVIL

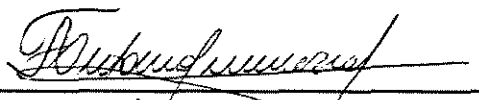
PARALELIZAÇÃO DE CÁLCULOS DE ELEMENTOS FINITOS UTILIZANDO PROGRAMAÇÃO ORIENTADA A OBJETOS

Érico Correia da Silva.

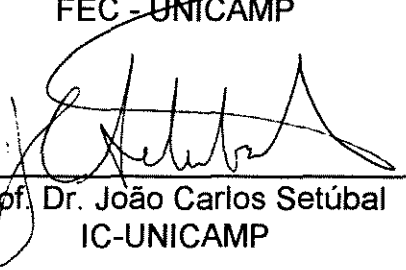
Dissertação de Mestrado defendida e aprovada, em 23 de junho de 1997, pela Banca Examinadora constituída pelos professores:



Prof. Dr. Philippe R. B. Devloo, presidente
FEC - UNICAMP



Prof. Dr. Francisco Antonio Menezes
FEC - UNICAMP



Prof. Dr. João Carlos Setúbal
IC-UNICAMP

Paralelização de Cálculos de Elementos Finitos utilizando Programação Orientada a Objetos

Érico Correia da Silva
RA 956347

Faculdade de Engenharia Civil
UNICAMP

À minha querida esposa Luciana, como prova do meu amor e símbolo da minha gratidão.

Agradecimentos

Ao Centro Nacional de Processamento de Alto Desempenho em São Paulo (CENAPAD-SP), pelo completo ambiente computacional de importância inestimável para o sucesso deste trabalho.

Ao amigo Prof. Philippe R. B. Devloo, pelo carinho e dedicação com os quais orientou os trabalhos.

A todos do CENAPAD-SP, em especial à Ana Drummond, por toda a colaboração e amparo e acima de tudo pela nossa grande amizade.

Aos amigos, professores e colegas, da Faculdade de Engenharia Civil, por todo o carinho com o qual fui acolhido.

A todos os meus familiares, pelo incentivo e pela compreensão.

Ao meu pai José e à minha mãe Luiza, por terem me amparado e apoiado, com o máximo de carinho, no decorrer destes longos anos dedicados a este trabalho.

À minha querida esposa Luciana, por todo o amor e dedicação, pela compreensão e pelo incentivo.

E sobretudo a Deus, por ter me protegido e iluminado, todos os dias... todos os momentos...

Resumo

Com o advento dos grandes computadores para processamento paralelo, veio a possibilidade de um aumento na dimensão dos problemas viáveis e diminuição do tempo de resposta para simulações por elementos finitos. Este trabalho trata da paralelização de um ambiente orientado a objetos, próprio para o desenvolvimento de aplicações de elementos finitos. Com este ambiente, baseado em arquiteturas de memória distribuída, todo o processamento é dividido em mais de um processador desde a leitura dos dados e montagem das malhas parciais, até a solução e geração dos arquivos de saída. A solução do sistema global de equações, gerado pelo problema de elementos finitos, deve utilizar neste trabalho, um algoritmo de gradiente conjugado pré condicionado paralelo, também desenvolvido como parte deste trabalho. Uma aplicação implementada com base no ambiente paralelizado de elementos finitos, poderá ser executada tanto nos grandes computadores paralelos, como em redes locais de estações de trabalho, o que viabilizará o estudo de problemas que antes não podiam ser analisados, seja por falta de memória ou pelo excessivo tempo de processamento.

Abstract

After the advent of big computers for parallel processing, came the possibility of increasing the dimension of the viable problems and decreasing the time for answer with finite element simulations. This work deals the parallelization of an object oriented environment for developing finite element applications. With this environment, based on distributed memory architectures, all computations are divided into more than one processor, since the data reading and building partial meshes, until the system solving and writing output files. For solving the global system of equations, generated by the finite element problem, will be used a parallel pre-conditioned conjugate gradient algorithm, developed also as a part of this work. An application developed based on the parallel finite element environment, can be executed as in big parallel computers as well in local net of workstations, making possible the study of problems that couldn't be analysed before, because of a lack of memory or excessive processing time.

Sumário

1. Introdução 9

1.1 O ambiente PZ 10
1.2 O ambiente OOPAR 11
1.3 Objetivos 11

2. Procedimentos para paralelização de cálculo de elementos finitos 14

2.1 O método dos elementos finitos 14
2.2 Elementos finitos e programação orientada a objeto 16
2.3 Processamento paralelo 17
2.3.1 Modelos computacionais paralelos 18
2.3.2 O modelo message-passing 19
2.3.3 Softwares para paralelismo 21
2.3.4 Modelagem por Redes de Petri 24
2.4 Cálculo de elementos finitos em paralelo 26
2.4.1 Decomposição do domínio 26
2.5 Métodos iterativos 28
2.6 O algoritmo de gradiente conjugado 30
2.7 Gradiente conjugado pré condicionado paralelo 32

3. Programação de elementos finitos orientada a objeto: O ambiente PZ 34

3.1 Classes do Ambiente PZ 35
3.1.1 Pré processamento 36
3.1.2 Pós processamento 37
3.1.3 Classes para aproximação da geometria 37
3.1.4 Classes para definição do espaço de interpolação 47
3.1.5 Classes para definição dos materiais e condições de contorno 53
3.1.6 Montagem e solução do sistema - A classe TAnalysis 55
3.2 Código exemplo 56

4. Computação paralela orientada a objeto: O ambiente OOPAR 59

4.1 Classes e Objetos do OOPAR 60
4.1.1 Data Manager (DM) - classe TDataManager 60
4.1.2 Task Manager (TM) - classe TTaskManager 63
4.1.3 Communication Manager (CM) - classe TCommunicationManager 64
4.1.4 Classe TSaveable 65
4.1.5 Classe TStorage 66
4.1.6 Classe TTask 66
4.2 Protocolo de Transferência do Data Manager 67
4.3 Dependência de tarefas sobre versões de dados 69

5. Exemplo de organização de programas com o OOPAR 71

5.1 Estrutura e distribuição dos dados 72
5.2 Divisão das tarefas 73
5.2.1 Verificação da convergência 76
5.3 Modelagem do sistema 77
5.4 Dependências 79
5.5 Codificação 82
5.5.1 Classe TTask1 83
5.5.2 Classe TTask2 (tarefas 2 e 5) 84
5.5.3 Classe TTask3 85
5.5.4 Classe TTask4 85
5.5.5 Classe TTask6 86

6. Gradiente conjugado pré condicionado: Versão paralela com o OOPAR88

6.1 Divisão das Tarefas ... 89
6.1.1 Divisão dos dados em Data Sets ... 90
6.2 Classe TMapManager ... 95
6.2.1 Criação dos Mapas ... 97
6.3 Procedimentos, Tarefas e Classes ... 98
6.3.1 Procedimentos iniciais ... 98
6.3.2 Classe TCGGlobalData ... 99
6.3.3 Classes TSvDouble e TSvInt ... 100
6.3.4 Classe TCGStartTask ... 100
6.3.5 Classe TCGTask1 ... 102
6.3.6 Classe TCGTask3 ... 105
6.3.7 Classe TCGTask5 ... 109
6.3.8 Classe TCGTask7 ... 112
6.3.9 Classe TCGTask8 (tarefas 8 e 10) ... 113
6.3.10 Classe TChkTask (tarefa 11) ... 115
6.3.11 Classe TVecUpdateTask (tarefas 4 e 6) ... 117
6.3.12 Classe TDblUpdateTask (tarefas 2, 7a e 9) ... 119
6.4 Dependência de dados ... 120
6.4.1 Dependências e controle das versões dos dados ... 125

7. Paralelização do cálculo de elementos finitos ... 131

7.1 Introdução ... 131
7.2 Determinação das informações de fronteira ... 132
7.2.1 Malha geométrica ... 133
7.2.2 Malha computacional ... 134
7.3 Solução por gradiente conjugado pré condicionado paralelo ... 138
7.3.1 Montagem dos dados do algoritmo PCG paralelo ... 139
7.3.2 Montagem dos mapas do Map Manager em paralelo ... 141
7.3.3 Início do algoritmo PCG paralelo ... 142
7.4 Geração dos resultados ... 143
7.5 Análise de elementos finitos em paralelo (implementação) ... 143
7.5.1 Objetos transmitidos de um processo para os demais ... 144
7.5.2 Objetos locais de cada processo ... 145
7.5.3 Procedimentos e Tarefas ... 152
7.5.4 Dependência de dados ... 158

8. Avaliação de desempenho ... 160

8.1 O problema de teste ... 162
8.2 Acompanhamento da execução ... 162
8.3 Escalabilidade e Eficiência relativa ... 166

9. Conclusão ... 170

10. Referências ... 173

Figuras

Figura 1 - Problema de análise de tensões discretizado para solução com o método dos elementos finitos15
Figura 2 - Troca de mensagens em baixo nível 20
Figura 3 - Rede de Petri 25
Figura 4 - Exemplo de rede de Petri - fonte MACIEL (1996) 25
Figura 5 - Modelos básicos de sincronismo e distribuição 26
Figura 6 - Seqüência de criação dos objetos PZ 36
Figura 7 - Classes gráficas - hierarquia 37
Figura 8 - Hierarquia de elementos geométricos PZ 39
Figura 9 - Ilustração dos dados tratados por TGeoEl 42
Figura 10 - Mapeamento, do elemento mestre para o domínio 44
Figura 11 - Classes de sistemas de coordenadas 45
Figura 12 - Problema de elementos finitos com o ambiente PZ 47
Figura 13 - Classes de elementos computacionais (hierarquia) 51
Figura 14 - Problema tratado no código de exemplo - fonte: SANTANA (1997), pag. 107 57
Figura 15 - Divisão de tarefas para o algoritmo de Jacobi 77
Figura 16 Rede de Petri da iteração de Jacobi para 2 processadores 78
Figura 17 - Exemplo de divisão de um sistema K x = f com 15 equações em 5 Data Sets 91
Figura 18 - Parte do sistema global de equações referente ao Data Set 3 91
Figura 19 - Divisão de tarefas do algoritmo PCG paralelo 93
Figura 20- Exemplo de organização de tarefas por dependência de versão de dados 125
Figura 21 - Análises parciais antes da determinação das informações de fronteira 132
Figura 22 - Alteração dos nós computacionais na fronteira 136
Figura 23 - Malhas parciais e informações de fronteira (configuração final) 137
Figura 24 - Configuração irregular de malhas parciais 137
Figura 25 - Esquema de seqüência de tarefas para dois processadores 153
Figura 26 - Visualização DX do momento e da deformação da placa de teste (calculados com 1, 3 e 6 procs.) 162
Figura 27 - Análise do programa de teste com o XPVM 164
Figura 28 - Geração dos resultados / Finalização do programa. 166
Figura 29 - Análise de escalabilidade com tamanho fixo: Tempo de execução 167
Figura 30 - Análise de escalabilidade com tamanho fixo: Eficiência 167
Figura 31 - Análise de escalabilidade com tamanho fixo: Tempo de execução (SP) 168
Figura 32 - Análise de escalabilidade com tamanho fixo: Eficiência (SP) 168

Paralelização de Cálculo de Elementos Finitos Utilizando Programação Orientada a Objetos

Érico Correia da Silva
Faculdade de Engenharia Civil
UNICAMP

Capítulo 1

1. Introdução

A informática é uma das áreas que mais progrediram nos últimos anos. Os micro computadores estão cada vez mais velozes e mais baratos, tornando-os ferramentas presentes no dia a dia dos profissionais de engenharia. Por volta de 1989 G. F. Carey [1] já citava que algumas estações de trabalhos da época podiam oferecer uma velocidade de processamento maior que muitos dos *mainframes* de 5 anos antes. Cada vez mais os esforços de pesquisa vão sendo direcionados à solução de problemas que antes sequer podiam ser resolvidos. É com esses problemas em vista que são criadas máquinas com poder computacional cada vez maior, a um custo relativamente baixo, impelindo os desenvolvedores de software científico para uma busca exaustiva de novos algoritmos e técnicas de desenvolvimento que tornem possível um melhor aproveitamento dos novos supercomputadores.

Incrementar o poder computacional de um processador não é o melhor caminho para diminuir o tempo envolvido na solução de um problema [29][30][33]. O aumento da velocidade de um processador é extremamente dispendioso. Atualmente novas arquiteturas que associam, em paralelo, o poder de muitas estações de trabalho [29](bem mais modernas que as citadas por Carey em [1]), são cada vez mais utilizadas em problemas de mecânica computacional e computação científica em geral. Porém, desenvolver software paralelo não é algo trivial. Bertsekas e Tsitsiklis citam em [7] algumas das diferenças de complexidade entre os algoritmos seqüenciais e os seus equivalentes paralelos, podemos destacar os seguintes problemas:

- determinação de tarefas;
- comunicação;
- sincronização.

Estes problemas não existem em algoritmos seqüenciais sendo, porém, pontos vitais em algoritmos paralelos.

São problemas como os acima citados que tornam necessário o desenvolvimento de ferramentas que minimizem a complexidade encontrada no desenvolvimento de software paralelo. Dongarra e Tourancheau reuniram em [40] diversos trabalhos sobre pesquisas buscando a minimização das dificuldades encontradas no desenvolvimento de software para computação de alto desempenho. Além das ferramentas citadas por Dongarra, a literatura indica várias outras que viabilizam um desenvolvimento rápido e eficaz de software científico, podemos citar o ambiente PZ e ambiente OOPAR, ambos desenvolvidos por Devloo et al. [10][5], respectivamente voltados para o método dos elementos finitos e para processamento paralelo em geral. Ferramentas assim possibilitam o desenvolvimento de algoritmos paralelos mais complexos, aumentando dessa forma o aproveitamento dos novos supercomputadores paralelos quando aplicados na solução dos grandes problemas que, finalmente, deixariam de ser "desafios".

1.1 O ambiente PZ

Para facilitar o desenvolvimento de aplicações com base no método dos elementos, P. R. B. Devloo desenvolveu um conjunto de classes C++ baseadas na metodologia de programação orientada a objetos. Com o ambiente PZ o pesquisador centraliza seus esforços na desenvolvimento da parte do código particular ao seu problema físico, pois todas as partes do método dos elementos finitos que independem do problema físico a ser estudado já são implementadas pelas classes do PZ.

Detalhado a fundo no capítulo 3, o ambiente PZ é extremamente portável estando disponível tanto em ambientes do tipo PC como em estações de trabalho baseadas em sistemas UNIX.

1.2 O ambiente OOPAR

O OOPAR é uma ferramenta desenvolvida para facilitar o desenvolvimento de software baseado em processamento paralelo em ambientes de memória distribuída, utilizando um modelo *message-passing*[30]. Tal como o ambiente PZ, o OOPAR é uma coleção de classes C++ que implementam todo o gerenciamento e características básicas de tarefas paralelas e dados distribuídos.

Na comunicação entre os vários processadores presentes no ambiente computacional em uso, o OOPAR utiliza uma ferramenta para troca de mensagens tal como o PVM, o MPI, o MPL ou uma outra semelhante. Dessa forma o programador não se preocupa com detalhes de cada ferramenta em particular, deixando tudo a cargo do OOPAR. Problemas como sincronismo e comunicação de dados são completamente controlados com o OOPAR facilitando o desenvolvimento de softwares baseados em processamento paralelo. Maiores detalhes sobre o OOPAR são encontrados em [5] e no capítulo 4.

1.3 Objetivos

Diversas alternativas estão sendo estudas, nos últimos anos, buscando a simplificação das tarefas envolvidas no desenvolvimento de software científico paralelo [40]. O método dos elementos finitos também tem sido alvo de um número crescente de pesquisas, sendo estudada a sua aplicação em problemas de análise de estruturas, análise dinâmica, transferência de calor, mecânica dos fluidos, etc. Diversos algoritmos para solução em paralelo de problemas específicos, via elementos finitos, estão sendo desenvolvidos nos [38]. Com base no ambiente OOPAR, este trabalho tem como principal meta estender o ambiente PZ para que o mesmo possa tratar de problemas de elementos finitos ainda maiores e em um tempo menor, utilizando processamento paralelo para isto. Em suma, o objetivo maior é o desenvolvimento de uma ferramenta para facilitar a implementação de programas de elementos finitos orientados a objetos, que utilizem processamento paralelo em ambientes de memória distribuída.

Os trabalhos foram divididos em duas partes:

1) **Implementação de um algoritmo paralelo de gradiente conjugado pré condicionado :**

A eficiência do algoritmo de gradiente conjugado pré condicionado é algo comprovado[21][22]. Este é apenas um dos motivos que determinaram sua escolha como método para ser usado na solução paralelizada de grandes problemas de elementos finitos. Será implementado um algoritmo de gradiente conjugado pré condicionado paralelo, usando o ambiente OOPAR [5] para este fim.

Descrito com detalhes no capítulo 4 deste trabalho, o OOPAR é um ambiente baseado na programação orientada a objetos (OOP, *Object Oriented Programming*). Associado aos recursos e características da OOP, o OOPAR proporciona grande flexibilidade na programação e segurança, tanto de dados como de execução de tarefas. Também é meta desse trabalho testar, avaliar e documentar o ambiente OOPAR.

2) **Paralelização do ambiente PZ:**

Diferente do que tem sido feito [38], este trabalho não trata da paralelização de uma aplicação específica do método dos elementos finitos. Trata sim, da paralelização de um ambiente orientado a objetos para desenvolvimento de software de elementos finitos, o ambiente PZ [10][13], utilizando para isto o ambiente OOPAR para processamento paralelo. Com o ambiente PZ paralelizado, o usuário limitar-se-á a programar apenas o que for específico do problema físico tratado, sem preocupar-se com detalhes oriundos do paralelismo. O capítulo 3 desse trabalho descreve o ambiente PZ.

Para comprovar a eficiência da solução proposta neste trabalho, será necessária uma avaliação de desempenho. Apenas com um intuito de ilustrar a escalabilidade[33] de uma aplicação implementada com base no ambiente PZ paralelo, este trabalho

efetuará testes utilizando os recursos do CENAPAD-SP[1], sob plataformas como o IBM SP [29] ou uma rede de estações RISC-6000 conectadas via TCP/IP, de modo a deixar claro a potencialidade da programação orientada a objetos associada a ambientes de desenvolvimento tais como o PZ e o OOPAR. Também serão feitos testes do programa desenvolvido, usando números variados de processadores, para uma posterior avaliação de escalabilidade [33].

[1] Centro Nacional de Processamento de Alto Desempenho em São Paulo (FINEP/MCT/UNICAMP)

Capítulo 2

2. Procedimentos para paralelização de cálculo de elementos finitos

Estuda-se neste capítulo os procedimentos encontrados na literatura para paralelização de cálculo de elementos finitos. Vários são os assuntos a serem considerados numa implementação paralela e orientada a objetos do método dos elementos finitos. Figuram dentre os principais itens a serem abordados os seguintes:

- O método dos elementos finitos propriamente dito;
- A programação orientada a objetos aplicada ao método dos elementos finitos;
- Conceitos de processamento paralelo;
- O método a ser utilizado na solução paralelizada do sistema "$A\ x = b$", resultante da análise de elementos finitos.

Para um melhor entendimento deste trabalho, deve-se contar com um certo conhecimento de assuntos como programação orientada a objetos e C++, o método dos elementos finitos e computação numérica. Para isso o leitor pode recorrer às referências bibliográficas indicadas no decorrer do texto.

2.1 O método dos elementos finitos

Segundo o raciocínio desenvolvido em Oden et al. [23], o método dos elementos finitos é uma técnica de construção de solução aproximada para problemas de valor de contorno. Consiste na divisão do domínio do problema em um número finito de subdomínios chamados elementos, para depois construir uma aproximação da solução usando conceitos de cálculo variacional aplicados ao conjunto de elementos.

O método dos elementos finitos, inicialmente idealizado para problemas de análise de tensão, é atualmente aplicado na solução de problemas das mais diversas

áreas da engenharia, como problemas de transferência de calor, mecânica dos fluidos, campos eletromagnéticos, etc.

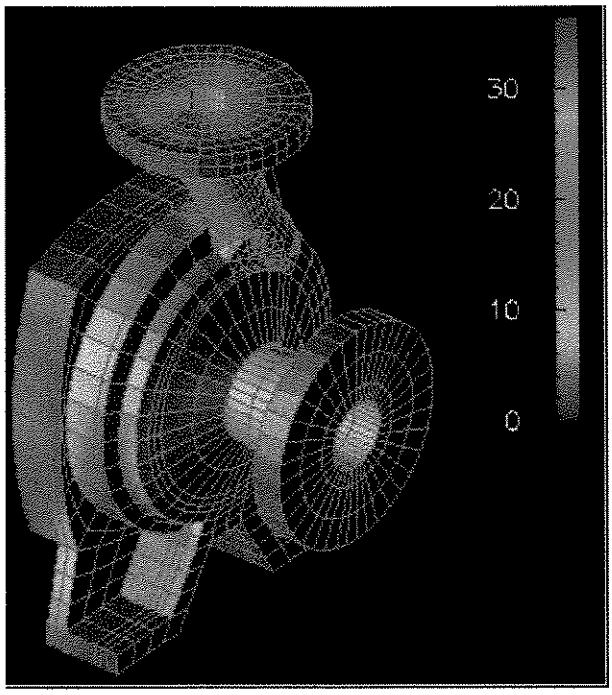


Figura 1 - Problema de análise de tensões discretizado para solução com o método dos elementos finitos

A aproximação pelo método dos elementos finitos em geral envolve os seguintes passos [24]:

1. Divisão do domínio em elementos finitos. Vários programas (pré processadores) foram desenvolvidos para a geração da malha de elementos.

2. Formulação das propriedades da equação diferencial (em análise de tensão, por exemplo, eqüivale a determinar as cargas nos nós devido a todos os estados de deformação possíveis para o elemento).

3. Montagem do modelo (*"Assemble"*), reunindo-se as contribuições dos elementos finitos.

4. Aplicação das cargas conhecidas.

5. Aplicação das condições de contorno (aplicação de valores de deslocamentos nodais já conhecidos).

6. Solução do sistema linear de equações algébricas, $K\,u = f$, para determinar a solução aproximada (tensões propriamente ditas, para o caso de uma análise de tensões).

7. Geração e/ou visualização gráfica dos resultados (pós processamento).

Todos os passos acima descritos podem ser programados para serem executados por um computador, seguindo uma sistemática muito bem definida.

Com os grandes avanços da informática ocorridos nas últimas décadas, o método dos elementos finitos ganhou grande popularidade, especialmente com o advento de computadores pessoais com alta capacidade de processamento além de interface gráfica para facilitar o desenvolvimento de modelos e a análise dos resultados.

A evolução do método é clara se considerarmos o número de artigos sobre o assunto nas últimas décadas [24]: cerca de 200 na década de 60, 7000 na década de 70 e 20000 até 1986.

2.2 Elementos finitos e programação orientada a objeto

As metodologias de programação tradicionais descrevem formas de programação com base em procedimentos, cada um descrevendo uma parte de um algoritmo. Um programa com base na metodologia de programação orientada a objetos (*OOP, Object Oriented Programming*), por sua vez, descreve objetos e como estes interagem entre si de forma sistemática, com um determinado objetivo. A programação orientada a objetos baseia-se no que deve ser feito, sendo que a forma com que tudo é feito fica restrito ao objeto. Trabalhando dessa forma o programador concentra-se no sistema como um todo, sem grande ênfase em pequenos detalhes. Os códigos ficam mais claros e o trabalho mais simples.

Linguagens orientadas a objeto como o C++ [12], concentram-se em estruturas que organizam dados e procedimentos para manipular estes dados, são as classes (C++).

As classes descrevem as entidades presentes em programas de elementos finitos (elemento, nó, malha, etc.) e outras classes descrevendo entidades da álgebra linear (matriz, vetor, solver, etc.). Com base nestas classes um programador pode desenvolver uma aplicação de elementos finitos, criando e associando objetos destas classes, e/ou

estendendo as mesmas de forma a criar novas classes com procedimentos mais específicos com respeito ao problema abordado. Tudo sem necessariamente conhecer detalhes sobre a implementação das classes.

Ulbin et al. propõem em [25] diversas classes para manipulação de vetores, matrizes, etc. dando ênfase a simplicidade de um código utilizando OOP. Deste trabalha destacamos o código a seguir, onde está ilustrado como seria um programa de elementos finitos orientado a objetos utilizando C++:

```
//Definição do objeto malha
FiniteElementMesh mesh("input.dat");
//Leitura dos dados da malha
mesh .read();
//Solução da malha
mesh .solve();
//Gravando a malha com os resultados
mesh .write();
```

Com o código acima, fica clara a importância de um objeto mesh. Sabe-se porém que objetos como elemento, nó, material, são de considerável importância, são dados da classe mesh, porém têm vida própria e podem ser aplicados em outros contextos [25].

Em [10], Devloo descreve o ambiente PZ, uma estrutura de classes para programação de elementos finitos onde cada classe busca ser tão abstrata quanto possível, para que tais classes possam ser aplicadas nas mais variadas aplicações de elementos finitos. Mais tarde, em [13], Santana ilustra o ambiente PZ com detalhes, acrescentando também uma descrição do conjunto de classes para análise matricial e algoritmos paralelos desenvolvidas por ele.

2.3 Processamento paralelo

Mesmo com os avanços tecnológicos que proporcionaram um aumento exponencial na velocidade de processamento, os computadores ainda não são velozes o bastante para que muitos problemas físicos sejam resolvidos, em tempo razoável, através de modelos matemáticos. Aumentar ainda mais a velocidade dos computadores é algo difícil e muito caro, devido as limitações impostas pelas arquiteturas convencionais de

hardware [29]. Como alternativa, surgiram os computadores paralelos, baseados em vários processadores ao invés de um. Gropp cita em [30] a seguinte frase:

"To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic oxen."

("Para puxar uma carroça maior, é mais fácil colocar mais bois do que conseguir um boi gigante.")

Esta frase resume no que consiste a computação paralela. É muito mais viável o uso de vários dos computadores disponíveis (em se tratando de uma rede de computadores), do que comprar um novo computador mais veloz (e mais caro) para a solução de um problema. A razão preço/desempenho é muito mais favorável quando parte dos recursos não precisam ser comprados. Se as redes disponíveis nos laboratórios não apresentam o desempenho exigido por um dado problema, os pesquisadores podem fazer uso dos grandes computadores paralelos, como o SP da IBM [29], que podem alcançar desempenhos inigualáveis por arquiteturas seriais.

Embora seja possível o alcance de níveis de desempenho inigualáveis, o programador encontra diversas dificuldades ao projetar e/ou implementar um sistema paralelo. Um grande problema é a divisão e distribuição das várias tarefas entre os processadores disponíveis. Axford cita este problema em [4], lembrando que o trabalho com grandes conjuntos de dados se enquadra como uma importante classe de problemas em processamento paralelo. Segundo Axford, a divisão do problema em *"Data Sets"* (conjuntos de dados) é amplamente aplicável. No capítulo 11 de [4], são propostas várias formas para o tratamento dessa classe de problemas. Outro grande problema é a organização das tarefas que compõem o programa. Este problema pode ser minimizado utilizando uma técnica de modelagem de sistemas como, por exemplo, redes de Petri.

2.3.1 Modelos computacionais paralelos

Um modelo computacional é uma descrição conceptual de quais tipos de operações estão disponíveis para o programa [30]. Um modelo computacional paralelo pode ser distinguido por diversos aspectos, tais como tipo de memória (compartilhada ou distribuída), características de execução do programa, etc. Os modelos computacionais não estão necessariamente associados ao hardware empregado.

Seguem abaixo alguns dos modelos computacionais citados por Gropp et al. [30]:

- **Paralelismo de dados**: O paralelismo é centralizado na divisão dos dados, o código paralelo é praticamente o mesmo seqüencial. Este é o modelo característico das arquiteturas vetoriais (veja [31]). É este o tipo de paralelismo implementado pelo HPF (High Performance Fortran).

- **Memória compartilhada**: O paralelismo é especificado explicitamente no código feito pelo programador, sendo que todos os processadores têm acesso a um mesmo espaço de memória. O acesso às posições de memória é controlado por algum mecanismo de *"locking"*, ou seja, enquanto um processador escreve um dado os outros têm o acesso a esse dado impedido.

- **Message passing**: Neste modelo um processador têm acesso apenas ao seu próprio espaço de memória, porém pode comunicar-se com os demais processadores e trocar dados com os mesmos. Não há dispositivos de hardware para locking ou sincronização dos processos[4].

2.3.2 O modelo message-passing

Em arquiteturas de memória distribuída tanto a comunicação entre os processos bem como a sincronização entre os mesmos é feita por troca de mensagens, ficando a cargo do programador lidar com essas atividades.

A troca de mensagens é feita usando, basicamente, um protocolo simples de *"handshake"* com os seguintes passos [4]:

1. O processador emissor (*"sender"*) envia uma mensagem colocando n bits nas linhas de dados e envia em seguida um sinal de pronto. Os dados são mantidos nas linhas de dados até que seja recebido um sinal de confirmação.

2. O processador receptor (*"receiver"*) sabe da existência de uma mensagem quando recebe um sinal de pronto, então ele lê os n bits enviados e envia de volta o sinal de confirmação.

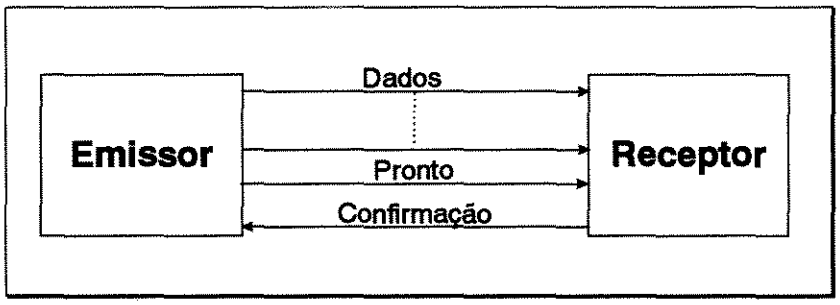


Figura 2 - Troca de mensagens em baixo nível

Em um nível mais alto, Axford divide o modelo message-passing em dois tipos:

Message passing assíncrona:

O programador conta com duas operações básicas, a saber:

- send(*message, destination*) Envia a mensagem *message* ao processador *destination.*

- receive(*message, source*) Espera até que uma mensagem seja recebida do processador *source.*

Com este esquema o tempo de execução de uma operação *send* pode não ser sincronizado diretamente com uma operação *receive*, devido a trafico na rede ou outro problema qualquer. Além disso, o receptor pode não estar esperando uma mensagem quando esta for enviada e mensagens poderão ser perdidas se uma segunda mensagem for enviada antes que a primeira tenha sido recebida. O software que implementar o modelo message-passing deverá contornar problemas como estes.

Message passing síncrona:

Existem sistemas message-passing que possuem um esquema equivalente ao *send* e *receive* exceto pelo fato da necessidade de uma sincronização completa entre os processos antes da comunicação. Aquele que começar a execução antes esperará os demais para continuar. Este esquema é chamado *rendezvous*, é mais simples para o programador sendo usado por diversas linguagens de alto nível para processamento paralelo (veja o capítulo 9 de [4]).

2.3.2.1 Vantagens do modelo message-passing

Gropp et al. citam em [30], as seguintes vantagens do modelo message-passing:

Universalidade: O modelo message-passing pode ser implementado na maioria das arquiteturas paralelas atuais, tirando vantagens de recursos de memória compartilhada quando estes estiverem presentes.

Fácil depuração: Um dos maiores problemas na depuração de programas paralelos é o acesso indevido de memórias compartilhadas. Em sistemas message-passing apenas um processador tem acesso a cada espaço de memória, tornando assim mais fácil a localização de erros de leitura e escrita em memória.

Desempenho: Uma das vantagens dos computadores de memória distribuída sobre os computadores de processador único, é o aumento no espaço de memória e cache. Com um modelo message-passing os processadores podem ser direcionados para o tratamento de dados específicos tirando maior vantagem dos recursos de cache da máquina.

Além das vantagens acima, o modelo message-passing proporciona um alto grau de escalabilidade, ou seja, é simples aumentar o poder computacional acrescentando mais uma CPU ao sistema.

2.3.3 Softwares para paralelismo

Bibliotecas de software oferecem várias vantagens tais como [30]:

- Garantem a consistência na correção do programa;
- Proporcionam implementação de boa qualidade;
- Ocultam detalhes e complexidades associadas à implementação;
- Minimizam esforços repetitivos.

As bibliotecas, mesmo as para processamento seqüencial, utilizam em suas implementações diversas técnicas para melhor atender aos objetivos aos quais se propõem. Dessa forma o programador é poupado da necessidade de um conhecimento mais aprimorado.

O desenvolvimento de aplicações científicas paralelas é significativamente mais complexo, considerando o nível de detalhamento que aplicações paralelas baseadas em message-passing requerem. As bibliotecas são extremamente necessárias para uma minimização dessa complexidade, e também para aumentar a portabilidade do código.

São várias as bibliotecas para processamento paralelo baseadas no modelo message-passing, das quais podemos destacar as seguintes:

PVM - Parallel Virtual Machine

O PVM é um software que permite ao programa enxergar uma rede heterogênea de computadores como sendo um único computador paralelo [32] [33]. O PVM foi projetado para proporcionar ao usuário uma plataforma paralela independente do número de tipos diferentes de computadores ou da distância entre eles. É composto por uma biblioteca de rotinas que permitem, dentre outras coisas, o início e término de tarefas dentro da máquina virtual e a comunicação e sincronização entre os processos. Além da biblioteca de rotinas, o PVM possui um ambiente gráfico para análise e execução de programas, o XPVM. Com o XPVM, o programador pode acompanhar a execução do programa de diversas formas, observando os eventos e transmissão dos dados com o auxílio de representações gráficas e relatórios, podendo inclusive analisar o programa após o término do mesmo.

MPL - IBM Message Passing Library

Semelhante ao PVM, o MPL possui um biblioteca de rotinas para paralelismo tipo message-passing. O MPL foi desenvolvido especialmente para o *IBM Scalable Power Parallel System 9076 - SP* e utiliza ao máximo os recursos disponíveis nesta arquitetura. O SP é uma máquina de memória distribuída constituída por estações RISC 6000, conectadas com um comunicador de alto desempenho o HPS (*High Performance Switch* - veja em [29]). Assim como o PVM o ambiente SP possui uma ferramenta de acompanhamento, depuração e análise de programas paralelos, o VT *Visualization Tool.*

MPI - Message Passing Interface

Mais que uma biblioteca de rotinas, o MPI busca ser um padrão para message-passing que seja portável, prático, eficiente e flexível[30] [33] [34]. O MPI implementa diversas características tais como comunicação ponto a ponto, operações coletivas, grupos de processos, domínios de comunicação, etc. Participaram no desenvolvimento do padrão MPI pesquisadores das grandes universidades, fabricantes de arquiteturas paralelas e laboratórios do governo americano. Sendo assim espera-se que o MPI seja amplamente difundido e utilizado.

Além dos softwares acima, Hsieh e Sotelino citam em [35] outros softwares de domínio público tais como PICL, p4, PARMARC, Chameleon, Zipcode e CHIMP, além dos softwares comerciais Express e Linda.

Além dos softwares seqüenciais citados, encontra-se na literatura softwares para processamento paralelo baseados na metodologia de programação orientada a objeto. Podemos citar os seguintes:

PPI++ - Parallel Portability Interface in C++

É uma biblioteca de message-passing orientada a objetos[35]. Foi desenvolvida para servir como interface entre a aplicação do usuário e ambientes de computação paralela. Baseada no padrão MPI, o PPI++ tira vantagem de características do paradigma de programação orientado a objetos, para esconder do usuário detalhes de implementação inerentes do modelo message-passing. Tornando o código do usuário mais simples e claro.

pC++ - Parallel C++

O pC++ [36] foi projetado para ser uma extensão do C++ capaz de implementar o modelo de paralelismo suportado pelo HPF *High Performance Fortran*. Sabe-se que o HPF implementa um modelo de paralelismo orientado por dados [30].

CC++ - Compositional C++

O CC++ [37] não é uma biblioteca, trata-se de uma linguagem de programação que estende o conhecido C++ para trabalhar com paralelismo. Pode-se implementar programas em CC++ baseados em message-passing, chamada remota de rotinas, paralelismo de dados e programação orientada a objetos paralelizada, sendo que todos estes paradigmas podem ser combinados num só programa.

OOPAR - Object Oriented Parallelism

Devloo classifica em [5] como sendo o OOPAR um ambiente integrado baseado na filosofia de programação orientada a objetos. O OOPAR possui três classes de gerenciamento: **TM** (Task Manager), **DM** (Data Manager) e **CM** (Communication Manager), que auxiliam e permitem um desenvolvimento de programas paralelos baseados em tarefas. O **DM** proporciona ao usuário um espaço de memória único, emulando memória compartilhada em sistemas de memória distribuída. Com suas classes de gerenciamento e demais classes básicas, o OOPAR busca oferecer ao programador de software científico, um ambiente seguro e flexível para execução de tarefas e manipulação de dados. O OOPAR é objeto deste trabalho sendo amplamente detalhado no capítulo 4.

2.3.4 Modelagem por Redes de Petri

Idealizada inicialmente em 1962 por C. A. Petri, na Alemanha, redes de Petri [4] [11] [28] é uma técnica de especificação de sistemas que possui diversos recursos permitindo uma verificação de propriedades de um algoritmo, detectando se o mesmo está ou não correto. Redes de Petri são eficazmente utilizadas na especificação de sistemas paralelos, concorrentes, assíncronos e não-determinísticos.

Com sua representação gráfica, as redes de Petri permitem a visualização dos processos envolvidos e a comunicação entre eles. Uma rede de Petri é composta por dois tipos de componentes: a transição (componente ativo) e o lugar (componente passivo).

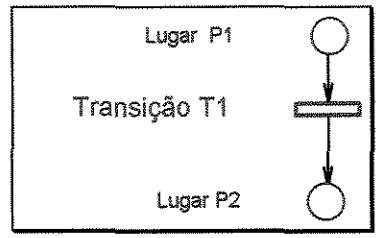


Figura 3 - Rede de Petri

Cada lugar é associado à uma variável de estado tida como condição para a execução de uma ação do sistema, representada por uma transição da rede. Os lugares são ligados às transições por meio de arcos. Uma transição só é ativada se todos os lugares que são pré condições para sua execução tiverem, pelo menos, uma marca. Por exemplo no estado indicado na Figura 4, apenas a transição T1 (início das atividades) pode ser disparada. Assim que T1 for disparada, desaparecem uma marca em cada lugar de pré condição (P1 e P7) e aparece uma marca tanto em P2 como em P3, que são os lugares de pós condição da transição T1. Nesse momento as transições T2 e T3 podem ser disparadas e assim sucessivamente. Existem softwares específicos para simulação de sistemas por redes de Petri. Para elaboração das redes presentes neste trabalho foi utilizado o software PESIM[11], que permite uma animação passo a passo de simulação de sistemas por redes de Petri, além de várias análise que podem ser feitas com este método de modelagem.

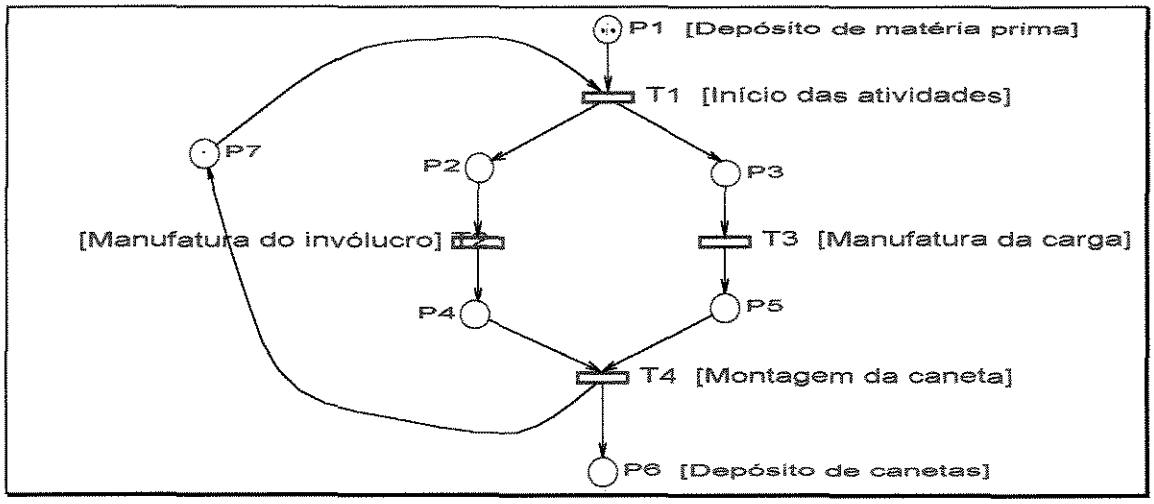


Figura 4 - Exemplo de rede de Petri - fonte MACIEL (1996)

Maciel et al. citam em [28] que o modelo do processo paralelo global é obtido pela união de modelos das tarefas envolvidas e os modelos básicos de sincronização e distribuição (Figura 5), detalhando minuciosamente o exemplo da Figura 4.

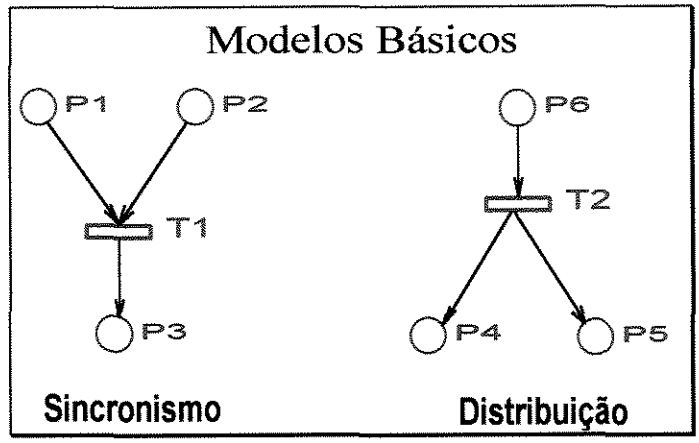


Figura 5 - Modelos básicos de sincronismo e distribuição

2.4 Cálculo de elementos finitos em paralelo

Em se tratando do método dos elementos finitos, o processamento pode ser feito seguindo duas linhas de ação:

- a resolução do sistema linear "$K\,u = f$" por um método direto paralelizado, ou
- a decomposição do domínio e posterior solução do sistema de equações resultante por um método iterativo.

Como o tipo de paralelização proposto na parte computacional deste trabalho baseia-se na decomposição do domínio, tratamos a seguir de algumas técnicas de decomposição de domínio presentes na literatura.

2.4.1 Decomposição do domínio

No nível de discretização do domínios, existem várias possibilidades de utilização de paralelismo. Segundo Carey [1] a mais óbvia maneira é efetuando um particionamento do domínio em subdomínios e posterior distribuição destes entre os processadores. Carey cita um exemplo de geração de malha onde o domínio é dividido em vários subdomínios, sendo que cada processador é encarregado na geração da malha de um dos subdomínios, para isto cada subdomínio tem os pontos no contorno pré definidos.

Em [38], vários são os métodos baseados em decomposição ilustrados. Podemos destacar o trabalho de Roux que descreve um método de decomposição de domínio para

solução de problemas de elasticidade não linear, utilizando uma arquitetura de memória distribuída:

"Cada subdomínio Ω_i é designado para um processador. A descrição do subdomínio é a mesma descrição de qualquer código de elementos finitos padrão. A única diferença se concentra nos nós de interface. Para cada interface entre Ω_i e outro subdomínio Ω_j a lista de nós de Ω_i que pertencem a esta interface deve ser conhecida.

Cada processador pode montar e fatorizar a matriz de rigidez local associada ao seu subdomínio, assim como o seu vetor de carga. A única operação global é o cálculo do salto dos campos locais ao longo da interface.

A implementação consiste em juntar em cada processador os valores do campo local v_i em cada interface $\Gamma_{ij} = \Omega_i \cap \Omega_j$, enviando para o processador responsável por , depois recebendo dados dos subdomínios vizinhos e calculando para cada interface o salto dos valores internos (locais) e externos (recebidos)."

Também em [38] Gropp e Smith apresentam várias implementações do algoritmo de Krylov para decomposição de domínios.

Em [26] Valkenberg et al. citam que o particionamento de malhas estruturadas é muito mais simples que o particionamento de malhas não - estruturadas:

- em malhas estruturadas basta dividir os nós em números aproximadamente iguais para uma boa divisão de carga entre os processadores;

- em malhas não estruturadas é muito mais difícil dividir o domínio de uma maneira ótima. Cada pré condicionador possui suas próprias restrições quanto a decomposição do domínio.

Valkenberg et al. usaram um esquema de particionamento de malhas pelo método de Greedy ou por particionamento multilevel, sempre utilizando o software Metis [39] no particionamento das malhas.

Karypis e Kumar descrevem em [39] o funcionamento do software Metis que é um particionador de grafos não estruturados. A solução de sistemas de equações lineares

do tipo "$A\ x = b$" por métodos iterativos em computadores paralelos originam um problema de particionamento de grafo. Um particionamento ótimo acarretará um menor índice de comunicação entre os processos.

2.5 Métodos iterativos

Barret et al. classificam em [2] como iterativos os métodos que consistem em usar aproximações sucessivas para a obtenção de uma solução satisfatória para um sistema linear. A cada iteração (a cada ciclo) consegue-se uma solução mais próxima da desejada.

Em aplicações de elementos finitos utilizam-se, em geral, os chamados *"métodos diretos"*, que são métodos que baseiam-se numa fatorização da matriz de coeficientes A. Porém, quando A trata-se de uma matriz grande e esparsa, os métodos diretos podem ser menos eficientes que métodos iterativos, baseados apenas em multiplicações envolvendo a matriz A e operações mais simples [22].

Jennings lembra em [21], pag. 187, que a operação básica de todos os métodos iterativos é a seguinte multiplicação:

$$x^{(k+1)} = A\,x^{(k)}$$

Portanto se a matriz A é pouco esparsa, o número de multiplicações por iteração é próximo de n^2. Daí conclui-se que sendo x de dimensão (n x 1) um método iterativo é vantajoso sobre um método direto, somente se o número de iterações necessárias para que se alcance um aproximação aceitável seja menor que $n/3$ (ou $n/6$ se a matriz A for simétrica).

Quando comparados com os métodos diretos, os métodos iterativos apresentam as seguintes vantagens e desvantagens:

Tabela 1 - Vantagens dos métodos iterativos em comparação aos métodos diretos

fonte: JENNINGS (1977), pag. 188

Vantagens	Desvantagens
1. Provavelmente mais eficiente para sistemas de ordem muito grande; 2. Mais simples de implementar, principalmente quando envolve reaproveitamento de área de armazenamento; 3. Pode-se tirar vantagem de uma solução aproximada conhecida; 4. Soluções grosseiras podem ser conseguidas rapidamente; 5. Requer menos espaço para armazenamento; 6. O espaço de armazenamento é facilmente definido a priori.	1. Problemas onde x tem mais de uma coluna não podem ser calculados rapidamente; 2. Mesmo que certa, a convergência pode ser lenta, tornando difícil prever o tempo de cálculo; 3. O tempo de cálculo e a precisão do resultado pode depender da escolha certa de certos parâmetros, que alguns métodos iterativos usam; 4. Se a velocidade de convergência é pequena, os resultados devem ser analisados com cuidado. 5. No caso de A ser simétrica não há ganho no tempo de cálculo, ao passo que para métodos diretos, o tempo pode ser dividido.

Os métodos iterativos dividem-se em *estacionários* e *não estacionários.*

Métodos estacionários: São os métodos que podem ser expresso pela forma

$$x^k = B\,x^{(k-1)} + c$$

onde B e c são independentes da iteração k.

São tidos como métodos estacionários[2][21] os métodos:

- Jacobi;
- Gauss Seidel;
- Successive Overrelaxation (SOR);
- Symmetric Successive Overrelaxation (SSOR).

Métodos não estacionários: Diferente dos métodos estacionários, os métodos não estacionários baseiam-se em informações que mudam a cada iteração para o cálculo da solução aproximada.

São métodos não estacionários os métodos:

- Gradiente Conjugado (CG - *Conjugate Gradient*);
- Minimum Residual (MINRES);
- Symmetric LQ (SYMMLQ);
- Conjugate Gradient on the Normal Equations (CGNE e CGNR);
- Generalised Minimal Residual (GMRES);
- Bi-conjugate Gradient (Bi-CG);
- Quasi-Minimal Residual (QMR);
- Conjugate Gradient Squared (CGS);
- Bi-conjugate Gradient Stabilised (Bi-CGSTAB).

Todos os métodos citados acima são descritos com detalhes em [2].

2.6 O algoritmo de gradiente conjugado

As análises de estruturas pelo método dos elementos finitos, resultam em sistemas do tipo $K\ u = f$, onde K é esparsa e simétrica definida positiva (SPD). O gradiente conjugado é um método de eficiência comprovada em problemas desse tipo ([21], [22], [26]).

Aplicável à sistemas definidos positivos simétricos, o método de gradiente conjugado (vide [2] e [7]) é um dos métodos iterativos de solução de sistemas mais consagrados. Consiste em gerar uma seqüência de vetores de solução aproximada por iteração, um vetor com o resíduo correspondente a solução da iteração e vetores de busca (direções de busca) para atualizar a solução e o resíduo. Por mais que o número de iterações possa crescer, apenas um número pequeno de vetores precisam ser mantidos em memória. Cada iteração necessita de dois produtos internos para calcular escalares que são definidos de forma a garantir que as seqüências satisfaçam certas condições de ortogonalidade, e para sistemas lineares definidos positivos, estas condições implicam que o erro da solução aproximada é minimizado seguindo a norma de energia.

A solução aproximada x_i é somada em cada iteração com um múltiplo do produto do vetor de busca p_i:

$$x_i = x_{(i-1)} + \alpha_i p_i$$

Da mesma forma, o resíduo $r_i = b - A x_i$ é atualizado:

$$r_i = r_{(i-1)} - \alpha_i q_i \text{ onde } \quad q_i = A p_i$$

Escolhendo $\alpha_i = r_i^t r_i / r_{(i-1)}^t r_{(i-1)}$ minimiza-se $r_i^t A^{-1} r_i$ sobre todas as escolhas possíveis para α na equação acima.

As direções de busca são atualizadas usando-se o resíduo

$$p_i = r_i + \beta_{(i-1)} p_{(i-1)}$$

e escolhendo-se $\beta_i = r_i^t r_i / r_{(i-1)}^t r_{(i-1)}$ garante-se que p_i e $Ap_{(i-1)}$ (ou r_i e $r_{(i-1)}$) são ortogonais. Além disso pode-se provar que essa escolha de $beta_i$ faz com que p_i e r_i sejam ortogonais a todos os Ap_i e r_i anteriores, respectivamente.

O algoritmo CG é descrito da seguinte forma [2]:

```
r(0) = b
para i = 1,2,3,...
        ρi-1 = r(i-1)T r(i-1)
        se i = 1
                p(i) = r(0)
        senão
                βi-1 = ρi-1 / ρi-2
                p(i) = r(i-1) + βi-1 p(i-1)
        fimse
        q(i) = A p(i)
        αi = ρi-1 /p(i)T q(i)
        x(i) = x(i-1) + αi p(i)
        r(i) = r(i-1) - αi q(i)
fim
```

2.7 Gradiente conjugado pré condicionado paralelo

O algoritmo de gradiente conjugado pré condicionado que é usado neste trabalho (veja capítulo 6), faz uso de mais um vetor além dos citados em 2.6, é o vetor z que consiste no vetor r pré condicionado por uma matriz M.

Em arquiteturas *message-passing*, Bertsekas e Tsitsiklis propõem em [7] o seguinte esquema de paralelização para o algoritmo CG:

Se n processadores estão disponíveis, o i-ésimo processador fica responsável pela componente *i* dos vetores, ou seja, *x*, *p* e *q*. Os produtos internos destes vetores, utilizados no cálculo de α e β, são feitos deixando o *i*-ésimo processador calcular o produto da componente *i* dos vetores, acumulando parcialmente por processadores para depois fazer o somatório total. Depois os valores dos produtos internos são espalhados para todos os processadores.

Para os produtos "*A p*", o vetor p é espalhado para todos os processadores (todos possuem *p* integralmente), sendo que o *i*-ésimo processador é responsável pelo cálculo do produto de *p* pela linha *i* de *A*.

Em [27], Kincaid e Oppe ilustram, para uma arquitetura de memória compartilhada, uma versão paralela do algoritmo CG com pré condicionador Jacobi

(semelhante ao usado neste trabalho), onde ficam claros os pontos de sincronismo. A seguir transcrevemos o algoritmo CG do item 2.6, inserindo o vetor z (resíduo pré condicionado) e marcando os pontos de sincronismo como o proposto em [27]:

$[r^{(0)}]_s = [b]_s$

para i = 1,2,3,...

$[z^{(i-1)}]_s = [M^{-1}]_s[r^{(i-1)}]_s$

$[\rho^{(i-1)}]_s = [z^{(i-1)}]_s{}^T[r^{(i-1)}]_s$

Sincroniza!!----------

$$\rho^{(i-1)} = \sum_{s=1}^{p} [\rho^{(i-1)}]_s$$

$\beta^{(i-1)} = \rho^{(i-1)} / \rho^{(i-2)}$

Divide!!----------

$[p^{(i)}]_s = [z^{(i-1)}]_s + \beta^{(i-1)} [p^{(i-1)}]_s$

Sincroniza!!----------

Divide!!----------

$[q^{(i)}]_s = [A\ p^{(i)}]_s$

$[\gamma^{(i-1)}]_s = [p^{(i)}]_s{}^T[q^{(i)}]_s$

Sincroniza!!----------

$$\gamma^{(i-1)} = \sum_{s=1}^{p} [\gamma^{(i-1)}]_s$$

$\alpha^i = \rho^{(i-1)} / \gamma^{(i-1)}$

Divide!!----------

$[x^{(i)}]_s = [x^{(i-1)}]_s + \alpha_i\ [p^{(i)}]_s$

$[r^{(i)}]_s = [r^{(i-1)}]_s - \alpha_i\ [q^{(i)}]_s$

fim

O índice s indica o número da subdivisão do sistema que foi dividido em p (número de processadores) divisões. O esquema de divisão dos produtos matriz-vetor e vetor-vetor é o mesmo descrito em [7].

Um sincronismo global é necessário em três pontos:

- Quando é necessário o resultado do primeiro produto escalar;
- Antes do produto matriz-vetor "$q = A\ p$";
- Quando é necessário o resultado do segundo produto escalar.

Capítulo 3

3. Programação de elementos finitos orientada a objeto: O ambiente PZ

Existem aplicações do método dos elementos finitos nas mais diversas áreas, tais como resistência dos materiais, transferência de calor, mecânica dos fluídos, etc. Devido a essa grande aplicabilidade do método, surgiram nas últimas décadas um vasto número de programas para análise de problemas físicos fazendo uso da tecnologia de elementos finitos. A grande maioria desses programas de elementos finitos possuem uma característica em comum: foram desenvolvidos em linguagens estruturadas como FORTRAN, Pascal ou C.

Tratando-se de elementos finitos, o código feito com base na metodologia de programação estruturada, para tratar de um dado problema físico, dificilmente pode ser reutilizado no desenvolvimento de programas que contemplem outros problemas, sem que a complexidade do código aumente a ponto de dificultar a manutenção dos programas.

Diferente da programação estruturada, o paradigma de programação orientada a objetos (*OOP, Object Oriented Programming*), tem como uma de suas vantagens o grande nível de reutilização do código, o que faz com que essa metodologia seja cada vez mais difundida principalmente no desenvolvimento de grandes projetos de software. Para um bom entendimento do presente trabalho, pode-se adquirir maiores conhecimentos sobre OOP em [12].

Desenvolvido por DEVLOO [10][13], o ambiente PZ foi idealizado e desenvolvido em C++, com base no paradigma de programação orientada a objetos. Suas classes base abstraem os três conceitos da formulação matemática do método dos elementos finitos (veja em [13]), são eles:

1. definição da geometria do domínio;

2. definição do espaço de funções de interpolação;

3. definição da equação diferencial que governa o problema e das condições de contorno.

As classes do ambiente PZ definem tipos inerentes de problemas de elementos finitos tais como malha, elemento, nó e material. Cada uma desta classes implementa procedimentos para o tratamento dos dados do tipo que define, por exemplo: integração numérica, montagem do sistema de equações, transformação de coordenadas, etc. Graças a OOP, cada uma das classes do ambiente PZ pode ser estendida de modo a aumentar a abrangência de um programa. Existem classes para o tratamento de problemas de uma ou duas dimensões (com quadriláteros ou triângulos), com interpolação de ordem arbitrária utilizando-se funções hierárquicas.

O ambiente PZ já foi utilizado em modelos matemáticos de diversos problemas físicos, tais como pórticos planos e espaciais, problema de Poisson, placas, cascas, membranas e escoamento em meio poroso (problema de Biot).

3.1 Classes do Ambiente PZ

Como os demais ambientes de programação orientada a objeto, o ambiente PZ é um conjunto de classes que o usuário utiliza no desenvolvimento dos seus programas. Todas as classes do ambiente PZ estão divididas em três categorias a saber:

- classes para aproximação da geometria do problema;

- classes para definição do espaço de interpolação;

- classes para definição dos materiais (equações diferenciais) e das condições de contorno;

Além das classes enquadradas em uma das categorias acima, existem ainda as classes de análise, responsáveis pela organização e controle dos dados da análise (malhas, elementos, nós e materiais), montagem do sistema de equações, definição do tipo de armazenamento da matriz de rigidez, etc. Cabe ao usuário a montagem de um

programa principal, responsável pela criação dos objetos do problema geométrico, do modelo computacional e das condições de contorno, montando em seguida uma análise. Tudo deve ser executado de forma ordenada, como o esquematizado na Figura 6.

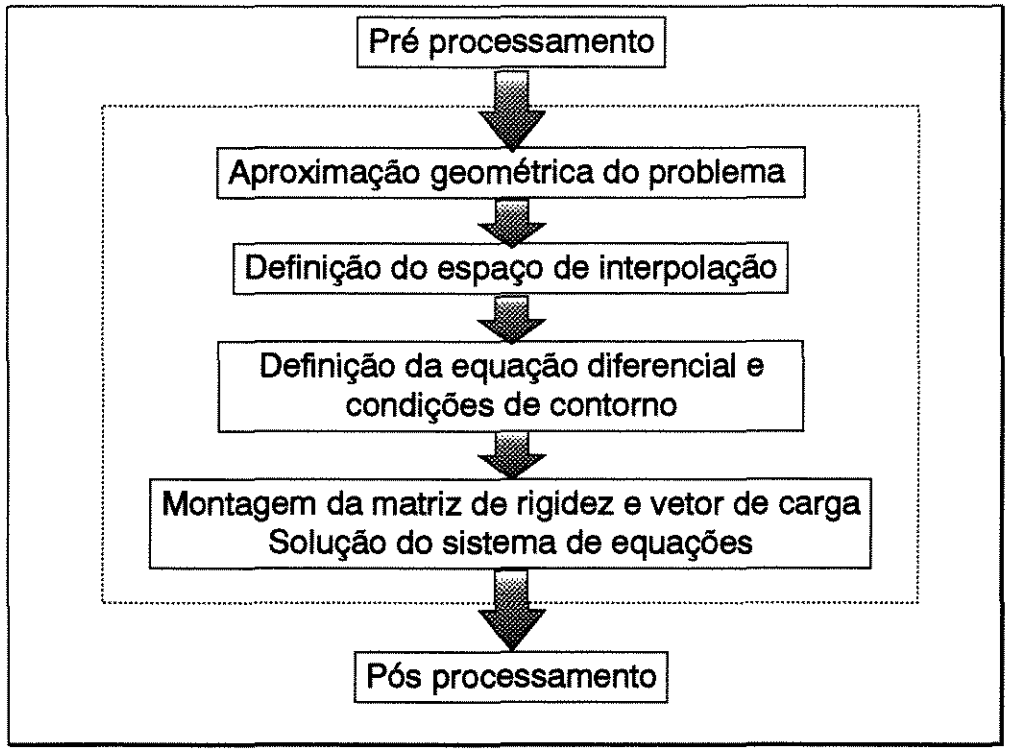


Figura 6 - Seqüência de criação dos objetos PZ

Em problemas bi-dimensionais, pode-se optar por uma discretização por elementos triangulares ou quadrilaterais. Uma mesma discretização pode apresentar comportamento diferente, de acordo com a classe de material associada aos elementos.

Com o ambiente PZ pode-se, para um mesmo domínio, criar diversas análises, variando-se o grau de interpolação, associando-se diversos materiais e tipos de elementos. O que faz do ambiente algo muito flexível.

3.1.1 Pré processamento

Além das classes cobrindo os três conceitos da formulação matemática, o ambiente PZ possui classes para pré processamento e para pós processamento.

Para pré processamento foi implementada uma classe TModulef para leitura de arquivos com descrições de um modelo gerado pelo programa MODULEF[15]. Além de TModulef foi implementada a classe TGenGrid para criação de malhas retangulares. Tanto TModulef como TGenGrid trabalham com elementos triangulares e quadrilaterais.

Já está sendo implementada uma classe para montagem de modelos a partir de problemas analisados com o programa ANSYS[16].

3.1.2 Pós processamento

Para pós processamento o ambiente PZ conta com uma estrutura de classes, chamadas gráficas, para geração de arquivos de entrada de dados (*input*) para programas de visualização científica, como é o caso do IBM Data Explorer ™. O ambiente PZ já conta com classes para geração de arquivos para o Data Explorer (TDXGrafGrid), MView (TMVGrafGrid) e View3D (TV3DGrafGrid)[17][18][19]. Cada uma das classes citadas são derivadas de TGrafGrid, uma classe base abstrata que implementa métodos e descreve as características necessárias para uma classe de pós processamento. Temos a seguinte hierarquia:

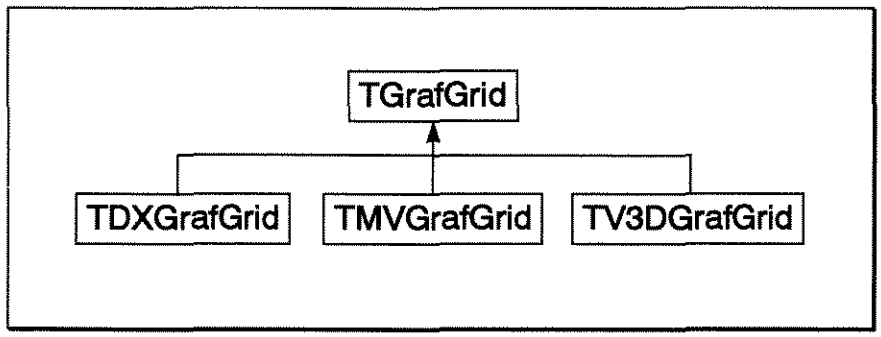


Figura 7 - Classes gráficas - hierarquia

A classe TGrafGrid pode ser estendida para criar novas classes capazes de gerar input para outros programas de visualização.

Com estas classes o ambiente PZ está preparado para gerar arquivos de input referente a análises com dados escalares e/ou vetoriais, que variam com o tempo ou não.

3.1.3 Classes para aproximação da geometria

Num problema de elementos finitos todo o domínio é dividido em elementos, sendo cada elemento caracterizado por uma parametrização responsável por um

mapeamento entre a configuração do elemento no domínio e um elemento mestre indeformado. Tal mapeamento é conseguido com funções de interpolação lagrangeanas lineares e quadráticas.

As classes PZ para aproximação da geometria de um problema de elementos finitos são as seguintes:

- TGeoGrid : malha geométrica;
- TCosys : sistema de coordenadas;
- TGeoEl : elemento geométrico;
- TGeoNod : nó geométrico.

Para cada classe serão descritos apenas os principais métodos e dados.

3.1.3.1 Classe TGeoGrid

A classe TGeoGrid é responsável pelo controle do conjunto de nós e elementos geométricos que definem a geometria do domínio em estudo. É através do objeto TGeoGrid do problema que o usuário tem acesso aos seguintes dados:

- nós geométricos;
- elementos geométricos;
- nós geométricos do contorno;
- elementos geométricos do contorno.

Todos os dados citados acima são armazenados em arvores binárias tipo TVoidPtrMap da GNU C++ Library[3], para garantir um acesso mais veloz ao dado procurado.

A classe TGeoGrid possui ainda um método para determinar a vizinhança dos elementos.

A classe TGeoGrid foi implementada para que, com uma única descrição geométrica do domínio, várias aproximações possam ser feitas. Conclui-se dessa forma que o ambiente PZ está preparado para uma implementação de algoritmos h-adaptativos e/ou multi-grid.

O ambiente PZ possui elemento unidimensional (classe TGeoEl1d), elementos bi-dimensionais quadrilaterais e triangulares (classes TGeoElQ2d e TGeoElT2d, respectivamente), sendo que todos os elementos podem ser quadráticos ou lineares. Todos os elementos do ambiente PZ são derivados de TGeoEl como o esquematizado na Figura 8.

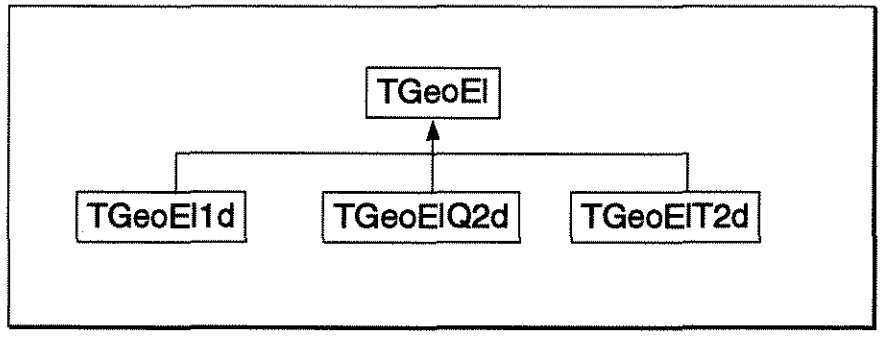


Figura 8 - Hierarquia de elementos geométricos PZ

Principais dados

- static TGeoGrid *gCurrent — É usado em problemas que possuam mais de uma malha geométrica, para determinar qual a malha está geométrica sendo trabalhada.
- TVoidPtrMap fElementMap[2] — conjunto de elementos geométricos (TGeoEl);
- TVoidPtrMap fNodeMap — conjunto de nós geométricos (TGeoNod);
- TVoidPtrMap fElBndCondMap — conjunto de elementos geométricos com condições de contorno;
- TVoidPtrMap fNodBndCondMap — conjunto de nós geométricos com condições de contorno.

[2] TVoidPtrMap é uma classe de árvore binária da GNU C++ Library [3]. Várias são as classes dessa biblioteca citadas no decorrer do texto. Ex.: TIntVec, LongVec, etc.

Principais métodos

- TGeoGrid(int d = 3)

  Construtor, o parâmetro d indica o número de dimensões do problema. Todas as listas são criadas inicialmente vazias.

- TGeoGrid(TGeoGrid &gr)

  Construtor de cópia: todos os dados são iniciados com os dados de gr.

- int NumNodes()

  Retorna o número de nós da malha.

- int NumElem()

  Retorna o número de elementos da malha.

- TVoidPtrMap &ElementMap()

  Retorna uma referência para o conjunto de elementos geométricos (TGeoEl).

- TVoidPtrMap &NodeMap()

  Retorna uma referência para o conjunto de nós geométricos (TGeoNod)

- TVoidPtrMap &ElementBcMap()

  Retorna uma referência para o conjunto de elementos geométricos com condições de contorno;

- TVoidPtrMap &NodeBcMap()

  Retorna uma referência para o conjunto de nós geométricos com condições de contorno.

- TGeoNod *FindNode(long nid)

  Retorna um ponteiro para o nó geométrico cujo id (número de identificação) é igual a nid. Retorna NULL se o mesmo não for encontrado.

- TGeoNod *FindNode(DoubleVec &co)

  Retorna um ponteiro para o nó geométrico mais próxima à coordenada indicada pelo vetor co[3]. Retorna NULL se o mesmo não for encontrado.

[3] Onde co é um vetor do tipo DoubleVec definido na CNU C++ Library, veja [3].

- TGeoEl *FindElem(long elid)

  Retorna um ponteiro para o elemento geométrico cujo id (número de identificação) é igual a elid. Retorna NULL se o mesmo não for encontrado.

- TGeoEl *FindCosys(long cosysid)

  Retorna um ponteiro para o sistema de coordenadas cujo id é igual a cosysid. Retorna NULL se o mesmo não for encontrado.

- void BuildConnectivity()

  Constrói a conectividade dos elementos na malha.

- void GetBoundaryElements(int NodFrom, int NodTo, VoidPtrVec &ElementVec, TIntVec &sides)

  Preenche ElementVec com ponteiros para os elementos no contorno do domínio, entre os nós NodFrom e NodTo, sides é preenchido com o número do lado do elemento que está no contorno em questão.

3.1.3.2 Classe TGeoEl

Classe base abstrata que define métodos e características imprescindíveis aos elementos geométricos. A classe TGeoEl é responsável pelo mapeamento entre o elemento deformado e o elemento mestre. Possui métodos para calcular o Jacobiano do mapeamento, dividir o elemento em sub-elementos, identificação de elementos vizinhos, etc. É responsável pelo armazenamento de informações como número de identificação do material, número de identificação da condição de contorno.

Principais dados

- TGeoGrid *fGrid — malha geométrica à qual o elemento pertence.
- long fId — número de identificação do elemento
- long fMatIndex — número do material do elemento
- VoidPtrVec fNodep — conjunto de nós que constituem o elemento
- VoidPtrVec fConnect — conjunto de elementos vizinhos (conectados ao elemento)
- LongVec fSide — vetor com o número do lado ao qual o elemento em

fConnect está ligado.

- TCompEl *fReference elemento computacional referente ao elemento geométrico em questão.

A seguir serão descritos os métodos de TGeoEl, porém é importante que esteja claro o que o elemento considera como lado, nó, vizinho, etc. Na Figura 9 está ilustrada a idéia de lado, vizinho, nó e malha, para um elemento geométrico quadrático.

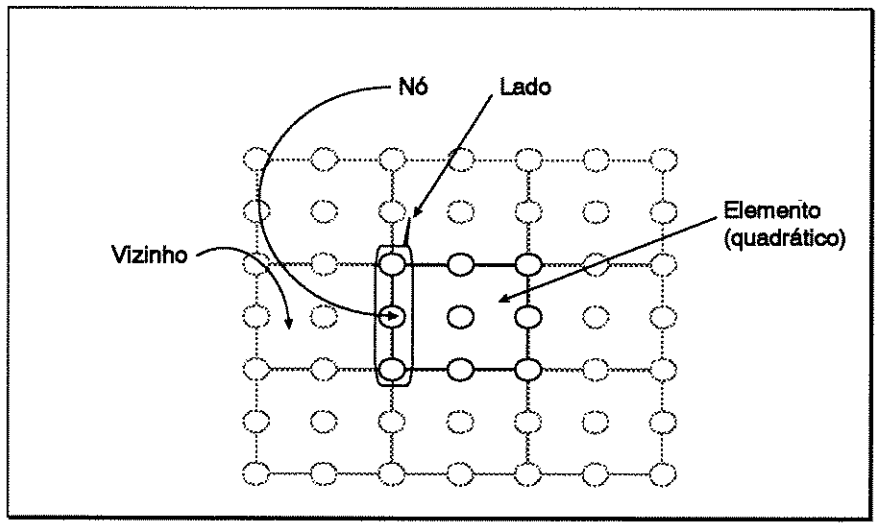


Figura 9 - Ilustração dos dados tratados por TGeoEl

Principais métodos

- TGeoEl(long Id)

  Construtor, sendo Id o número de identificação do elemento. O elemento é criado com os dados sem valores iniciais.

- TGeoEl(TGeoEl &el)

  O elemento é criado inicialmente com os dados de el.

- TGeoGrid *Grid()

  Retorna um ponteiro para a malha da qual o elemento faz parte.

- void SetGrid(TGeoGrid *gr)

  Indica ao elemento que o mesmo faz parte da malha apontada por gr.

- long Id()

  Retorna o id do elemento.

- void SetReference(TCompEl *cel)

  Indica ao elemento que o elemento computacional referente é cel.

- TCompEl *Reference()

  Retorna um ponteiro para o elemento computacional referenciado pelo elemento.

- virtual TGeoNod *SideNode(int side, int node)=0

  Retorna o nó número *node* no lado *side* do elemento.

- void SetSide(int i, int siden)

  Muda o número do lado i para siden. Se siden for menor que zero significa que o lado está no contorno e siden identifica o número da condição de contorno.

- TGeoEl *Neighbour(short int is)

  Retorna o elemento vizinho conectado no lado is.

- void Shape(double x, int n, TFMatrix &phi, TFMatrix &dphi)

  Calcula n funções de forma (unidimensionais) no ponto x, preenchendo phi com os valores da função de forma e dphi com os valores de sua derivada. O ambiente PZ usa funções lagrangeanas lineares e quadráticas em uma dimensão (veja [13]):

Para n = 2:

$$phi = \begin{bmatrix} \frac{1-\xi}{2} \\ \frac{1+\xi}{2} \end{bmatrix} \qquad dphi = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$$

Para n = 3:

$$phi = \begin{bmatrix} \frac{\xi(\xi-1)}{2} \\ 1-\xi^2 \\ \frac{\xi(\xi+1)}{2} \end{bmatrix} \quad dphi = \begin{bmatrix} \xi-\frac{1}{2} \\ -2\xi \\ \xi+\frac{1}{2} \end{bmatrix}$$

- virtual void Jacobian(DoubleAVec &coord, TFMatrix &jac, TFMatrix &axes)[4]

  Calcula o jacobiano do ponto definido por coord colocando-o em jac, sendo os eixos aos quais o jacobiano se refere são colocados em axes.

[4] TFMatrix, e a grande maioria das classes de matrizes citadas neste trabalho, são parte da biblioteca TMatrix [13].

- virtual void X(DoubleAVec &coord, DoubleAVec &result)

  Mapeia um ponto definido por coord no elemento mestre, para o ponto correspondente no domínio deformado, colocando as coordenadas deste em result.

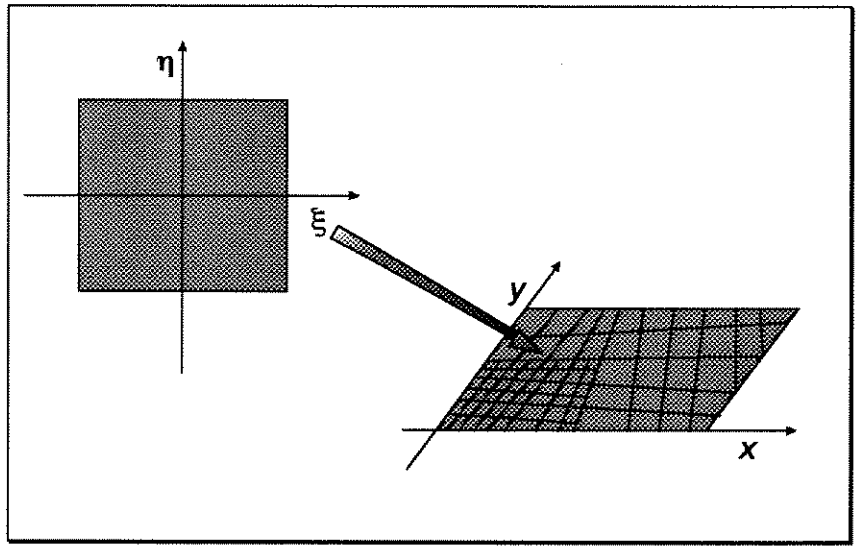


Figura 10 - Mapeamento, do elemento mestre para o domínio

3.1.3.3 Classe TGeoNod

A classe TGeoNod é responsável pela representação do nó geométrico. Possui coordenadas no espaço euclidiano tridimensional e um de seus dados é seu sistema de coordenadas.

Principais dados

- long fId — id do nó
- double fCoord[3] — coordenadas do nó
- TCosys *fSys — sistema de coordenadas
- TDofNod *fDofNod — ponteiro para o nó computacional referente, ou seja, o objeto TDofNod responsável pelos graus de liberdade do nó.

Principais métodos

- TGeoNod(long id, int d = 3, double *xp = NULL, TCosys *ref = NULL)

  Construtor onde id é o número de identificação do nó, d o número de dimensões, xp as coordenadas segundo o sistema global, ref é o sistema de coordenadas adotado para o nó.

- TGeoNod(TGeoNod &gn)

  Construtor de cópia. O objeto é criado com os dados de gn.

- void SetCoord(double *x, int d = 3)

  Atualiza as coordenadas para os valores em x.

- void SetCosys(TCosys *cos)

  Muda o sistema de coordenadas para o apontado por cos.

- void SetReference(TDofNod *dn)

  Indica ao nó que o nó computacional referente é dn.

- TDofNod *Reference()

  Retorna um ponteiro para o nó computacional referenciado pelo nó.

3.1.3.4 Classe TCosys

Define as características e métodos necessários para uma classe de sistema de coordenadas, dessa forma ficam disponíveis para o usuário tantos tipos de sistemas de coordenadas (globais e/ou locais), quantas forem as classes derivadas de TCosys.

O ambiente PZ possui classes para coordenadas cartesianas (classe TCartsys), coordenadas esféricas (classe TEsfersys) e coordenadas cilíndricas (classe TCylinsis).

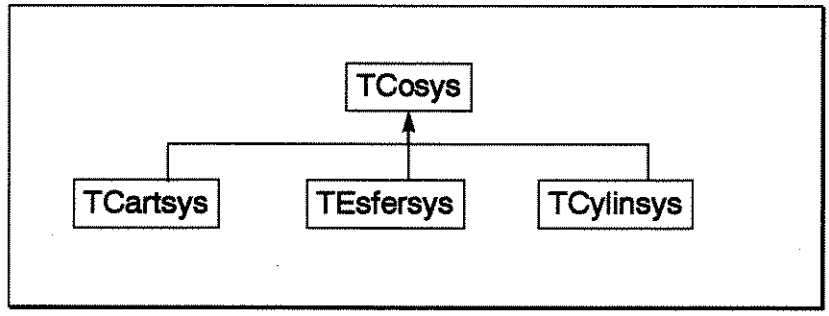


Figura 11 - Classes de sistemas de coordenadas

- TCosys(int num, TCosys *ref = NULL, float *org = NULL)

  Construtor onde num é o id do sistema, ref é a referência do sistema e org a origem.

- void SetReference(TCosys *co, float *org = NULL)

  Atualiza a referência e a origem com os dados passados nos parâmetros co e org.

- void SetOrigin(FloatAVec &org)

  Atualiza a origem para org, sendo org no sistema de referência corrente.

- void SetAxes(FloatAVec &x, FloatAVec &z)

  Define o terceiro eixo (y) perpendicular a x e z.

- void FromReference(float point[3])

  Muda as coordenadas em point para as correspondentes no sistema corrente.

- void ToGlobal(float point[3])

  Muda as coordenadas em point para as correspondentes no sistema global.

- void FromGlobal(float point[3])

  Muda as coordenadas em point para as correspondentes no sistema corrente.

- void ToReference(float point[3])

  Muda as coordenadas em point para as correspondentes no sistema de referência.

3.1.3.5 Classe TGeoNodBc

Trata-se da estrutura que define a condição de contorno aplicada a um nó geométrico.

Dados

- TGeoNod *fNod — nó da condição de contorno
- int fId — id da condição de contorno

Construtor

- TGeoNodBc(TGeoNod *nod, int idbc)

  Cria um objeto com fNod = nod e fId = idbc.

3.1.4 Classes para definição do espaço de interpolação

O ambiente PZ define o espaço de funções de interpolação com base em polinômios ortogonais, gerando funções de interpolação independentes das funções de mapeamento. Por trabalhar com funções de interpolação hierárquicas, o ambiente PZ implementa a chamada adaptatividade tipo p, ou seja, caso ao final de uma análise o erro não for satisfatório, pode-se aumentar o grau de interpolação (veja [13][10]).

Para trabalhar com o mapeamento em uma classe distinta da classe que trata da interpolação, o ambiente PZ utiliza, além do conceito de malha geométrica, um conceito de malha computacional composta por elementos computacionais e nós computacionais. Cada elemento computacional é associado a um elemento geométrico. O elemento geométrico é usado no mapeamento entre o elemento mestre e o domínio, ao passo que o elemento computacional calcula as funções de forma e suas derivadas nos pontos de integração. Logo, o elemento computacional utiliza o elemento geométrico ao qual se refere para montar a matriz de rigidez do elemento.

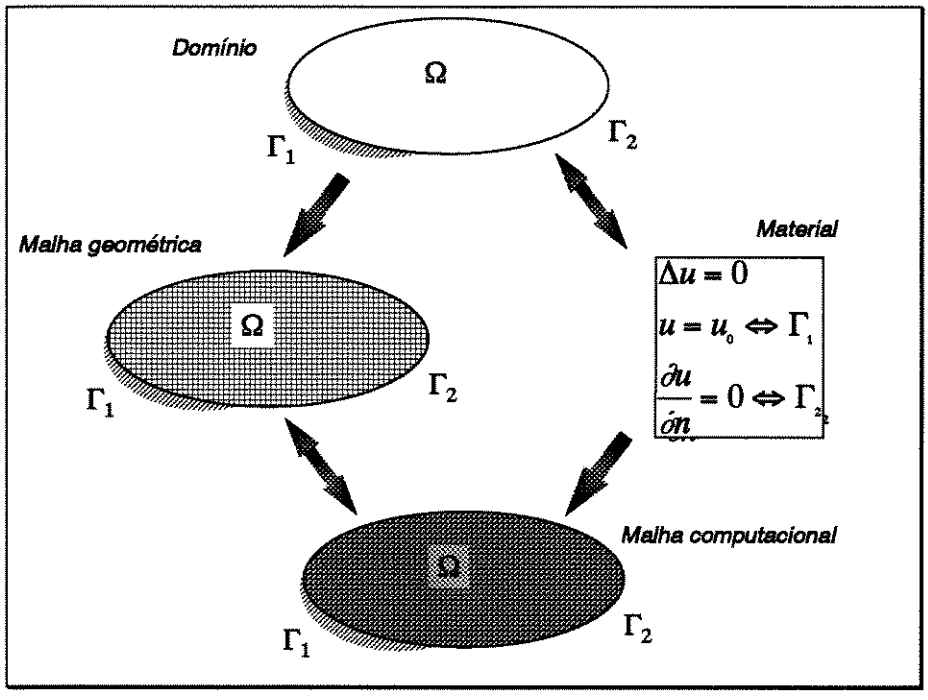


Figura 12 - Problema de elementos finitos com o ambiente PZ

3.1.4.1 Classe TCompGrid

A classe TCompGrid é responsável pelo gerenciamento dos nós e elemento computacionais, materiais e condições de contorno. Para levar o que foi calculado a

nível de elemento para o domínio do problema, a malha computacional necessariamente relaciona-se com uma malha geométrica, pois esta é responsável pelo calculo do jacobiano. Estão implementados na malha computacional, dentre outros, métodos para:

- calcular o número de equações do problema;
- numerar os nós de acordo com o algoritmo de Cuthill-McKee;
- montar a matriz de rigidez e o vetor de carga do problema.

Cada malha computacional refere-se a uma única malha geométrica, todavia uma malha geométrica pode relacionar-se com diversas malhas computacionais, logo conclui-se que a adaptatividade tipo *h*, bem como o método multi-grid, podem ser implementados utilizando-se uma malha geométrica e varias malhas computacionais.

Principais dados

• TGeoGrid *fReference	malha geométrica à qual a malha computacional se refere.
• static TCompGrid *gCurrent	É usado em problemas que possuam mais de uma malha computacional, para determinar qual a malha que está sendo trabalhada.
• TVoidPtrMap fElementMap	conjunto de elementos computacionais (TCompEl);
• TVoidPtrMap fNodeMap	conjunto de nós computacionais (TDofNod);
• TVoidPtrMap fElBndCondMap	conjunto de elementos computacionais com condições de contorno;
• TVoidPtrMap fNodBndCondMap	conjunto de nós computacionais com condições de contorno.
• TVoidPtrMap fBndCondMap	conjunto de condições de contorno;
• TVoidPtrMap fMaterialMap	conjunto de materiais.

Principais métodos

- TCompGrid(TGeoGrid *gr)

  Construtor que recebe como parâmetro a malha geométrica à qual a malha computacional de referenciará.

- TCompGrid(TCompGrid &gr)

  Construtor de cópia: todos os dados são iniciados com os dados de gr.

- int NumNodes()

  Retorna o número de nós da malha.

- int NumElem()

  Retorna o número de elementos da malha.

- int NumMat()

  Retorna o número de materiais da malha.

- int NumBc()

  Retorna o número de condições de contorno da malha.

- TVoidPtrMap &ElementMap()

  Retorna uma referência para o conjunto de elementos geométricos (TGeoEl).

- TVoidPtrMap &NodeMap()

  Retorna uma referência para o conjunto de nós geométricos (TGeoNod)

- TVoidPtrMap &ElementBcMap()

  Retorna uma referência para o conjunto de elementos geométricos com condições de contorno;

- TVoidPtrMap &NodeBcMap()

  Retorna uma referência para o conjunto de nós geométricos com condições de contorno.

- TVoidPtrMap &MaterialMap()

  Retorna uma referência para o conjunto de materiais.

- TVoidPtrMap &BndCondMap()

  Retorna uma referência para o conjunto de condições de contorno.

- TDofNod *FindNode(long nid)

  Retorna um ponteiro para o nó computacional cujo id (número de identificação) é igual a nid. Retorna NULL se o mesmo não for encontrado.

- TCompEl *FindElement(long elid)

  Retorna um ponteiro para o elemento geométrico cujo id (número de identificação) é igual a elid. Retorna NULL se o mesmo não for encontrado.

- void ComputeNodeSequence(long elid)

  Numera os nós segundo o algoritmo de Cuthill-McKee, partindo do elemento cujo id é elid.

- long NumEquations()

  Retorna o número de equações do sistema global.

- long BandWidth()

  Retorna a largura da banda da matriz de rigidez.

- Assemble(TBlock &block, TMatrix &rhs)

  Monta a matriz de rigidez na matriz referenciada por block e o vetor de carga em rhs.

3.1.4.2 Classe TCompEl

Trata-se da classe base abstrata que define métodos e características de um elemento computacional para o ambiente PZ. Em TCompEl estão definidos, dentre outros, métodos para:

definição da ordem de interpolação (p-adaptatividade);

- cálculo das funções de forma;

- cálculo da matriz de rigidez do elemento;

- aplicação das condições de contorno.

Atualmente o ambiente PZ conta com quatro elementos computacionais distintos:

- elemento uni-dimensional (classe TCompEl1d);

- elemento bi-dimensional quadrado (classe TCompElQ2d);
- elemento bi-dimensional triangular (classe TCompElT2d);
- elemento bi-dimensional quadrado para o problema de Biot (classe TElBiot2d).

Os elementos acima seguem a hierarquia esquematizada na figura abaixo:

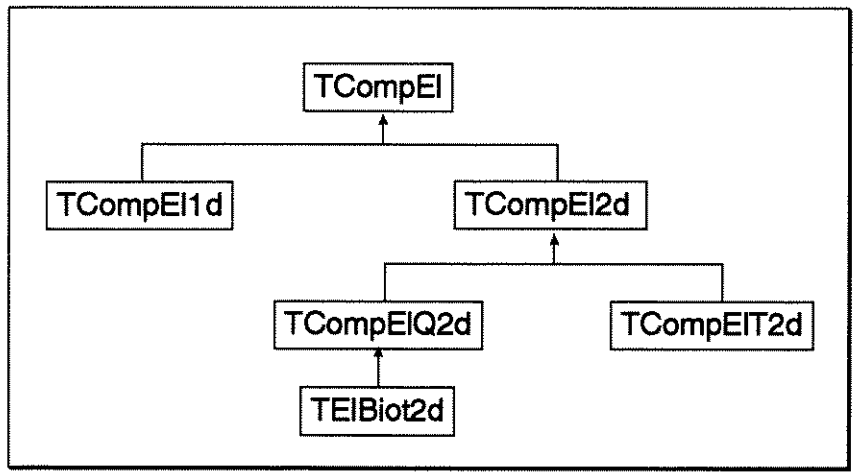


Figura 13 - Classes de elementos computacionais (hierarquia)

O ambiente PZ implementa a p-adaptatividade, pois os elementos utilizam funções de forma com base hierárquica. Cada função de forma é implementada baseando-se nos polinômios de Chebyshev e polinômios lagrangeanos lineares. Uma descrição detalhada da montagem das funções de forma do ambiente PZ pode ser encontrada em [13].

Principais dados

- TCompGrid *fCGrid — malha computacional à qual o elemento pertence.
- long fId — número de identificação do elemento
- TGeoEl *fReference — elemento geométrico referente ao elemento computacional em questão.

Principais métodos

- TCompEl(int dim)

  Construtor. O elemento é criado com os dados sem valores iniciais e dimensão igual a dim.

- TCompEl(TCompEl &el)

  O elemento é criado inicialmente com os dados de el.

- TCompGrid *Grid()

  Retorna um ponteiro para a malha da qual o elemento faz parte.

- void SetGrid(TCompGrid *gr)

  Indica ao elemento que o mesmo faz parte da malha apontada por gr.

- long Id()

  Retorna o id do elemento.

- TGeoEl *Reference()

  Retorna um ponteiro para o elemento geométrico referenciado pelo elemento.

- virtual TDofNod *SideNode(int side, int node)

  Retorna o nó número *node* no lado *side* do elemento.

- TCompEl *Connect(short int side)

  Retorna um ponteiro para o elemento vizinho conectado ao lado side.

- short int SideConnect(short int side)

  Retorna o número do lado do elemento vizinho conectado ao lado side.

- void Chebyshev(double x, int n, TFMatrix &phi, TFMatrix &dphi, long *id)

  Calcula o valor da função de forma unidimensional de Chebyshev no ponto x, sendo n o número de funções a serem calculadas.

- virtual void Shape(DoubleAVec &x, TFMatrix &phi, TFMatrix &dphi)=0

  Calcula as funções de forma no ponto x, preenchendo phi com os valores da função de forma e dphi com os valores de sua derivada.

- virtual void CalcStiff(TElementMatrix &ek, TElementMatrix &ef)

  Calcula a matriz de rigidez do elemento e seu vetor de carga, colocando respectivamente em ek e ef.

3.1.4.3 Classe TDofNod

Define o comportamento do nó computacional do ambiente PZ. Possui métodos para, por exemplo, retornar o número de elementos conectados ao nó, retornar a solução atual associada aos graus de liberdade do nó, etc.

Principais dados

- long fNodeId — id do nó
- DoubleAVec fVar — vetor com os graus de liberdade
- static long gNodeCounter — contador usado para criar um id para o nó.
- TGeoNod *fReference — ponteiro para o nó geométrico referente, ou seja, o objeto TGeoNod responsável pelo mapeamento entre o elemento mestre e o domínio do problema.

Principais métodos

- TDofNod(int ndof, TGeoNod *ref = NULL)

  Construtor onde ndof é o número de graus de liberdade do nó e ref é o nó geométrico de referência.

- TGeoNod *Reference()

  Retorna um ponteiro para o nó geométrico de referência.

- int NumElCon()

  Retorna o número de elementos computacionais conectados ao nó.

3.1.4.4 Classe TDofNodBc

Trata-se da estrutura que define a condição de contorno aplicada a um nó computacional.

Dados

- TDofNod *fNod — nó da condição de contorno
- TBndCond *fBc — condição de contorno

Construtor

- TDofNodBc(TDofNod *nod, TBndCond *bc)

  Cria um objeto com fNod = nod e fBc = bc.

3.1.5 Classes para definição dos materiais e condições de contorno

Na montagem da matriz de rigidez em um problema de elementos finitos, é feita a integral de uma função que tem como argumentos os valores das funções de forma e

suas derivadas. Essa função a ser integrada é definida no ambiente PZ por uma classe derivada de TMaterial, que recebe essa denominação por armazenar os coeficientes que caracterizam o material em um problema de mecânica computacional[13].

Existem também, no ambiente PZ, classes para tratamento das condições de contorno.

3.1.5.1 Classe TMaterial

Classe abstrata que define as características básicas de um material para o ambiente PZ. Como a equação diferencial de um material uni-dimensional difere da equação diferencial dos materiais bi-dimensionais pelo número de derivadas das funções de forma que utiliza, além da classe TMaterial, o ambiente PZ conta com outras duas classes bases: TMat1dLin, base para materiais unidimensionais, e TMat2dLin, base para materiais bi-dimensionais.

O ambiente PZ conta ainda com uma classe para o material do problema de Poisson (classe TPMat), uma classe para problemas de elasticidade (classe TElasticityMaterial) e uma classe para o material do problema de Biot (classe TMatBiot), todas derivadas de TMat2dLin.

Dado

- long fId id do material

Principais métodos

- TMaterial(long id)

  Construtor, o parâmetro id é usado para iniciar fId.

- virtual TBndCond *CreateBc(long idbc, int t, TFMatrix &val1, TFMatrix &val2)

  Cria um objeto da classe TBndCond com o id igual a idbc, com tipo indicado por t e com os valores contidos nas matrizes val1 e val2..

- virtual short int NumVariables()

  Retorna o número de variáveis do material.

3.1.5.2 Classe TBndCond

Estrutura que armazena os dados das condições de contorno utilizadas por um material. Só é possível criar um objeto TBndCond a partir de um material, pois o construtor de TBndCond é privado sendo as classes de materiais declaradas como amigas (*"friend"*) da classe TBndCond (veja classe friend em [12]), o que possibilita o acesso dos materiais aos métodos privados de TBndCond.

Dados

- long fNumber — número da condição de contorno
- int fType — tipo da condição de contorno (Dirichlet, Neuman, misto)
- TFMatrix fBcVal1 — primeira matriz da condição de contorno
- TFMatrix fBcVal2 — segunda matriz da condição de contorno
- TMaterial *fMatPtr — material da condição de contorno

Construtor

- TBndCond(long number, int t, TFMatrix &val1, TFMatrix &val2)
  Construtor onde os parâmetros são os valores iniciais para fNumber, fType, fBcVal1 e fBcVal2 respectivamente.

3.1.6 Montagem e solução do sistema - A classe TAnalysis

O ambiente PZ implementa uma classe de análise com o objetivo de simplificar o manuseio das demais classes do ambiente. Uma vez criados os objetos do ambiente PZ, a classe TAnalysis usa cada objeto e chama os métodos necessários para a montagem e solução do sistema, escolha da matriz de rigidez, etc.

Principais dados

- TGeoGrid *fGeoGrid — malha geométrica
- TCompGrid *fCompGrid — malha computacional
- TMatrix *fStiffness — matriz de rigidez. Por se tratar de um ponteiro do tipo TMatrix podemos usar qualquer classe de

matriz derivada de TMatrix (veja [13]).

- TFMatrix *fRhs — vetor de carga
- TFMatrix *fSolution — vetor solução

Principais métodos

- TAnalysis(TCompGrid *cgrid)

  A análise é criada com os dados de cgrid.

- void SetMatrix(TMatrix *k)

  Usa a matriz k como matriz de rigidez.

- void Assemble()

  Monta a matriz de rigidez e o vetor de carga.

- void Solve()

  Soluciona o sistema de equações.

- virtual void Run(VoidPtrVec &scalnames, VoidPtrVec &vecnames, char *plotfile, ostream &out = cout)

  Chama de forma ordenada cada um dos métodos necessários, desde a montagem das matrizes até a geração do arquivo para pós processamento. O tipo de arquivo de pós processamento é definido pela extensão do nome de arquivo apontado por plotfile. Em scalnames devem estar os nomes das variáveis escalares do material e em vecnames os nomes das variáveis vetoriais. As mensagens são impressas em out.

3.2 Código exemplo

Em [13], M. L. M. Santana detalha linha a linha o código de um programa exemplo para a solução de um problema de elasticidade (classe TElasticityMaterial). O programa exemplo utiliza dois arquivos de entrada, um com a malha, outro com as condições de contorno. A malha do exemplo é lida a partir de um arquivo MODULEF. A seguir transcrevemos o referido código sendo que para uma descrição completa deve-se consultar o capítulo 4 de [13].

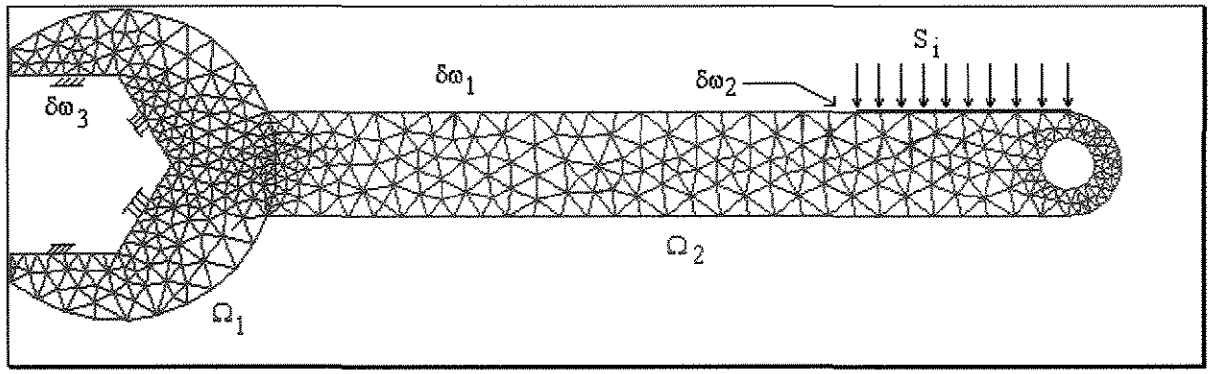


Figura 14 - Problema tratado no código de exemplo - fonte: SANTANA (1997), pag. 107

```
void main() {
      //Criação das malhas
      TGeoGrid *g = new TGeoGrid;
      TCompGrid *c = new TCompGrid(g);
      TGeoGrid::gCurrent = g;
      TCompGrid::gCurrent = c;

      //Leitura dos dados
      TModulef arq("chave.msh");
      arq .Read(*c);
      TAnalysis *a = 0;
      ReadMaterialBC("chave.mat", *c, a);

      //Nomes das variáveis para pós processamento
      VoidPtrVec scalnames(4),vecnames(4);
      scalnames[0] = "Pressure";
      scalnames[1] = "MaxStress";
      scalnames[2] = "SigmaX";
      scalnames[3] = "SigmaY";
      vecnames[0] = "Displacement";
      vecnames[1] = "PrincipalStresses_1";
      vecnames[2] = "PrincipalStresses_2";
      vecnames[3] = "Displacement6";

      //Escolha da matriz de rigidez
      int numeq = (int) a ->NumEquations();
      a ->SetBlockNumber();
      Int band = (int) a ->BandWidth();
      TSBMatrix *s = new TSBMatrix(numeq, band);
      a ->SetMatrix(s);
      a ->Solver().SetDirect(ECholesky);

      //Montagem de K e f, solução e geração dos arquivos de saída
      a ->Run(scalnames, vecnames, "sample.dx");

      //Finalização
      delete g;   delete c;   delete a;
}
```

Capítulo 4

4. Computação paralela orientada a objeto: O ambiente OOPAR

Para trazer os benefícios da programação orientada a objetos ao mundo da computação paralela, foi desenvolvido um conjunto de classes para facilitar a programação de algoritmos paralelos, em especial aqueles voltados à computação científica. Trata-se do OOPAR (*Object Oriented PARallelism*), um ambiente integrado baseado na filosofia de programação orientada a objetos[5]. Com suas classes de gerenciamento e suas classes básicas o OOPAR busca oferecer ao programador de software científico, um ambiente seguro e flexível para execução de tarefas e manipulação de dados, preenchendo dessa forma algumas lacunas deixadas pelas limitações das linguagens convencionalmente usadas.

Cada programa baseado no ambiente OOPAR possui necessariamente três "gerentes", são eles: o gerente de tarefas ou "Task Manager" - **(TM)**, o gerente de dados ou "Data Manager" - **(DM)** e o gerente de comunicação ou "Communication Manager" - **(CM)**. Cada gerente é obrigatoriamente instanciado em cada processo, lembrando que suas classes podem ser derivadas e aprimoradas de acordo com as necessidades da aplicação. Cada gerente pode ser usado em todo o escopo do programa via três ponteiros globais, nomeadamente: **TM**, **DM** e **CM**. Estes ponteiros devem ser iniciados pelo programador antes que qualquer tarefa seja executada, ou qualquer um deles seja usado.

O **TM** possui uma lista de todas as tarefas a serem executadas tanto no processador corrente como nos demais processadores e tem a responsabilidade de executar as tarefas inerentes ao seu processador e enviar as tarefas "alheias" aos seus respectivos processadores.

O **DM** tem o objetivo de prover um espaço único de armazenamento de dados, ou seja, um ambiente de memória compartilhada é emulado pelo **DM** em ambientes de memória distribuída. Cada dado que tiver sua existência "cadastrada" no **DM**, pode ser

usado em qualquer processador. Para isso o **DM** controla o tipo de acesso que cada processador tem ao dado distribuído. Um processador pode ter acesso a leitura ou escrita de um dado e um dado pode ser válido, inacessível, desatualizado, etc.

O **TM** só executa uma tarefa se o estado dos dados que tal tarefa utiliza permitirem isso. Para fazer tal verificação o **TM** usa métodos do **DM**, e de acordo com o resultado a tarefa é ou não é executada. Se o atual estado de um dado não permitir a execução de uma certa tarefa, o **TM** pede para o **DM** tomar providências no sentido de liberar o acesso requerido ao dado, de forma que a execução da tarefa possa então ser consumada. Só exemplificando, se uma tarefa precisa de acesso a escrita o **DM** vai negar o acesso sempre que uma outra tarefa, seja qual for o seu processador, esteja lendo esse dado e, após requisição do **TM,** o **DM** toma providências para que o acesso a leitura de um dado seja suspenso em todos processadores para que, dessa forma, o dado possa ser atualizado pela tarefa que requer acesso a escrita. Todo esse processo será devidamente detalhado mais adiante.

O **CM** é o responsável por mandar e receber dados entre os processadores, usando comunicação assíncrona sempre que possível e compacta o dado que será enviado de um processador para outro. É o **CM** que faz a interface entre o OOPAR e o PVM ou outro padrão de comunicação, como é o caso do MPI.

4.1 Classes e Objetos do OOPAR

Para tirar vantagem dos recursos presentes no OOPAR, cada programa precisa criar objetos de gerenciamento, bem como derivar suas próprias classes das classes base do OOPAR, pois tais classes definem os comportamentos inerentes a objetos geridos pelo ambiente paralelo. Comportamentos estes que permitem que um objeto seja transferido de um processador para outro.

4.1.1 Data Manager (DM) - classe TDataManager

O gerente de dados (**DM**) tem como atribuição principal o controle do uso de objetos distribuídos. Objetos distribuídos são aqueles que podem ser usados em diversos processadores, simultaneamente ou não. Vários processadores podem ter acesso simultâneo para leitura a um dado sob o controle do **DM**, e, no caso de uma atualização desse dado, é o **DM** que negará ao outros processadores o acesso a esse dado até que o

mesmo tenha sido atualizado. Apenas um processador pode atualizar um mesmo dado independente do momento. Todas essas transações são controladas por um rigoroso protocolo o que garante a segurança dos dados. Tal protocolo será descrito mais a diante.

Os principais métodos do **DM** são:

- TDataManager(int *ProcId*): O construtor, responsável pela "inicialização" dos dados da classe no processador indicado por *ProcId*;

- int GetProcID(void): Retorna o número do processador do **DM** em questão;

- long SubmitObject(TSaveable *obj, int trace = 0): Esse método registra o objeto indicado por *obj* junto ao gerente de dados **DM** e retorna um identificador único para todos os processadores. Dessa forma, um objeto pode ser utilizado em qualquer processador bastando para tal que se conheça o ID (identificador) desse objeto. Se *trace* for diferente de zero, o **DM** encarregar-se-á da geração de um arquivo contendo as transações feitas entre os processadores com tal objeto. Esse arquivo pode ser usado na depuração do algoritmo construído usando-se o OOPAR.

- int DeleteObject(long ObjId): Inicia o processo de "destruição" do objeto indicado por *ObjId*. Um objeto não pode, em hipótese alguma, ser arbitrariamente destruído, pois o mesmo pode estar sendo usado em outro processador. Sendo assim, se o presente **DM** é o dono do objeto a ser destruído, uma mensagem é enviada aos demais processadores com acesso a esse dado e o mesmo entra num estado de transição de destruição (*"delete transition state"*). Cada processador que recebe a mensagem de destruição de objeto "*deleta*" o referido objeto. Em seguida, uma mensagem de confirmação (*"delete object confirmation message"*) é enviada como resposta. Só depois que todos os processadores tiverem respondidos à mensagem de destruição de objeto é que o gerente de dados dono do objeto (o que desencadeou o processo) "deleta" o mesmo. Se o **DM** usado para desencadear o processo de destruição do objeto não for o dono do mesmo, uma mensagem de requisição de destruição de objeto é enviada para o **DM** dono.

- int TransferObject(long ObjId, int ProcessorId, long TaskId, MDataState AccessRequest, long Version): Retorna 1 se o objeto indicado por *ObjId* está disponível com o tipo de acesso definido por *AccessRequest* e com a versão igual a *Version* no processador indicado por *ProcessorId*, para a tarefa indicada por *TaskId*. Caso contrário retorna 0 e toma as devidas providências para satisfazer tal requisição, porém não consuma tal requisição. É esse método que requisita uma mudança no estado do objeto indicado por *ObjId*. MDataState é um dado tipo enumerado que pode indicar os seguintes estados:

  ⇒ ENoAccess : O processador não precisa do objeto;

  ⇒ EReadAccess : O processador precisa ter acesso ao objeto para leitura. Como o dado não vai ser modificado, vários processadores podem ter acesso a leitura de um mesmo dado simultaneamente;

  ⇒ EWriteAccess : O processador precisa ter acesso ao objeto para modificá-lo. Se tal processador não é o atual dono do dado, um processo de *"transfer ownership"* é iniciado, pois o processador que pode alterar um dado é o seu dono por definição. Se algum outro processador possui acesso a leitura para esse objeto, uma mensagem de cancelamento de acesso a leitura (*"cancel read access message"*) é enviada para tal processador, que por sua vez toma as devidas providências e responde com uma mensagem de confirmação (*"cancel read access confirmation message"*).

  ⇒ EExclusiveReadAccess: Acesso de leitura que não pode ser interrompida;

  ⇒ EExclusiveWriteAccess: Acesso de escrita que não pode ser interrompida.

- int HasAccess(long ObjId, int ProcessorId, long TaskId, MDataState AccessRequest, long Version) : Retorna 1 se o objeto indicado por *ObjId* está disponível com o tipo de acesso definido por *AccessRequest* e com a versão igual a *Version* no processador indicado por *ProcessorId*, para a tarefa

indicada por *TaskId*. Caso contrário retorna 0 e não toma nenhuma providência;

- void IncrementVersion(long ObjId) : Incrementa a versão do objeto indicado por *ObjId*;

- TSaveable *GetObjPtr(long ObjId) : Retorna um apontador para o dado indicado por *ObjId* se este estiver disponível no presente processador, caso contrário retorna NULL.

4.1.2 Task Manager (TM) - classe TTaskManager

A classe TTaskManager é bem simples. Seu principal propósito é gerir sua fila de tarefas. Os principais métodos públicos são:

- long Submit(TTask *task): A tarefa apontada por *task* entra para a fila de tarefas, um identificador único (*id*) é associado à mesma e retornado;

- int NumberOfTasks(): retorna o número de tarefas na fila deste TM;

- int GlobalNumberOfTasks():retorna o número de tarefas nas filas de todos os TM do ambiente, ou seja, em todos os processadores;

- int ChagePriority(long taskId, int newPriority): Muda a prioridade da tarefa referente a *taskId*. Muda a prioridade mesmo das tarefas de outros processadores;

- int CancelTask(long taskId): Cancela a execução da tarefa referente a taskId e aborta a execução de todas as tarefas que dependam da execução da mesma;

- int ExistsTask(long taskId): retorna 1 caso exista uma tarefa cujo *id* seja *taskId*;

- int TaskProcessor(long taskId): retorna o número do processador onde a tarefa cujo o *id* é *taskId* está sendo executada. Retorna -1 caso está tarefa não exista;

- void AddTaskDependence(long dependent, long execFirst): faz com que a execução da tarefa cujo *id* é *dependent* dependa do sucesso da execução da tarefa referente ao *id execFirst*;

- void Execute(): Tenta executar cada tarefa da fila de execução, uma por vez.

Quando o **TM** (Task Manager: gerente de tarefas) tenta executar uma tarefa são tomadas as seguintes medidas:

- Verifica-se se esta tarefa depende da execução de uma outra. Se existe esse tipo de dependência, verifica-se a existência dessa outra tarefa, caso a mesma não exista mais, essa dependência não mais surtirá efeito;

- Para cada dependência da tarefa a um dado, o Data Manager é consultado para saber se o referido dado encontra-se acessível com o tipo de acesso e a versão necessários. Se esse não é o caso o método *TransferObject* é então chamado com o estado de acesso apropriado.

As tarefas sob o controle do **TM** classificam-se em duas categorias a saber:

- Time Consuming: São as tarefas que podem levar um tempo considerável para a sua execução e podem ser dependentes de versões de dados e/ou da execução de outras tarefas;

- Daemon Tasks: São as tarefas de execução imediata, em geral são as tarefas geradas pelo próprio ambiente OOPAR para enviar mensagens de um processador para outro, sobre o estado de um dado ou tarefa.

4.1.3 Communication Manager (CM) - classe TCommunicationManager

O Communication Manager é o responsável pela transferência dos dados entre os processos, é ele que faz a interface entre o OOPAR e uma determinada biblioteca de passagem de mensagem como é o caso do PVM, MPI, etc. Também existe uma versão que faz toda a comunicação através de arquivos. Esta derivação de TCommunicationManager é muito útil na fase de depuração de algoritmos paralelos, pois identifica-se facilmente quais as mensagens que foram de um processador para outro olhando-se o conteúdo do arquivos de comunicação. Pode-se fazer toda a depuração do algoritmo paralelo usando-se um único processo (thread) onde o ambiente paralelo é simulado, o que possibilita o uso de computadores tipo PC e os diversos pacotes de desenvolvimento existentes para tal plataforma.

Todos os métodos do Communication Manager não precisam nunca ser chamados diretamente pelo usuário, pois lidar com o gerente de comunicações é atividade realizada pelos outros dois gerentes do OOPAR, e os mesmos fazem isso de forma transparente ao usuário. Os métodos públicos são:

- int Initialize(char *processName, int numberOfProcess): Inicia o Communication Manager;
- int GetProcID(): retorna o id do processador corrente;
- int NumProcessors(): retorna o número de processadores em serviço no ambiente OOPAR;
- virtual int SendTask(TTask *pObject): Envia a tarefa *pObject* ao processador indicado na estrutura interna de dados do objeto tipo *TTask*;
- virtual int ReceiveMessages(): Lê as mensagens armazenadas no buffer de recebimento;
- virtual int SendMessages(): Envia as mensagens armazenadas no buffer de envio, descarregando o mesmo;
- virtual int IAmTheMaster(): retorna 1 se o processador corrente e aquele que iniciou o ambiente paralelo, o que não quer dizer que o mesmo e o mestre e os demais são escravos, pois não existe qualquer hierarquia implícita no ambiente OOPAR.

4.1.4 Classe TSaveable

A classe TSaveable deve ser a classe base para qualquer objeto que precise ser transferido de um processador para outro. Dado que o numero de tipos de dados (classes) presentes num programa orientado a objeto pode exceder uma centena, e impraticável conhecer cada um das n estruturas de dados utilizadas. Por esse fato o CM conhece e trata um único tipo de dado, a classe TStorage, e as classes derivadas de TSaveable obrigatoriamente implementam um método capaz de transformar os dados do objeto em TStorage e um outro método capaz de fazer o inverso, ou seja, partindo de um ponteiro para um objeto tipo TStorage, criar um objeto da classe especifica. Resumindo,

todo e qualquer tipo de dado tratado no programa (matrizes, elementos, nós, etc.) deve ser derivado de TSaveable e, necessariamente, apresentar as características supra citadas.

4.1.5 Classe TStorage

Um objeto da classe TStorage armazena todos os dados necessário para uma reconstrução dos objetos nele armazenados numa seqüência de bytes. A única forma imposta à essa seqüência de bytes é que o primeiro dado de cada objeto armazenado deve ser o seu id, o identificador da classe do objeto. É esse id que indicará qual o método que deve ser chamado no momento da reconstrução do objeto após uma transferência de dados entre processadores, partindo-se da seqüência de bytes armazenados num objeto da classe TStorage.

4.1.6 Classe TTask

Toda classe que definir uma tarefa a ser submetida ao **TM** deve ser derivada de TTask, e conseqüentemente, implementar um método Execute() que definirá a tarefa (no sentido literal da palavra) a ser executada, por exemplo a multiplicação de uma matriz por um vetor. A classe TTask implementa métodos que possibilitam um registro das dependências da tarefa sobre versões de dados e/ou execução de outras tarefas, e é baseado nessas dependências que a tarefa pode ou não ser executada. O método Execute() de um objeto TTask sempre retorna um valor que indica sucesso, continuação ou falha (ESuccess, EContinue e EFailure). Caso a execução da tarefa retorne ESuccess, o objeto tarefa é destruído e marcado como terminado. Se for retornado EContinue, o objeto continua na fila do **TM** esperando para ser executado novamente num ciclo subsequente. Se for retornado EFailure, o objeto é excluído da fila do **TM** e marcado como abortado. Caso isto aconteça, todas as tarefas com dependência desta também são abortadas.

Como uma tarefa pode ser transferida de um processador para outro em tempo de execução, a classe TTask é derivada de TSaveable e cada derivação de TTask deve implementar os métodos capazes de transformar o objeto num objeto TStorage e vice-versa.

4.2 Protocolo de Transferência do Data Manager

A transferência de dados entre processos é controlada por um protocolo próprio do Data Manager. Para transferir um dado de um processador para outro é usado o método TransferObject do **DM**, que inicia todo o processo envolvido, sem criar uma fila de requisições. Assume-se que uma nova requisição será feita caso a transferência não se consume, ou seja, se uma tarefa não possui um certo tipo de acesso a um dado objeto, uma nova requisição de acesso é feita a cada ciclo de execução do Task Manager (**TM**).

A ação tomada pelo **DM** quando uma requisição de acesso a um dado é feita pode enquadrar-se em dois casos:

1) **O processador não é dono do dado:** Se um processador não é dono do dado, ele não pode mudar o estado de acesso (access state) do mesmo. Sendo assim, o **DM** manda uma mensagem ao processador dono do dado para que o mesmo tente satisfazer a requisição. (o processador que quer mudar o estado de acesso do dado). Isto é feito da seguinte forma: uma tarefa que será executada no processador dono do dado é criada, essa tarefa quando executada usa o método TransferObject do Data Manager para submeter a requisição.

2) **O processador é dono do dado:** São quatro os diferentes estados de transição possíveis:

   Grant Read Access Transition: Quando um objeto encontra-se com esse estado de transição, o **DM** permitirá apenas acessos do tipo EReadAccess;

   Cancel Read Access Transition: Nenhum acesso ao dado é permitido quando o mesmo estiver nesse estado;

   Transfer Ownership Transition: Nenhum acesso ao dado é permitido quando o mesmo estiver nesse estado;

   No Transition: Qualquer pedido de acesso pode ser atendido, mas antes de permitir o acesso o **DM** toma as seguintes medidas:

Confere-se a versão. Caso o pedido de acesso seja referente à uma versão posterior à versão atual do dado, nada é feito (a função retorna). Presume-se que outra tarefa ainda precisa atualizar o dado;

Se o pedido de acesso é do tipo EReadAccess: Uma mensagem de confirmação do tipo *Grant Read Access* desde que nenhuma outra tarefa permissão de acesso ao dado do tipo EExclusiveWrite. O estado de transferência do dado fica em *Grant Read Access* até que o processador que requisitou o acesso responda com uma mensagem do tipo *Grant Read Access Confirmation.* Se uma tarefa possui direito exclusivo de escrita (EExclusiveWrite) o Data Manager verifica com o Task Manager se a mesma terminou ou não, caso tenha terminado, o estado do dado (MDataState) deixa de ser EExclusiveWrite;

Se o pedido de acesso é do tipo EWriteAccess: Se alguma tarefa tem acesso para leitura exclusiva (EExclusiveReadAccess) o **DM** pergunta ao Task Manager se esta tarefa terminou sua execução ou não. Se ela ainda existir, nada mais é feito e a função retorna (return). Se esta tarefa (com EExclusiveReadAccess) já terminou, termina o acesso de leitura exclusivo. O mesmo é feito para tarefas com acesso de escrita exclusivo. Se algum processador possui acesso de leitura a esse dado, uma mensagem *Cancel Read Access* é enviada a este processador e a função retorna (return). O objeto entra num estado de transição tipo *Cancel Read Access* até que os processadores respondam com uma mensagem de confirmação tipo *Cancel Read Access Confirmation.* Se o pedido de acesso é referente a um outro processador o objeto (dado) é enviado ao outro processador e entra, então, num estado do tipo *Transfer Ownership.*

Se o pedido de acesso é do tipo EExclusiveReadAccess: As mesma medidas tomadas no caso de um pedido de acesso do tipo EReadAccess, e se o pedido é concedido, a tarefa entra para a lista de tarefas com acesso de leitura exclusivo (EExclusiveReadAccess);

Se o pedido de acesso é do tipo EExclusiveWriteAccess: As mesma medidas tomadas no caso de um pedido de acesso do tipo EWriteAccess, e se o pedido é concedido, a tarefa é registrada como a única tarefa com acesso de escrita exclusivo (EExclusiveWriteAccess). Esse tipo de acesso existe para o caso em que é necessário processar um dado que está em um estado de transferência. No caso de uma decomposição de matriz, por exemplo, não faz sentido que uma outra tarefa possa alterar a matriz antes que a mesma tenha sido decomposta.

4.3 Dependência de tarefas sobre versões de dados

No ambiente OOPAR, para que as tarefas envolvidas no algoritmo implementado sejam executadas na seqüência de forma a garantir a consistência dos dados, cada tarefa pode ser "dependente" de uma dada *versão* de um ou mais dados. Quando informa-se ao OOPAR que uma tarefa é dependente de uma versão de um dado, a tarefa só será executada quando tal dado esteja na versão igual ou superior da qual a tarefa depende, isto é, se a tarefa "a" só pode ser executada quando o dado "b" for atualizado 30 vezes, o **TM** só permitirá que a tarefa "a" seja executada quando o **DM** informar-lhe que a trigésima versão de "b" já está disponível.

Com o mecanismo descrito no parágrafo anterior, pode-se criar uma tarefa e submete-la ao **TM** uma única vez, poupando o gasto computacional envolvido na criação e submissão ao **TM** de uma nova instância da classe que define a tarefa. Em contra partida, tal mecanismo implica num controle das versões de cada dado, haja visto que o incremento da versão do dado é de responsabilidade do programador. Se não for informado uma dependência de uma tarefa para uma dada versão de um dado, ou se for informado uma dependência de uma tarefa para um número errado de versão de dado, o **TM** poderá executar uma tarefa num momento errado ou, na maioria das vezes, nunca

executará a tarefa por não haver a versão necessária do dado disponível no ambiente. Estes fatos acarretam horas de depuração que variam de acordo com o grau de complexidade do algoritmo implementado.

É dado a esse problema descrito que o controle das tarefas do algoritmo de gradiente conjugado poderá parecer um tanto quanto complexo. Porém, tal complexidade é justificada pela eficiência, criando-se a maior parte das tarefas uma única vez. Poderíamos criar as tarefas sempre antes de sua execução e fazer com que elas fossem executadas uma única vez. A consistência do algoritmo seria garantida pois cada tarefa seria dependente da execução da sua antecessora, mas esse método demandaria mais processamento, pois na criação de cada tarefa existe uma carga de comunicação entre os processadores envolvidos. Tal método pode e deve ser usado, sem maiores efeitos, para algoritmos relativamente simples.

O registro da dependência de uma tarefa para uma versão de um dado é feito através do método AddDataDepend() da classe TTask. Esse método recebe dois parâmetros:

- o id do dado que a tarefa é dependente;
- o modo de acesso ao dado que a tarefa precisa ter, que pode ser EReadAccess, EWriteAccess, EExclusiveReadAccess ou EExclusiveWriteAccess;
- o número da versão do dado que a tarefa precisa para ser executada, se esse parâmetro for -1 significa que a execução da tarefa não depende da versão do dado, apenas do modo de acesso.

Por via de regra, a dependência de uma tarefa é registrada logo na sua criação, ou seja, logo após a criação de uma tarefa o método AddDataDepend() é chamado uma vez para cada um dos dados aos quais a tarefa possui dependência. Esse procedimento pode ser embutido no construtor da classe da tarefa. Além do momento da criação, as dependências também podem ser atualizadas a cada execução da tarefa.

Capítulo 5

5. Exemplo de organização de programas com o OOPAR

Para garantir a eficiência do código desenvolvido com o OOPAR, deve-se seguir alguns passos desde a definição do algoritmo, até a codificação em si. Podemos citar os seguintes:

- Definição das estruturas de dados utilizadas no algoritmo;
- Definição da distribuição dos dados nos processadores;
- Divisão das tarefas;
- Modelagem do sistema;
- Dependências;
- Codificação.

Para descrever cada um dos passos acima, será usado um exemplo que trata da paralelização do método iterativo de Jacobi para solução de sistemas lineares:

```
r^k = f - (K u^k)
u^(k+1) = u^k + (D^-1 r^k)
onde:
        r = resíduo
        f = Vetor de carga
        K = Matriz de coeficientes;
        u = Solução
        D = diagonal da matriz K
        k = número da iteração
```

5.1 Estrutura e distribuição dos dados

Para solução de sistemas (K u = f) pelo método de Jacobi são usados os seguintes vetores e matrizes:

- K : matriz de coeficientes;
- f : vetor de carga;
- u : vetor solução;
- r : vetor de resíduo;
- D : matriz diagonal.

Resta saber como dividir os dados entre os processadores de forma que todos trabalhem em paralelo de forma que o esforço computacional gasto para a solução do sistema seja reduzido.

Os vetores e matrizes têm a seguinte configuração:

$$K = \begin{bmatrix} x & x & x & x & x & & \\ x & x & x & x & x & & \\ x & x & x & x & x & & \\ x & x & x & x & x & x & x \\ x & x & x & x & x & x & x \\ & & & x & x & x & x \\ & & & x & x & x & x \end{bmatrix}; u = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix}; f = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix}; r = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix}; D = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix};$$

O exemplo acima pode ser dividido em dois processadores da seguinte forma:

$$K = \begin{bmatrix} x & x & x & x & x & & \\ x & x & x & x & x & & \\ x & x & x & x & x & & \\ x & x & x & x & x & x & x \\ x & x & x & x & x & x & x \\ & & & x & x & x & x \\ & & & x & x & x & x \end{bmatrix}; u = \begin{Bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{Bmatrix}; f = \begin{Bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{Bmatrix}; r = \begin{Bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{Bmatrix}; D = \begin{Bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \end{Bmatrix}; NP = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix};$$

onde o que está demarcado pelos retângulos tracejados ficaria em um processador e o que está demarcado pelos retângulos de linha contínua ficaria num outro processador.

Tratamento da interseção entre os dois conjuntos:

Para lidar com os dados na interseção dos dois processadores, ou seja, os dados presentes em ambos os processadores, é feito o seguinte:

1. Os coeficientes da matriz K que estão na parte presente em ambos processadores são divididos, indo apenas uma fração desses coeficientes para cada processador;

2. Cada processador fica responsável por parte das equações e é criado um vetor NP, onde é armazenado o número do processador responsável pela equação;

3. Os valores dos vetores r e u, são atualizados e distribuídos a partir do processador responsável pela equação. Esse processo é feito com base no vetor NP.

NOTA: Com a divisão descrita acima, o resíduo global é obtido somando os resíduos dos diversos processadores.

5.2 Divisão das tarefas

Partindo do algoritmo exposto no início do capítulo, monta-se a configuração das tarefas que compõem a iteração.

Dada a simplicidade da iteração de Jacobi, podemos reescrever o algoritmo acrescentando alguns passos referente a atualização dos dados de um processador nos demais, e também do envio da contribuição do resíduo de um processador para os demais. Vejamos:

```
Tarefa 1.   r_i^k = f - (K_i u_i^k)
Tarefa 2.   envio da contribuição de r_i^k para os demais processadores
Tarefa 3.   u_i^{k+1} = u_i^k + (D^{-1} r_i^k)
Tarefa 4.   envio de u_i^{k+1} para os demais processadores

      onde:
            r = resíduo
            f = Vetor de carga
            K = Matriz de coeficientes;
            u = Solução
            D = diagonal da matriz K
            k = número da iteração
            i = número do processador
```

Cada linha enumerada do algoritmo acima configura uma tarefa envolvida no cálculo paralelo. São quatro as tarefas:

1. **Cálculo do resíduo do processador:**

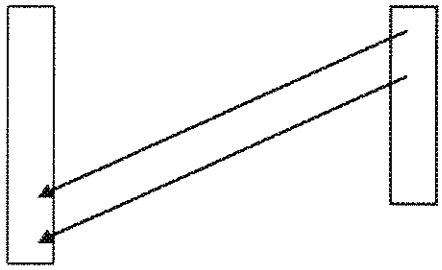
Cada processador calcula a parte que lhe cabe do vetor de resíduo:

Processador 0(r_0): Processador 1(r_1):

2. **Envio da contribuição do resíduo:**

Cada processador envia sua contribuição do resíduo para o processador responsável pela equação:

Processador 0 (r_0): Processador 1(r_1):



3. **Cálculo da solução aproximada:**

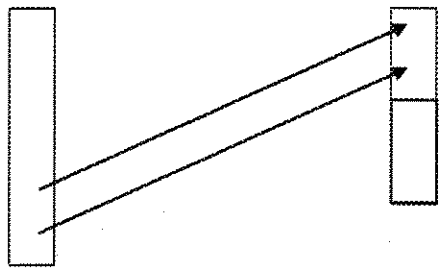
Cada processador calcula a parte do vetor u referente às equações que lhe cabem:

Processador 0 (u_0): Processador 1(u_1):

4. **Envio da solução aproximada:**

Cada processador envia a sua parte do vetor u para os demais processadores:

Processador 0 (u_0): Processador 1(u_1):



5.2.1 Verificação da convergência

As tarefas definidas e descritas até agora compõem a iteração de Jacobi, resta agora definirmos como verificar se a solução aproximada é satisfatoriamente próxima da solução exata, e consequentemente as tarefas devem ser finalizadas.

Para verificar a convergência usaremos como critério a norma euclidiana [21] do vetor de resíduo r:

$\|r\|_e = [(r \cdot r)/n]^{1/2}$

onde n = número total de equações

Quando a norma de r for um número pequeno as tarefas terminam sua execução.

Dado que r é dividido entre os processadores envolvidos, sua norma pode ser calculada de forma paralela. Para tal o produto interno do resíduo ("r . r") é calculado em paralelo, direto na tarefa 1. Cada processador calcula o produto interno da sua parte de r e envia para ser acumulado em um único processador. Assim obtêm-se o produto interno do resíduo global r, a partir da soma dos produtos internos dos resíduos de cada processador.

Finalmente cria-se uma tarefa que utiliza a soma dos produtos internos dos resíduos para calcular a norma do resíduo global r, e efetuar a verificação da convergência. Essa tarefa atualiza uma variável (status) que servirá de sinal para as demais tarefas, indicando se estas devem ou não parar.

O algoritmo paralelo final tem a seguinte forma:

```
Tarefa 1.  se status não é ok
               r_i^k = f - (K_i u_i^k)
               cria tarefa 2
               calcula (r_i . r_i) e cria tarefa 3
           senão
               fim de execução

Tarefa 2.  envio da contribuição de r_i^k para os demais processadores

Tarefa 3.  envio de (r_i . r_i) para ser acumulado em (r . r)

Tarefa 4.  se status não é ok
               u_i^{k+1} = u_i^k + (D^{-1} r_i^k)
               cria tarefa 5
           senão
               fim de execução

Tarefa 5.  envio de u_i^{k+1} para os demais processadores

Tarefa 6.  se ||r||_e pequena
               status = ok
               fim de execução
           senão
               status = continue
```

Figura 15 - Divisão de tarefas para o algoritmo de Jacobi[5]

5.3 Modelagem do sistema

O sistema todo pode e deve ser modelado por um método formal antes de ser codificado. Para o exemplo do algoritmo de Jacobi, foi feito um modelo através de uma rede de Petri [4][11][28] que será analisada a seguir.

[5] Na implementação do algoritmo, a flag status que aparece no esquema, pode receber o valor ECanStop que eqüivalente ao "ok" ou EContinue equivalente ao "continue".

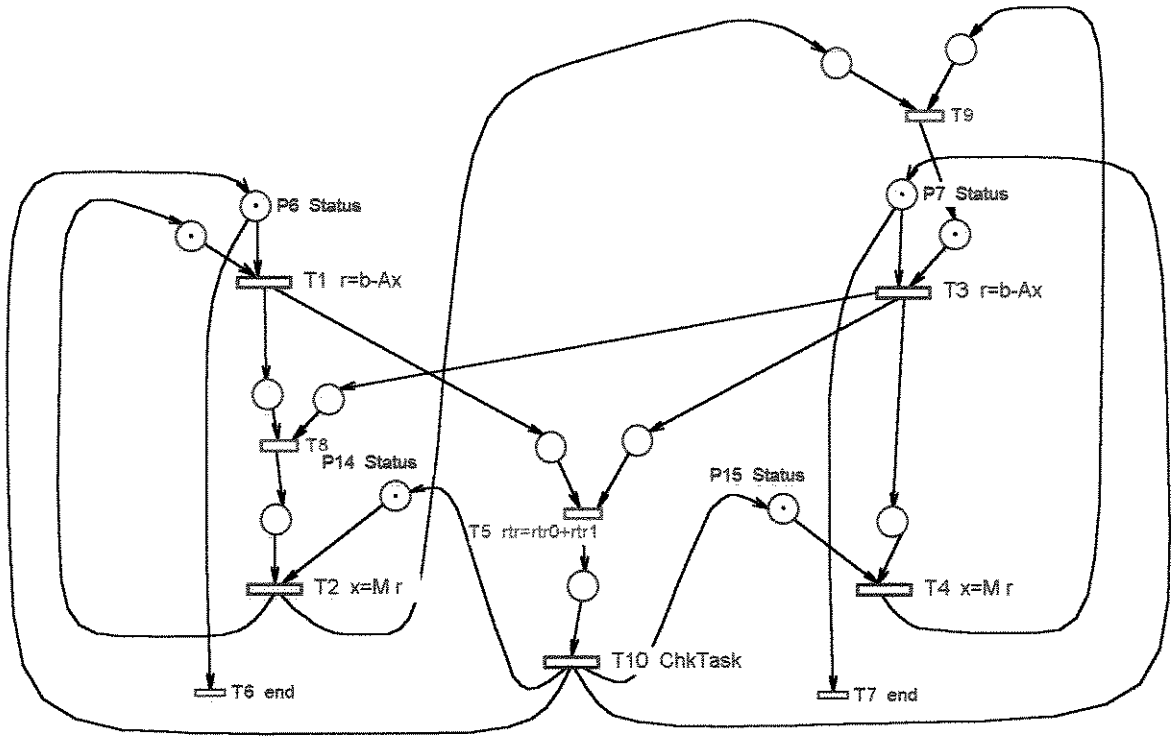


Figura 16 Rede de Petri da iteração de Jacobi para 2 processadores

Na rede de Petri acima está representado o algoritmo de Jacobi para 2 processadores.

Transições

- As transições T1 e T3 representam a tarefa 1 do algoritmo;
- As transições T2 e T4 representam a tarefa 4 do algoritmo;
- A transição T10 representa a tarefa 6 (verificação);
- A transição T5 representa a disponibilidade do produto interno do resíduo global;
- A transição T8 representa a disponibilidade do resíduo já somado à contribuição do processador 1;
- A transição T9 representa a disponibilidade do vetor u (solução aproximada) já com a parte sobre responsabilidade do processador zero.

- As transições T6 e T7 representam o final do algoritmo e só são ativadas caso o status indique isso. A ativação de T6 e T7 implica em um impedimento da ativação das transições T1 e T3 o que faz com que a execução pare.

Dependências

- A transição T1 depende do status fornecido por T10 e do u fornecido por T2;

- Por sua vez a transição T3 depende do status fornecido por T10, da parte de u fornecida por T2 e da parte de u fornecida por T4;

- A transição T4 depende do status fornecido por T10 e do resíduo fornecido por T3;

- Por sua vez a transição T2 depende do status fornecido por T10, da parte de r fornecida por T3 e da parte de r fornecida por T1;

- A transição T10 (verificação da convergência) depende das partes do produto interno do resíduo global que são fornecidas por T1 e T3.

Pode-se esquematizar a rede de Petri do sistema a ser desenvolvido usando um software específico para isso. No caso da Figura 16, a rede foi feita usando um software chamado PESIM[11], capaz de modelar a rede de Petri e simular passo a passo a execução do algoritmo.

5.4 Dependências

Antes da codificação do algoritmo usando o OOPAR, um último passo deve ser cuidadosamente elaborado, a definição do que cada tarefa precisa para ser executada, ou seja, as dependências de cada tarefa.

Trabalhando-se com o OOPAR deve-se determinar qual é o estado que define o momento de execução de cada tarefa. Para algoritmos de execução única, não baseados em iterações, o momento de execução pode ser facilmente determinado associando-se o início de uma tarefa ao final de uma outra. Porém, num algoritmo como o Jacobi, que é executado diversas vezes (as tarefas não são eliminadas a cada execução) até que um certo estado determine o seu final, deve-se determinar o momento da execução de cada tarefa a partir do estado de cada dado. Com o OOPAR, o estado de um dado é

determinado através da sua versão, cada tarefa que altera um dado muda a versão do mesmo, informando dessa forma ao ambiente que o dado já foi alterado.

Com a rede de Petri do sistema a ser desenvolvido em mãos, pode-se definir com exatidão do que cada tarefa depende para ser executada. Mesmo tratando-se de uma rede representando o funcionamento do sistema em dois processadores, podemos extrapolar a análise para o caso genérico de n processadores. Basta considerar que: a transição T5 representa o sincronismo de todas as tarefas e as transições T8 e T9 representam o sincronismo entre o processador e os processadores vizinhos.

Para cada tarefa representada na rede de Petri, observa-se de quais dados a mesma depende e quais dados a mesma altera. Para cada alteração de um dado, associamos um incremento da versão do mesmo, logo as tarefas seguintes da rede são dependentes da versão de saída deste dado.

Na rede do exemplo de Jacobi (Figura 16), as transições T5, T8 e T9 não estão associadas a uma tarefa a ser desenvolvida, mas sim ao estado de um dado. Por outro lado, as tarefas encarregadas de levar o dado de um processador para outro não estão representadas na rede mas deverão ser implementadas, pois as transições T5, T8 e T9 representam o final dessas atualizações.

Vejamos para cada uma das 5 tarefas quais são suas dependências:

Dados alterados a cada ciclo:

- Status — Indica se uma aproximação aceitável foi encontrada;
- r_i — resíduo do processador
- u_i — solução aproximada do processador
- rtr_i — produto interno do resíduo do processador
- rtr — somatório dos produtos dos resíduos dos processadores (produto interno do resíduo global)

Tarefas e dependências:

Tarefa	Dado	Acesso	Versão
TTask1	Status	EReadAccess	nc
	r_i	EWriteAccess	-
	u_i	EReadAccess	nc * NumRet
TTask2[6]	r_k	EWriteAccess	-
TTask36	rtr	EWriteAccess	-
TTask4	r_i	EReadAccess	nc * NumContrib
	u_i	EWriteAccess	-
TTask56	u_k	EWriteAccess	-
TTask6	rtr	EReadAccess	nc * NumProcessors
	Status	EWriteAccess	-

onde:

- i : número do processador
- k : número do processador de destino
- nc : número da iteração
- NumProcessors : número de processadores
- NumContrib : número de processadores que contribuem com o cálculo de r_i
- NumRet : número de processadores envolvidos no cálculo de u_i

Além das dependências acima, cada tarefa deve registrar também a dependência de acesso a leitura dos dados que não são alterados, tais como a matriz K, o vetor NP, etc.

[6] Tarefa executada uma única vez. Criada e submetida por uma tarefa de execução contínua.

5.5 Codificação

A codificação consiste em criar uma derivação de TTask para cada tarefa do algoritmo. O método Execute() de cada tarefa implementa a parte do algoritmo a ser trabalhada e atualiza suas dependências usando o método AddDataDepend de TTask. Cada tarefa tem como dados os ids dos vetores e coeficientes envolvidos na sua parte do algoritmo. Além disso cada tarefa pode obter os ids dos dados de outros processadores.

Para que cada processador possa saber os ids dos dados dos demais, estes ids são guardados em estruturas chamadas TGlobalData, submetidas ao **DM**, e os ids dessas estruturas são armazenados num vetor, DataIdVec. Tanto as estruturas TGlobalData como o vetor DataIdVec, são espalhados para todos os processadores.

Cada estrutura TGlobalData possui os seguintes dados:

- int fProcNumber : número do processador;
- long fNPID : id do vetor NP;
- long fKID : id da matriz K_i;
- long fUID : id do vetor u_i;
- long fFID : id do vetor f_i;
- long fRID : id do vetor r_i;
- long fGInProdId : id da variável onde o produto interno do resíduo é acumulado;
- long fGStatusId : status que indica se o algoritmo convergiu ou não;

Para o algoritmo de Jacobi definiu-se as classes TTask1, TTask3, TTask4 e TTask6 para respectivamente cada uma das tarefas 1, 3, 4 e 6. Mais uma classe TTask2 que pode atualizar u (tarefa 5) ou r (tarefa 2).

5.5.1 Classe TTask1

```
Tarefa 1.  se status não é ok
           r_i^k = f - (K_i u_i^k)
           cria tarefa 2
           calcula (r_i . r_i) e cria tarefa 3
       senão
           fim de execução
```

A classe TTask1 (veja Figura 15) é criada e submetida ao **TM** uma única vez, sendo que só termina sua execução quando a flag Status for igual a ECanStop.

Dados:

- long fVersion — número da iteração;
- long fDataId — id da estrutura TGlobalData do processador onde a tarefa TTask1 reside;
- long fDataIdVecId — id do vetor com os ids das demais estruturas TGlobalData.

Método Execute:

- Usando fDataId recupera-se a estrutura TGlobalData para obter-se os ids de K, f e r;
- Com os ids de K, f e r usa-se o método GetObjPtr do **DM** para conseguir um ponteiro para cada um desses dados;
- Executa-se então o cálculo do resíduo ($r_i = f_i - K_i u_i$);
- Para cada um dos processadores, monta-se um vetor com as posições de r_i que devem ser enviadas e cria-se um objeto TTask2 submetido ao **TM** em seguida.
- Efetua-se o cálculo do produto interno de r_i, rtr_i, e uma tarefa TTask3 é criada para acumular rtr_i na variável global rtr;
- A versão de r_i é incrementada;

- fVersion é incrementada, e registram-se as dependência para o próximo ciclo, usando fVersion para o cálculo das novas versões das quais a próxima execução depende;

- Se Status é igual a ECanStop retorna-se ESuccess, caso contrário retorna-se EContinue.

5.5.2 Classe TTask2 (tarefas 2 e 5)

Tarefa 2. envio da contribuição de r_i^k para os demais processadores
Tarefa 5. envio de u_i^{k+1} para os demais processadores

A classe TTask2 é criada e submetida ao **TM** em cada execução de TTask1 (veja Figura 15), para atualizar o r dos demais processadores, e em cada execução de TTask4, para atualizar u.

Dados:

- long fDestObjId : id do vetor de destino;

- TFMatrix fR : matriz com os valores para serem somados ou atribuídos;

- int fIsU : se 1 fR é atribuído no vetor de destino, senão é somando;

Método Execute:

- Com o id fDestObjId usa-se o método GetObjPtr do **DM** para conseguir um ponteiro para o vetor de destino;

- Se fIsU for verdadeiro, o vetor fR é atribuído no vetor de destino, do contrário é somado;

- A versão do vetor de destino é incrementada com o método IncrementVersion() do **DM**;

- A execução termina retornando-se ESuccess.

5.5.3 Classe TTask3

```
Tarefa 3.  envio de (r_i . r_i) para ser acumulado em (r . r)
```

A classe TTask3 (veja Figura 15)é criada e submetida ao **TM** em cada execução de TTask1, para acumular o produto interno de r_i em uma variável global.

Dados:

- long fGInProdId — id da variável global onde é acumulado o produto interno do resíduo;
- double fInProd — valor para ser acumulado.

Método Execute:

- Com o id fGInProdId usa-se o método GetObjPtr do **DM** para conseguir um ponteiro para a variável global;
- Soma-se nessa variável o valor fInProd ("r_i . r_i");
- A versão de fGInProdId é incrementada com o método IncrementVersion() do **DM**;
- A execução termina retornando-se ESuccess.

5.5.4 Classe TTask4

```
Tarefa 4.  se status não é ok
               u_i^{k+1} = u_i^k + (D^{-1} r_i^k)
               cria tarefa 5
           senão
               fim de execução
```

A classe TTask4 é criada e submetida ao **TM** uma única vez, sendo que só termina sua execução quando a flag Status for igual a ECanStop (Figura 15).

Dados:

- long fVersion — número da iteração;
- long fDataId — id da estrutura TGlobalData do Data Set sobre o qual a tarefa

TTask4 vai atuar;

- long fDataIdVecId — id do vetor com os ids das demais estruturas TGlobalData.

Método Execute:

- Usando fDataId recupera-se a estrutura TGlobalData para obter-se os ids de u, D e r;
- Com os ids de u, D e r usa-se o método GetObjPtr do **DM** para conseguir um ponteiro para cada um desses dados;
- Executa-se então o cálculo da solução aproximada ($u_i = D^{-1} r_i$);
- Para cada um dos processadores, monta-se um vetor com as posições de u_i que devem ser enviadas e cria-se um objeto TTask2 submetido ao **TM** em seguida.
- A versão de u_i é incrementada;
- fVersion é incrementada, e registram-se as dependência para o próximo ciclo, usando fVersion para o cálculo das novas versões das quais a próxima execução depende;
- Se Status é igual a ECanStop retorna-se ESuccess, caso contrário retorna-se EContinue.

5.5.5 Classe TTask6

```
Tarefa 6.  se ||r||∞ pequena
               status = ok
               fim de execução
           senão
               status = continue
```

A classe TTask6 é criada e submetida ao **TM** uma única vez, sendo que só termina sua execução quando a norma do resíduo global r for menor que um dado valor. Só é instanciada uma vez e roda no processador 0 onde o produto interno do resíduo é acumulado (Figura 15).

Dados:

- fNumProcessors número total de processadores;
- fGStatusId id da variável Status;
- fGInProdId id da variável onde é acumulado o produto interno do resíduo (GInProd);
- fVersion número da iteração;
- fNumEq número de equações do problema.

Método Execute:

- Usando fGInProdId usa-se o método GetObjPtr do **DM** para conseguir um ponteiro para GInProd;
- Executa-se então o cálculo da norma do resíduo:

  normr = Sqrt(GInProd/fNumEq);

- Se normr for menor que uma dada constante, muda-se o valor de Status para ECanStop, senão Status continua com EContinue);
- Incrementa-se fVersion e, usando o método AddDataDepend, registra-se que a próxima execução de TTask6 depende agora de mais NumProcessors atualizações de GInProd;
- A versão de Status é incrementada;
- Se Status é igual a ECanStop retorna-se ESuccess, caso contrário retorna-se EContinue.

Capítulo 6

6. Gradiente conjugado pré condicionado: Versão paralela com o OOPAR

Para implementar o algoritmo paralelo de gradiente conjugado pré condicionado [2] [7], usando os recursos e facilidades do OOPAR, fez-se necessária uma subdivisão do algoritmo em tarefas. Para cada tarefa foi feita uma classe derivada de TTask com métodos para transformar o objeto em TStorage e criar uma tarefa partindo de um ponteiro para TStorage, além de, é claro, um método Execute() onde é procedido uma parte do algoritmo de gradiente conjugado paralelo. Ao todo foram criadas onze tarefas diferentes, porém são, ao todo, dez classe diferentes pois uma das classes implementa duas tarefas, dado a semelhança existente entre ambas. Os detalhes serão tratados no decorrer da descrição das classes.

Parte das tarefas são executadas de forma continua, ou seja, são criadas logo de início e permanecem na fila de execução do Task Manager até que uma solução aceitável seja alcançada (convergiu). Uma outra parte das tarefas são tarefas de atualização de vetores de processadores vizinhos ou coeficientes globais (únicos em todo o ambiente paralelo) e são criadas e submetidas ao **TM** pelas tarefas de execução contínua. O método Execute() destas tarefas de atualização de dados retorna ESuccess, logo estas tarefas são removidas da fila do **TM** e destruídas, logo após sua execução. Sendo assim as tarefas de execução continua "vivem" durante quantos ciclos forem necessários até que a solução seja alcançada, ao passo que as tarefas de atualização "nascem" e "morrem" a cada ciclo do algoritmo.

A tarefa TCGStartTask cria e submete todas as tarefas continuas (exceto a tarefa que verifica a convergência). Essa tarefa é necessariamente a primeira tarefa do algoritmo de GC e é destruída e removida da fila do **TM** logo após sua execução.

6.1 Divisão das Tarefas

O algoritmo de GC pré condicionado, proposto em [2], que serviu de base para todo o desenvolvimento foi o seguinte:

```
M z = r
ro = r^t z
beta = ro/ro_(i-1)
se i = 0
      p = z
senão
      p = z + beta p_(i-1)
fimse
q = A p
alfa = ro/p^t q
r = r_(i-1) - alfa q
x = x_(i-1) + alfa p
se convergiu para, senão repete
```

onde:

M : matriz pré condicionante;

z : resíduo pré condicionado;

r : resíduo;

ro : r . z;

beta : razão entre o ro atual e o ro do ciclo anterior;

p : vetor de busca;

A : matriz de coeficientes;

q : produto de A por p;

alfa : ro dividido por p. q;

x : solução aproximada.

O algoritmo acima é dividido da seguinte forma:

```
Tarefa 1
      M z = r
      ro = r^t z

Tarefa 2
      beta = ro/ro_(i-1)
        se i = 0
              p = z
        senão
            p = z + beta p_(i-1)
      fimse

Tarefa 3
        q = A p

Tarefa 4
      alfa = ro/p^t q
        r = r_(i-1) - alfa q

Tarefa 5
      x = x_(i-1) + alfa p

Tarefa 6
      verificação da convergência
```

Até aqui pouco poderia ser executado em paralelo. Para ser mais exato, apenas a tarefa 5 poderia ser executada simultaneamente as tarefas 4 e 6, pois pode-se notar que cada uma das outras tarefas são dependentes da execução das tarefas predecessoras. Dessa observação pode-se concluir que o trabalho com um único conjunto de dados, não proporciona um alto grau de paralelização.

6.1.1 Divisão dos dados em Data Sets

Para elevar o grau de paralelização do algoritmo empregado, fez-se necessária a divisão dos dados em conjuntos chamados Data Sets[1]. Como o CG é um algoritmo composto por produtos matriz - vetor e vetor - vetor [2][7][21][22][26], todas as matrizes e vetores são divididos em partes menores, e cada parte atribuída a um Data Set, ou seja, cada Data Set é responsável por uma parte das equações do problema. Dessa forma, associando-se cada Data Set a um processador, todo o problema é tratado paralelamente pelos processadores. Todos os processadores tratam simultaneamente do grupo de equações associado a cada um deles. Com isso os produtos matriciais são calculados de forma paralela.

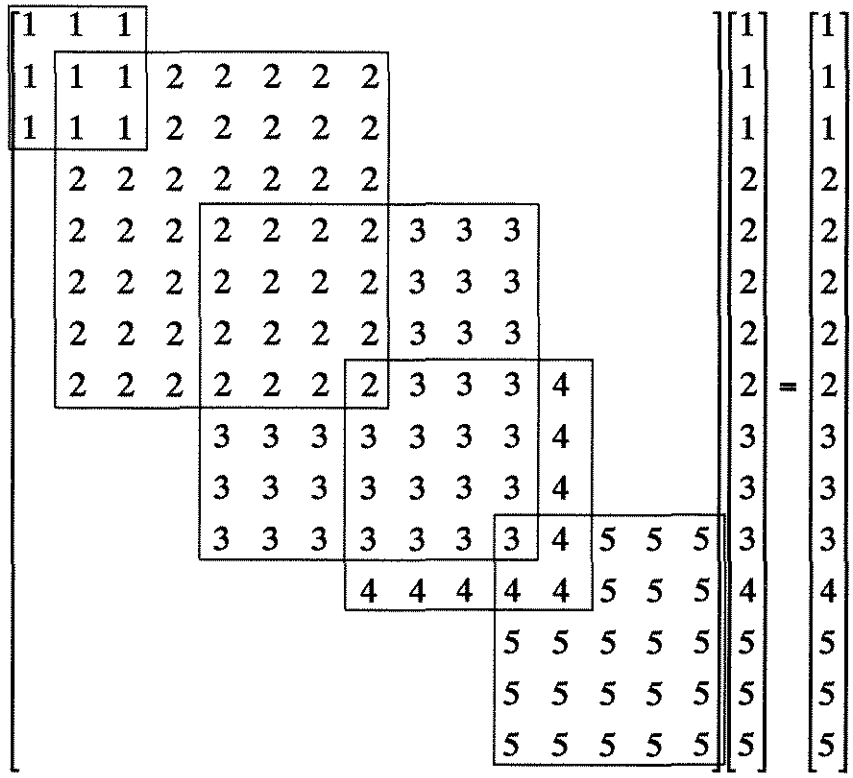


Figura 17 - Exemplo de divisão de um sistema $Kx = f$ com 15 equações em 5 Data Sets[7]

Para o exemplo da Figura 17, o Data Set de número três ficaria com o seguinte sistema $K_3x_3=f_3$:

$$\begin{bmatrix} 2 & 2 & 2 & 2 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix}$$

Figura 18 - Parte do sistema global de equações referente ao Data Set 3

Analisando a Figura 17 e a Figura 18, nota-se que o resíduo calculado para as quatro primeiras equações do Data Set 3 vão contribuir para o cálculo das quatro últimas equações do Data Set de número 2. Nota-se ainda que os Data Sets 4 e 5 vão contribuir para o cálculo do resíduo da última equação do Data Set 3.

[7] Os números nas posições das matrizes e vetores indicam o número do Data Set "dono" da equação. Nas partes da matriz de coeficientes onde há sobreposição de Data Sets, apenas uma fração do coeficiente vai para cada Data Set.

Cada Data Set deve saber quais dos demais Data Sets necessitam dos seus resultados e, da mesma forma, deve saber de que outros Data Sets ele necessita nos cálculos dos seus próprios resultados. Isto é feito tendo como base uma distribuição como a do exemplo da Figura 18, onde nota-se que cada equação local de um Data Set deve ser "mapeada" para uma outra equação local de um outro (ou mais) Data Set, quando existem mais de um Data Set envolvido no cálculo de uma mesma equação do problema global. Foi criado então uma classe especial para gerir esse "mapeamento", essa classe chama-se TMapManager e será descrita mais adiante. Com mais esse gerente garante-se que, num caso como o do exemplo, as posições onde não existem números (onde estariam zeros), não ocuparão espaço algum na memória de qualquer um dos processadores envolvidos. No caso do exemplo as equações de cada Data Set são contíguas, porém com a classe TMapManager os Data Sets poderão ser formados por equações não necessariamente contíguas no sistema global.

Tabela 2 - Dimensões das matrizes e vetores de cada Data Set do exemplo

Data Set	**A**	**x**	**r**	**z**	**p**	**q**
1	3x3	3	3	3	3	3
2	7x7	7	7	7	7	7
3	7x7	7	7	7	7	7
4	5x5	5	5	5	5	5
5	5x5	5	5	5	5	5

A divisão dos dados em Data Sets possibilita a criação e execução de um conjunto de tarefas, como o anteriormente citado, para cada processador disponível. Porém deve-se adicionar algumas tarefas para fazer a atualização tanto dos coeficientes, que são dados únicos para todos os processadores, como dos vetores (resíduos) inerentes a cada processador.

A divisão em tarefas do algoritmo GC pré condicionado tomou então a seguinte forma:

```
Tarefa 1
        M^k z^k = r^k
        ro^k = (r^k).(z^k)
          ->    Tarefa 2
                      soma ro^k ao ro

Tarefa 3
        beta = ro/ro(i-1)
        se i = 0
                p^k = z^k
        senão
                p^k = z^k + beta p^k(i-1)
        fimse
          ->    Tarefa 4
                      atualiza p nos demais processadores

Tarefa 5
        q^k = A^k p^k
          ->    Tarefa 6
                      atualiza q nos demais processadores
          ->    Tarefa 7
                      calcula contribuição ptq^k = (p^k).(q^k)
                ->    Tarefa 7a
                            soma ptq^k ao ptq global

Tarefa 8
        alfa = ro/p^tq
        r = r(i-1) - alfa q
          ->    Tarefa 9
                      soma (rtr = (r^k). (r^k)) a rtr

Tarefa 10
        x^k = x^k(i-1) + alfa p^k

Tarefa 11
        verificação da convergência
```

Figura 19 - Divisão de tarefas do algoritmo PCG paralelo

onde o índice k indica o número do Data Set e rtr, ptq, ro e $ro_{(i-1)}$ são coeficientes globais.

Deve ressaltar-se que cada processador (Data Set) está sujeito a configuração de tarefas acima descrita.

Com a divisão em Data Sets, o produto global é dividido em produtos matriz vetor que são feitos localmente em cada Data Set. Isto significa que:

- O Data Set precisa dispor de valores que não pertencem a ele;

- O Data Set vai contribuir para parte do vetor resultado, incluindo equações que não pertencem a ele.

Por isso, a estrutura de dados associada a cada Data Set contém informações referentes a:

1) **dados do algoritmo CG:**

z : parte do resíduo pré condicionado tratada pelo Data Set

r : parte do resíduo tratada pelo Data Set

p : parte de p tratada pelo Data Set

x : parte da solução tratada pelo Data Set

A : parte da matriz de coeficientes tratada pelo Data Set

q : parte de q tratada pelo Data Set

d : parte da diagonal da matriz A tratada pelo Data Set

2) **dados de controle do algoritmo paralelo:**

DataNumber : número do Data Set;

ProcNumber : número do processador onde o Data Set vai residir;

NP : vetor com tamanho igual ao número de equações do seu Data Set, onde é armazenado o número do Data Set "dono" de cada equação, ou seja, aquele que é responsável pela equação;

GNodeNumber: vetor com o mesmo tamanho de NP, onde é armazenado o id de cada equação para o problema global. Esse número é importante para a troca de dados entre os Data Sets;

Cada Data Set utiliza ainda os seguintes dados globais:

1) **dados do algoritmo CG:**

ro : r . z

$ro_{(i-1)}$: ro do ciclo anterior

ptq : produto p . q;

rtr : produto interno do resíduo

2) **dados de controle do algoritmo paralelo:**

DSetProc : vetor com tamanho igual ao número de Data Sets em que o problema global foi dividido, contém o número do processador onde cada Data Set vai residir;

Status : flag indicando se deve haver um novo ciclo

6.2 Classe TMapManager

Responsável pelo mapeamento das equações locais de um Data Set para as equações locais dos Data Sets vizinhos.

Cada Data Set contém dois vetores de controle (vide pagina 94):

- NP: contém o número do Data Set que é "dono" de cada equação;
- GNodeNumber: contém o número global da equação.

Estes dois vetores têm a mesma dimensão das demais matrizes do Data Set não só para facilitar a escrita dos algoritmos, como também para economizar espaço em memória, haja visto que, como cada Data Set deve ocupar um processador, cada processador deveria manter uma cópia desses vetores, inclusive da parte que não diz respeito ao Data Set em questão.

Com base nesses dois vetores o **MM** (Map Manager) monta dois vetores de índices para cada par de Data Sets possível. Esses índices são os índices locais, em cada um dos Data Sets, que se referem a uma mesma equação global. O primeiro desses vetores é chamado

de SendEqIdx e possui os índices das equações locais que devem ser enviadas para um outro determinado Data Set. O segundo chama-se ReceiveEqIdx e armazena os índices das equações locais que devem receber a contribuição proveniente de um determinado Data Set.

Cada SendEqIdx é montado e submetido ao **DM**, sendo que seu id é guardado em uma devida posição de uma matriz de mapeamento chamada fSendMap. Da mesma forma, cada ReceiveEqIdx é montado e submetido ao **DM**, sendo que seu id é guardado em uma devida posição de uma outra matriz de mapeamento chamada fReceiveMap. Tanto fSendMap como fReceiveMap são dados privados da classe TMapManager.

Veja o seguinte exemplo:

Dado um sistema A x = b, onde:

$$\begin{vmatrix} A_{00} & A_{01} & A_{02} & & & \\ A_{10} & A_{11} & A_{12} & & & \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & \\ & & A_{32} & A_{33} & A_{34} & \\ & & A_{42} & A_{43} & A_{44} & \end{vmatrix} \begin{vmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{vmatrix}$$

e temos dois Data Sets configurados como segue

Data Set 0
NP={0,0,0}
GNodeNumber ={0,1,2}

Data Set 1
NP={0,1,1}
GNodeNumber ={2,3,4}

Sendo assim ficaremos com os seguintes mapas:

fSendMap	S_{00}	0
	S_{10}	S_{11}

fReceiveMap	R_{00}	0
	R_{10}	R_{11}

Quando o Data Set 1 calcular seu resíduo, a função GetSendEqIdx do **MM** vai retorna o id do seguinte vetor de índices:

$SendEqIdx_{10}=\{0\}$

A equação 0 é então calculada e armazenada num vetor. Esse vetor é enviado ao Data Set 0, para que o mesmo atualize seu resíduo. Para atualizar o seu resíduo o Data Set 0 vai usar a função GetReceiveEqIdx do **MM** para pegar um vetor com os índices das equações que devem ser atualizadas. Receberá o seguinte vetor:

$ReceiveEqIdx_{10}=\{2\}$

O Data Set 0 sabe então que a equação 2 deve ser atualizada com os dados que chegaram do Data Set 1.

Nos seus cálculos, os Data Sets devem esperar que os Data Sets vizinhos atualizem seus resíduos. Para informar cada Data Set quantas atualizações ele deve esperar o Map Manager implementa um método UpdatesToWait() que funciona da seguinte forma:

1. UpdatesToWait(dset,0) : retorna o número de Data Sets para os quais dset envia dados. Este valor, chamado nwaits, é conseguido contando o número de valores não nulos da linha dset de qualquer um dos dois mapas.

2. UpdatesToWait(dset,1) : retorna o número de Data Sets que enviam dados para dset. Este valor, chamado nwaits1, é conseguido contando o número de valores não nulos da coluna dset de qualquer um dos dois mapas.

6.2.1 Criação dos Mapas

A classe TMapManager recebe um único parâmetro em seu construtor: o id do vetor que com os ids de cada Data Set. O construtor chama então, uma função privada chamada Init() que faz a criação e montagem dos dados. Com o id do vetor de ids dos Data Sets a função Init pede ao **DM** um ponteiro para esse vetor. São criados os dois mapas, fSendMap e fReceiveMap, que são matrizes quadradas de *long int* com dimensão igual ao número de Data Sets. É só então que a criação dos mapas propriamente dita começa.

Para criar os mapas é feito o procedimento descrito a seguir para cada Data Set. Pega-se um ponteiro para a estrutura TCGGlobalData do Data Set "A", para dessa forma obter o id do vetor com os números globais de cada equação e vetor com o números dos Data Sets donos de cada equação, estes vetores são respectivamente o fGNodeNumber e o fNP do Data Set "A". Com esses vetores e para cada equação é feito o seguinte: utilizando fNP referente a equação, sabe-se a qual Data Set a mesma pertence, chamemos este de Data Set "B". Criam-se dois vetores, um para guardar índices locais no Data Set "A" e outro para guardar índices locais no Data Set "B". Dentro do vetor com os números globais de cada equação do Data Set "B" procura-se aquelas

pertencentes a "A". Quando esse número de equação global é encontrado no fGNodeNumber do Data Set "B", tem-se o índice local em "B" da equação de "A" que está sendo trabalhada. O vetor com os índices locais das equações em "A" é submetido ao **DM** e o seu id é guardado na posição (A,B) do mapa fSendMap. Analogamente, o vetor com os índices locais das equações em "B" é submetido ao **DM** e o seu id é guardado na posição (A,B) do mapa fReceiveMap. Ambos os mapas são dados privados do Map Manager e só podem ser obtidos usando os métodos GetSendEqIdx() e GetReceiveEqIdx() respectivamente. Essa medida foi tomada para garantir total reaproveitamento do código se uma forma de mapeamento mais eficaz for implementada.

6.3 Procedimentos, Tarefas e Classes

Antes de começar as iterações são criados cada um dos vetores que serão usados por cada Data Set. Com os ids destes vetores, são criadas estruturas TCGGlobalData que servem como uma descrição de cada Data Set. O algoritmo de gradiente conjugado pré condicionado foi dividido em tarefas sendo que para cada tarefa foi criada uma classe. A seguir serão descritos os procedimentos iniciais, as classes das tarefas e as demais classes. Antes de ler detalhes sobre a implementação o leitor deve situar cada tarefa dentro da divisão do algoritmo proposta na Figura 19.

6.3.1 Procedimentos iniciais

Cada uma das matrizes e vetores (A, z, r, etc.) de cada Data Set, bem como cada dado global (ro, ptq, etc.) são criados pelo processador que vai iniciar todo o algoritmo. Chamaremos tal processador de "processador 0". Antes de criar a tarefa de iniciação do ambiente (TCGStartTask), o processador 0 cria cada uma das matrizes citadas acima, além dos coeficientes globais, e submete os mesmos ao **DM** que retorna um id para cada um desses objetos. Cria-se então um objeto da classe TCGGlobalData por Data Set para armazenar esses ids, que serão usados posteriormente pelas tarefas, quando as mesmas precisarem ter acesso aos dados do Data Set.

Cada objeto TCGGlobalData é então submetido ao **DM** e seu id é guardado num vetor. Esse vetor é submetido ao Data Manager e é o seu id que é passado para o objeto TCGStartTask, que, quando executada, cria e submete cada uma das tarefas de execução

contínua usando para isso o conteúdo desse vetor para recuperar cada estrutura TCGGlobalData.

6.3.2 Classe TCGGlobalData

A classe TCGGlobalData é usada unicamente para armazenar os dados de um Data Set que uma tarefa pode precisar. Os dados armazenados num objeto da classe TCGGlobalData são todos públicos e os métodos da classe se resumem ao seu construtor, junto aos métodos imprescindíveis a qualquer classe derivada de TSaveable.

Os dados armazenados num objeto TCGGlobalData para cada Data Set são:

- int fDataNumber : número do Data Set;
- int fProcNumber : número do processador que tratará desse Data Set;
- long fNPID : id do vetor que contém o número do Data Set dono de cada equação[8];
- long fGNodeNumberId : id do vetor que contém o número da equação para o problema global [8];
- long fDSetProcId : id do vetor que contém o número do processador que tratará de cada Data Set [8];
- long fAID : id da matriz com a parte da matriz de coeficientes A (problema global) referente a esse Data Set[9];
- long fDID : id do vetor com a diagonal da matriz A do problema global [8];
- long fxID : id do vetor com a parte do vetor de solução x referente a esse Data Set [8];
- long fpID : id do vetor p desse Data Set [8];

[8] este vetor tem tamanho igual ao número de equações do Data Set.

[9] Esta matriz é quadrada e tem dimensão igual ao número de equações do Data Set.

- long frID : id do vetor com a parte do vetor de resíduo r referente a esse Data Set [8];
- long fzID : id do vetor z desse Data Set [8];
- long fqID : id do vetor q desse Data Set [8];
- long fGRoId : id do coeficiente global Ro;
- long fGPRoId : id do coeficiente global PRo (previous ro = $ro_{(i-1)}$);
- long fGptqId : id do coeficiente global $p^t q$;
- long fGrtrId : id do coeficiente global $r^t r$ (para verificação da convergência);
- long fGStatusId : id da flag global que indica se o algoritmo convergiu ou não.

6.3.3 Classes TSvDouble e TSvInt

Para que os coeficientes pudessem ser submetidos ao **DM**, e todos os processadores pudessem ter acesso a eles, foi criada uma classe TSvDouble que é derivada de TSaveable e tem apenas um double como dado. O mesmo foi feito, analogamente, para que a flag Status, um inteiro, pudesse ser visível de qualquer processador do ambiente paralelo. Criou-se uma classe TSvInt.

Métodos e dados públicos:

- double/int buffer : único dado da classe;
- double/int GetValue(void) : retorna o valor armazenado em buffer;
- void SetValue(double/int nval) : muda o valor armazenado em buffer para nval;

6.3.4 Classe TCGStartTask

Como já foi citado, é um objeto da classe TCGStartTask o responsável pela criação e submissão das tarefas de execução contínua (que permanecem na fila de tarefas

do TM até que a solução seja alcançada). Seu construtor recebe apenas dois parâmetros: o processador onde deverão ser criadas as tarefas, isto é, o processador onde a TCGStartTask será executada. Vale lembrar que cada uma das tarefas criadas por TCGStartTask serão transferidas e executadas nos processadores especificados nos respectivos Data Sets, mais exatamente na estrutura TCGGlobalData (vide pagina 94) referente ao Data Set da tarefa.

Dados:

- long fDataIdVecId: id do vetor onde estão armazenados os ids de cada estrutura TCGGlobalData;

Execução:

Executada uma única vez por análise, a tarefa TCGStartTask começa seu método Execute() recuperando um ponteiro para o vetor com os ids dos TCGGlobalData. Esse ponteiro foi batizado com o nome de DataIdVec. O número de Data Sets é determinado verificando-se o tamanho de DataIdVec através do seu método capacity()[3].

Para cada Data Set cria-se:

- Um ponteiro para a sua estrutura TCGGlobalData usando-se o id armazenado no vetor DataIdVec;

- Um ponteiro para um vetor chamado DSetProc (vide pagina 95) contendo o número do processador de cada Data Set. Esse ponteiro é conseguido com o id desse vetor usando o método GetObjPtr() do **DM**. Tal vetor é único para todos Data Sets e seu id é um dos dados da estrutura TCGGlobalData.

Com esses dois ponteiros, para TCGGlobalData e para o vetor DSetProc, cada tarefa de execução contínua já pode ser criada e submetida, e é esse o próximo passo.

Um objeto da classe TCGTask1 é criado e usando o método AddDataDepend que tal classe herda de TTask, toda a dependência dessa tarefa aos dados que a mesma utiliza é acertada. Só então o objeto da classe TCGTask1, que define a primeira tarefa do

algoritmo CG e será descrita a seguir, é submetido ao **TM**. Um procedimento análogo é executado para submeter, em seguida, as tarefas 3, 5, 8, e 10.

Após a criação e submissão de cada tarefa, o método TransferObject é usado para pedir a transferência de cada um dos dados importantes para cada tarefa, aqueles aos quais cada tarefa registrou sua dependência com o método AddDataDepend.

ESuccess é retornado indicando que esta tarefa foi executada com sucesso e deve ser removida da fila de tarefas do TM.

6.3.5 Classe TCGTask1

Esta tarefa é responsável pelo pré-condicionamento do vetor de resíduo r. Nesse caso foi utilizado o pré condicionador Jacobi, ou seja, divide-se cada posição do vetor de resíduo pela respectiva componente da diagonal da matriz global de coeficientes A. O pré condicionador Jacobi sabidamente não é o mais eficiente pré condicionador, todavia, ele é adequado para uma implementação paralela, o que justifica a utilização de tal pré condicionador.

Cada Data Set traz consigo uma parte da diagonal da matriz de coeficientes, e é o inverso dessa parte que é multiplicado pela parte do vetor de resíduo tratada por este Data Set. Logo, o pré-condicionamento, que consiste na multiplicação do resíduo pelo inverso da diagonal da matriz de coeficientes, é feito em paralelo.

O construtor da TCGTask1 recebe como parâmetro apenas o número do processador onde a tarefa deverá ser executada e o id da estrutura TCGGlobalData referente ao Data Set que será tratado.

Como não só a estrutura TCGGlobalData do Data Set, mas também todas as demais matrizes e vetores referentes a esse Data Set estão sob o controle do Data Manager, cada dado destes é transferido automaticamente para cada processador que precisa usá-los. Dessa forma a única informação própria da classe TCGtask1 que é levada de um processador para outro pelo objeto (com os métodos herdados de TSaveable), é o id da estrutura TCGGlobalData referente ao Data Set. Então pede-se acesso a cada dado do Data Set procedendo dois pequenos passos: com o id da estrutura TCGGlobalData a mesma é recuperada, e a partir dos ids que esta contém o método

GetObjPtr() do DM é chamado e consegue-se um ponteiro para cada dado referido naquela estrutura[10].

Dados:

- long fVersion : inteiro indicando qual é o ciclo corrente do algoritmo de gradiente conjugado. Esse dado é usado para calcular-se a versão que necessita-se de cada dado;

- long fDataId : id da estrutura TCGGlobalData com os dados do Data Set que será trabalhado;

- TCGGlobalData *fGlobalData : ponteiro para a estrutura TCGGlobalData com os dados do Data Set que será trabalhado;

- LongVec *fNP : ponteiro para o vetor contendo o número do Data Set "dono" de cada equação tratada por esse Data Set;

- TFMatrix *fD : ponteiro para o vetor com a parte da diagonal da matriz de coeficientes principal referente as equações tratadas pelo Data Set em questão, isto é, pelo Data Set referente a esse objeto da classe TCGTask1;

- TFMatrix *fz : ponteiro para o vetor de resíduo pré condicionado, vetor z, referente ao Data Set em questão;

- TFMatrix *fr : ponteiro para o vetor de resíduo, vetor r, referente ao Data Set em questão;

Os ponteiros fz, fr e fD são dados da classe para que os mesmos sejam criados apenas na primeira execução, poupando o processamento gasto com tal procedimento nas demais execuções. Isto é possível pois, de acordo com o implementado, o **DM** não mudará esses dados de processador ou de posição na memória, o que garante a validade dos ponteiros em todas as execuções.

---

[10]Cada ponteiro é iniciado na primeira execução do método Execute() da classe TCGTask1.

Execução:

Todos os ponteiros para dados do Data Set, usados pela tarefa 1, são "inicializados" caso ainda não tenham sido. Se o Data Manager não conseguir recuperar um desses dados um erro é gerado e a execução é abortada (exit(-1)). Com os ponteiros para todos os dados o pré-condicionamento é feito da seguinte forma:

- Num laço varrendo todos os itens dos vetores fz, fr, fD, e fNP, atribui-se a z o valor de r dividido pelo valor D, se e somente se, o item referente do vetor NP (com o número do Data Set dono de cada equação) for igual ao número do Data Set referente a tarefa em execução. Em outras palavras o pré-condicionamento é feito somente para as equações próprias do Data Set. Vide código abaixo:

```
int Dr = fD ->Rows();
for(int h = 0; h < Dr; h++)
{
      if((*fNP)[h] == (fGlobalData ->fDataNumber))
            (*fz)(h,0) = (*fr)(h,0)/((*fD)(h,0));
}
```

- Num laço varrendo todos os itens dos vetores fz, fr, e fNP, acumula-se o produto de cada item de z por cada item de r, somente para as equações cujo dono seja o Data Set referente a essa tarefa 1. Isto é feito para calcular a parte de produto interno $r^t z$ inerente ao Data Set desse objeto tipo TCGTask1. Vide código abaixo:

```
double rtz = 0.0;
for(i = 0; i < Dr; i++)
      if((*fNP)[i] == (fGlobalData ->fDataNumber))
            rtz += ((*fr)(i,0))*((*fz)(i,0));
```

- Cria-se então um objeto da classe TDblUpdateTask para acumular essa parte de rtz (Ro) referente a esse Data Set, no Ro global que será usado pelas tarefas subsequentes (tarefa 2 da lista de tarefas).

- Atualiza-se as dependências aos dados e também a versão de z e incrementa-se o valor de fVersion. Maiores detalhes serão dados na parte desse capítulo que se refere a dependência de dados.

Depois de executados os passos citados acima, o valor da flag global Status é verificado, caso o valor seja ECanStop o método Execute termina retornando ESuccess, caso contrário EContinue é retornado indicando que esta tarefa foi executada mas deve continuar na fila de tarefas do TM, para ser executada novamente assim que os dados necessários para execução estejam disponíveis com as versões necessárias.

6.3.6 Classe TCGTask3

A tarefa 3 (classe TCGTask3) é responsável pelo cálculo do vetor p, resultado da soma de z com o p do ciclo anterior multiplicado pelo coeficiente beta, exatamente como o descrito na relação de tarefas. No primeiro ciclo o coeficiente beta é zero e nos demais é a razão entre o ro atual e o ro do ciclo anterior. Dessa forma garante-se que o p do primeiro ciclo é igual ao z do primeiro ciclo. É bom lembrar que estamos falando dos vetores z e p do Data Set referente a essa tarefa.

No caso da tarefa 3, existe a necessidade de uma troca de dados entre os diversos Data Sets, isto pois cada equação não necessariamente está num único Data Set. Por esse motivo, a classe TCGTask3 precisa de um dado a mais que é o id do Map Manager. Como já foi citado no item que descreve o funcionamento do Map Manager (vide item Classe TMapManager, pagina 95), é através do Map Manager que as tarefas conseguem saber quais os Data Sets que contribuem para essa equação, ou seja, quais os Data Sets que tratam de uma mesma equação global e qual é o índice local dessa equação dentro do Data Set.

Além do id do Map Manager o construtor da classe TCGTask3 recebe mais 3 argumentos: o número do processador onde a tarefa será executada, o id do Data Set referente a essa tarefa 3 e o id do vetor DataIdVec, que contém os ids de todos os Data Sets.

Dados:

- long fVersion : inteiro indicando qual é o ciclo corrente do algoritmo de gradiente conjugado;
- long fDataId : id da estrutura TCGGlobalData com os dados do Data Set que será trabalhado;

- long fDataIdVecId: id do vetor DataIdVec que contém os ids de cada estrutura TCGGlobalData;

- long fMapId : id do Map Manager;

- TCGGlobalData *fGlobalData : ponteiro para a estrutura TCGGlobalData com os dados do Data Set que será trabalhado;

- LongVec *fNP : ponteiro para o vetor contendo o número do Data Set "dono" de cada equação tratada por esse Data Set;

- TFMatrix *fp : ponteiro para o vetor p, iniciado a partir do id armazenado na estrutura TCGGlobalData do Data Set a ser trabalhado;

- TFMatrix *fz : ponteiro para o vetor de resíduo pré condicionado, vetor z, referente ao Data Set em questão;

- unsigned short *fEqOk : ponteiro para um vetor de n flags, onde n é o número de equações locais do Data Set. Essas flags indicam se a i-ésima equação foi calculada (fEqOk[i]=1) ou não (fEqOk[i]=0). A necessidade do vetor fEqOk, bem como sua funcionalidade será discutida no decorrer da descrição da execução da tarefa 3.

Os ponteiros acima relacionados só são dados da classe para poupar esforço computacional. Cada ponteiro é iniciado apenas na primeira execução da tarefa, o que só é possível por serem dados da classe. Essa medida pôde ser tomada pois da forma como o algoritmo foi implementado o **DM** não muda estes dados de posição de memória, o que invalidaria estes ponteiros.

Execução:

Passo a passo, os primeiros procedimentos envolvidos na execução da tarefa 3 (e na maioria das tarefas de execução contínua do algoritmo CG) são os seguintes:

- Se o ponteiro para a estrutura TCGGlobalData ainda é invalido (contém NULL) este é iniciado passando-se o valor DataId (id da estrutura TCGGlobalData do Data Set) para o método GetObjPtr() do **DM**;

- O mesmo é feito em seguida para cada um dos demais ponteiros. Se a recuperação de qualquer ponteiro falhar, o programa é abortado de imediato.

- O vetor de flags fEqOk (unsigned short[]) é criado dinamicamente em memória na primeira execução da tarefa 3. Esse vetor não precisa estar sob o controle do DM pois é privado de cada tarefa 3 e é "zerado" no inicio de cada execução, o que dispensa a necessidade de transportá-lo de um processador para outro.

- O coeficiente beta é calculado. Se é a primeira execução dessa tarefa 3, beta tem valor zero, do contrário beta é igual a ro dividido pelo valor de ro(i-1). Dado que ro e ro(i-1) (objetos globais Ro e PRo, vide pagina 94).

- Atualiza-se fVersion e as dependências dessa tarefa 3 para com os dados de seu Data Set. Essa etapa será descrita detalhadamente no item que tratará de descrever a dependência das tarefas aos dados (vide item Dependência de tarefas sobre versões de dados, pagina 69).

- Determina-se o número de Data Sets em que o problema global foi dividido. O número de Data Sets é igual ao tamanho do vetor DSetProc (vide pagina 95);

Sabendo-se que a comunicação entre os processadores é responsável pela maior parte do tempo gasto em processos paralelos, optou-se por calcular primeiro aqueles itens do vetor p que devem ser enviados aos demais processadores, para só então calcular os demais itens do vetor. Em termos de elementos finitos, são calculados primeiro os nós de interface para depois calcular os nós internos da malha local do processador (ou Data Set). Isto é feito da seguinte forma, para cada Data Set (excetuando-se o referente a esta tarefa) faz-se o seguinte:

- Com o método GetReceiveEqIdx() do Map Manager (vide item Classe TMapManager, pagina 95) consegue-se um ponteiro para um vetor com os índices locais (referentes ao Data Set da tarefa 3) das equações que recebem contribuição do i-ésimo Data Set, chamaremos esse vetor de idx. Se não existir tal vetor (idx igual a NULL) conclui-se que o i-ésimo Data Set não contribui para nenhuma equação do Data Set referente a esta tarefa 3 e, sendo assim, o Data Set i é ignorado;

- Cria-se um vetor "tmpp" de comprimento igual ao de idx para armazenar os valores a serem enviados ao Data Set i;

- Usando-se a i-ésima ocorrência de DataIdVec, recupera-se um ponteiro para a estrutura TCGGlobalData do i-ésimo Data Set que é para onde deve-se enviar tmpp;

- O vetor tmpp é preenchido da seguinte forma:

Sendo j um índice que varia de zero ao tamanho de tmpp e u a j-ésima ocorrência do vetor de índices locais idx, se fEqOk[u] é zero o novo valor de p[u] é calculado e armazenado também na posição j de tmpp:

```
if(!fEqOk[u])
{
        tmpp(j,0) = (*fz)(u,0)+(beta * ((*fp)(u,0)));
        (*fp)(u,0) = tmpp(j,0);
        fEqOk[u] = 1;
}
else
        tmpp(j,0) = (*fp)(u,0);
```

Se fEqOk[u] não é zero significa que p[u] já foi calculado, logo tmpp[j] é simplesmente igual a p[u]. O vetor fEqOk, como pode-se observar, evita que, caso p[u] deva ser enviado a mais de um Data Set, este seja calculado novamente, o que, além de incorrer num processamento redundante desnecessário, acarretaria um erro, pois o código acima calcula a equação:

$$p_i = z_i + \text{beta}\ p_{i-1}$$

e como o p do lado direito do igual deve ser o p do ciclo anterior, executar duas vezes esse cálculo num mesmo ciclo (numa mesma execução da tarefa) acarreta um erro. Vejamos um exemplo numérico:

Supondo que p_i fosse calculado novamente antes de ser enviado para cada Data Set, teríamos o seguinte (onde i é o número do ciclo):

$z_i[u] = 4$

beta = 1

$p_{i-1}[u] = 5$

para o Data Set 3:

$p_i[u] = 4 + 1 * 5$

para o Data Set 4:

$p_i[u] = 4 + 1 * 9$ ERRADO! $p_i[u]$(Data Set 3) ≠ $p_i[u]$(Data Set 4)

Uma vez estando o vetor tmpp preenchido com as contribuições de um Data Set para com o outro, uma tarefa TVecUpdateTask (vide pagina 117) é criada para atualizar os valores de p no outro Data Set. Essa tarefa, é claro, será executada no processador relativo ao Data Set que terá seu vetor p atualizado.

Depois que o processo acima descrito foi executado para todos os Data Sets que recebem contribuições do Data Set relativo a essa tarefa 3, os demais valores de p são calculados, ou seja, se fEqOk[u] é zero p[u] é então calculado. Para uma malha de elementos finitos, estas posições de p seriam referentes aos nós internos na malha parcial, pois os elementos de fronteira já foram calculados (fEqOk[u] = 1).

Para finalizar após terem sido calculados e enviados os valores de p para os demais Data Sets e também terem sido calculados os valores de p particulares desse Data Set, verifica-se o valor da flag GStatus. Se esse valor for igual a ECanStop a tarefa atual retorna ESuccess e é retirada do Task Manager, do contrário EContinue é retornado e a tarefa 3 será executada novamente assim que suas dependências forem atendidas.

6.3.7 Classe TCGTask5

A classe TCGTask5 define os objetos para execução da tarefa de número 5. Essa tarefa tem por objetivo o cálculo do vetor q, resultado do produto matricial da matriz A (parte da matriz de coeficientes relativa ao Data Set em questão) pelo vetor p (calculado pela tarefa 3).

Assim como a tarefa 3, a tarefa 5 também envia dados a outros Data Sets, isto deve-se ao fato de que o vetor q de cada Data Set deve conter os valores do produto da matriz global A de coeficientes pelo vetor global p, o que consegue-se somando os valores dos produtos de A por p de cada Data Set, ou seja:

$$q = \sum_{i=1}^{nDSets} A^i p^i$$

A tarefa 5 segue as características da tarefa 3, ou seja, assim como a tarefa 3, a tarefa 5 também armazena o id do Map Manager e usa o mesmo para "mapear" suas equações locais para as equações locais do Data Set a receber a contribuição.

O construtor da classe TCGTask5 recebe 4 argumentos: o número do processador onde a tarefa será executada, o id do Data Set referente a essa tarefa 3, o id do Map Manager e o id do vetor DataIdVec, que contém os ids de todos os Data Sets.

Dados:

- long fVersion : inteiro indicando qual é o ciclo corrente do algoritmo de gradiente conjugado;
- long fDataId : id da estrutura TCGGlobalData com os dados do Data Set que será trabalhado;
- long fDataIdVecId: id do vetor DataIdVec que contém os ids de cada estrutura TCGGlobalData;
- long fMapId : id do Map Manager;
- TCGGlobalData *fGlobalData : ponteiro para a estrutura TCGGlobalData com os dados do Data Set que será trabalhado;
- LongVec *fNP : ponteiro para o vetor contendo o número do Data Set "dono" de cada equação tratada por esse Data Set;

- TFMatrix *fp : ponteiro para o vetor p, iniciado a partir do id armazenado na estrutura TCGGlobalData do Data Set a ser trabalhado;

- TFMatrix *fq : ponteiro para o vetor q a ser calculado com o produto da matriz A pelo vetor p referente ao Data Set em questão;

- TFMatrix *fA : ponteiro para a matriz A com os coeficientes referente ao Data Set em questão;

Execução:

No início da execução, a tarefa 5 também faz uma verificação para saber se os ponteiro para a estrutura com os dados do Data Set e para os dados do Data Set foram iniciados, se ainda não foram iniciados isto é feito (vide item Classe TCGTask3, pagina 105).

Ainda seguindo o mesmo padrão que TCGTask3, o próximo passo é o acerto do dado fVersion da tarefa 5 sendo executada. Só depois disso é que a execução da tarefa em si começa realmente, pois o que foi descrito até aqui são procedimentos de controle (iniciação de ponteiros e acerto de versão e dependência).

Calcula-se fq como o produto de fA por fp.

O passo seguinte é o envio de q para cada Data Set (excetuando-se o referente a esta tarefa):

- Usando o método GetSendEqIdx() do Map Manager consegue-se um ponteiro para um vetor com os índices locais (referentes ao Data Set da tarefa 5) das equações que enviam contribuição para o Data Set i (caso tal Data Set receba contribuição do Data Set referente a essa tarefa 5, do contrário passa-se ao próximo Data Set);

- Cria-se um vetor "tmpq" de comprimento igual ao de idx para armazenar os valores a serem enviados ao Data Set i;

- Com o id armazenado na posição i do vetor DataIdVec recupera-se o a estrutura TCGGlobalData do Data Set i que receberá o vetor de contribuição tmpq;

- O vetor tmpq é preenchido com as posições de q a serem enviadas para o Data Set i;

- Com o vetor tmpq preenchido, cria-se uma tarefa TVecUpdateTask para atualizar os valores de q no Data Set de destino.

- Cria-se então um objeto da classe TCGTask7 (vide pagina 112), responsável pelo cálculo do coeficiente ptq (p transposto multiplicado por q), que é submetido ao Task Manager. Esse coeficiente é calculado em tarefa a parte, pois tal cálculo só pode ser feito depois que a contribuição dos demais Data Sets para o vetor q (tarefa 6) tiver sido computada;

- Por fim, verifica-se o valor da flag Status, se este é igual a ECanStop a execução termina retornando ESuccess e posteriormente essa tarefa 5 é retirada do Task Manager, do contrário EContinue é retornado e essa tarefa 5 será executada novamente assim que suas dependências forem atendidas.

6.3.8 Classe TCGTask7

Esta classe define a tarefa 7 que consiste no calculo do produto interno de p por q que é então acumulado em um coeficiente global chamado ptq. Foi criada uma tarefa exclusiva para esse calculo devido a necessidade do mesmo ser executado somente depois que todas as contribuições dos demais Data Set, tanto para p como para q, tenham sido computadas.

Observa-se que essa tarefa é relativamente simples, quando comparada com as tarefas 1, 3 e 5, haja visto que a tarefa 7 não envia contribuição na forma de vetor para nenhum Data Set, sendo sua contribuição acumulada numa variável global, ou seja, visível a todos processadores. Uma tarefa simples também pelo fato de ser criada e submetida ao Task Manager a cada ciclo do algoritmo (retorna sempre ESuccess);

Seu construtor recebe apenas dois parâmetros: o número do processador onde será executada e o id da estrutura TCGGlobalData do seu Data Set.

Dados:

- long fDataId : id da estrutura TCGGlobalData com os dados do Data Set que ser á trabalhado;

- TCGGlobalData *fGlobalData : ponteiro para a estrutura TCGGlobalData com os da dos do Data Set que será trabalhado;

- TFMatrix *fp : ponteiro para o vetor p, iniciado a partir do id armazenado na estrutura TCGGlobalData do Data Set a ser trabalhado;

- TFMatrix *fq : ponteiro para o vetor q a ser calculado com o produto da matriz A pelo vetor p referente ao Data Set em questão;

- LongVec *fNP : ponteiro para o vetor contendo o número do Data Set "dono" de cada equação tratada por esse Data Set.

Execução:

Seguindo o mesmo padrão das demais tarefas, os ponteiros são iniciados e verificados antes do cálculo do produto de p transposto por q. Feito isso, os valores de p e de q referentes ao Data Set dessa tarefa 7 são multiplicados e acumulados em uma variável temporária. Sabe-se que a equação pertence ao Data Set em questão tomando como base o vetor fNP. Veja o código abaixo:

```
double ptq = 0.0;
for(int i = 0; i < fNP ->capacity(); i++)
 if((*fNP)[i] == fGlobalData ->fDataNumber)
  ptq += ((*fp)(i,0))*((*fq)(i,0));
```

Essa variável com o resultado do produto interno das partes de p e q referentes ao Data Set da tarefa 7 é então passada para um objeto da classe TDblUpdateTask (vide pagina 119) que será responsável pelo acúmulo desse valor no coeficiente global Gptq. Essa tarefa TDblUpdateTask será executada no processador principal, "dono" da variável Gptq, que por convenção é o processador zero.

Finalizando, ESuccess é retornado indicando que essa tarefa deve ser removida da fila do Task Manager.

6.3.9 Classe TCGTask8 (tarefas 8 e 10)

A classe TCGTask8 define dois tipos de tarefas de execução contínua, uma para o cálculo do resíduo r e outra para o cálculo do vetor da solução aproximada x. Uma

instância da classe TCGTask8 trabalhará no cálculo de x se o parâmetro type do seu construtor for igual a um, do contrário calculará o resíduo r. Além do tipo da tarefa, o construtor recebe como parâmetro o número do processador onde será executada a tarefa, o id da estrutura TCGGlobalData do Data Set da tarefa, o id do vetor DataIdVec e o id do Map Manager.

Optou-se por usar uma única classe para duas tarefas por se tratarem de cálculos extremamente semelhantes (o cálculo de r e de x), exceto pelo fato da tarefa responsável pelo cálculo de r calcular também a contribuição do Data Set para o produto interno do vetor de resíduo r, que é armazenado na variável global Grtr que é usada na verificação da convergência. Quando tratar-se de um objeto TCGTaks8 para o cálculo de r, ao final da execução da tarefa, o produto interno $r^t r$ é calculado e uma tarefa TDblUpdateTask é criada para acumular na variável global Grtr. Isso não acontece para as tarefas de cálculo de x, pois para montar o vetor x basta organizar os vetores x de todos os Data Sets.

Dados:

- int fType : inteiro que indica o tipo da tarefa, se diferente de zero é um objeto para calculo de r do contrário para o calculo de x;
- long fVersion : inteiro indicando qual é o ciclo corrente do algoritmo de gradiente conjugado;
- long fDataId : id da estrutura TCGGlobalData com os dados do Data Set que será trabalhado;
- long fDataIdVecId : id do vetor DataIdVec que contém os ids de cada estrutura TCGGlobalData;
- long fMapId : id do Map Manager;
- TCGGlobalData *fGlobalData : ponteiro para a estrutura TCGGlobalData com os dados do Data Set que será trabalhado;

- LongVec *fNP : ponteiro para o vetor contendo o número do Data Set "dono" de cada equação tratada por esse Data Set;

- TFMatrix *fV : ponteiro para o vetor de que receberá o resultado da equação: ou o resíduo, vetor r, ou a solução, vetor x, ambos referentes ao Data Set em questão;

- TFMatrix *fV2 : ponteiro para o vetor usado no cálculo: ou o vetor p, ou o vetor q, ambos referentes ao Data Set em questão;

Execução:

A iniciação e verificação dos ponteiros só difere das demais tarefas pois depende do tipo da tarefa, se for para o cálculo de r os ponteiros são iniciados com p e r, se for para o cálculo de x os ponteiros são iniciados com q e x. Após a iniciação dos ponteiros o coeficientes C usado na formula é calculado como a razão entre ro e ptq, se a execução for para o cálculo de x, C é multiplicado por -1, o que possibilita o uso do seguinte código independente de se tratar do cálculo de r ou x:

```
for(int u = 0; u < fNP ->capacity(); u++) {
 if((*fNP)[u] == DataSetNumber)
  (*fV)(u,0) = (*fV1)(u,0)-(C * ((*fV2)(u,0)));
}
```

É bom ressaltar que, como em algumas das demais tarefas, o cálculo é feito somente para as equações pertencentes ao Data Set em questão, usando para tal o vetor fNP.

Caso trate-se do cálculo de x, a execução termina por aqui e o valor de retorno depende do conteúdo da flag GStatus. Caso contrário a execução prossegue calculando o produto interno do vetor r do Data Set referente a tarefa. Uma vez calculado esse valor (rtr), cria-se e submete-se uma tarefa tipo TDblUpdateTask para acumular esse valor na variável global Grtr e, só depois, o valor de GStatus é verificado e a execução termina retornando ESuccess ou EContinue.

6.3.10 Classe TChkTask (tarefa 11)

A maior responsabilidade da tarefa definida pela classe TChkTask é verificar a convergência do algoritmo, mas além disso, essa tarefa também zera os coeficientes

globais tais como rtr, Ro, ptq, além de mudar o valor de PRo (ro do ciclo anterior) para o valor atual de ro pois essa é a última tarefa a ser executada num ciclo do algoritmo.

Dados:

- long fGStatusId : id da flag global que indica se o algoritmo convergiu ou não.
- long fGrtrId : id do coeficiente global rtr (para verificação da convergência);
- long fGptqId : id do coeficiente global ptq;
- long fGPRoId : id do coeficiente global PRo (previous ro = $ro_{(i-1)}$);
- long fGRoId : id do coeficiente global Ro;
- long fVersion : inteiro indicando qual é o ciclo corrente do algoritmo de gradiente conjugado. Esse dado é usado para calcular-se a versão que necessita-se de cada dado;
- int fNumDataSets : Numero total de Data Sets;
- int fNumEq : Numero de equações do problema (dimensão do problema global).

Execução:

São criados ponteiros para cada um dos coeficientes (ptq, ro, etc.) e para a flag GStatus a partir dos respectivos ids. Caso estes ponteiros sejam criados com sucesso, verifica-se então o valor de GStatus, se este for ECanStop a execução é finalizada retornando-se ESuccess, caso contrário a execução prossegue com os seguintes passos:

- Com o valor acumulado em Grtr (produto interno do vetor de resíduo) calcula-se o resíduo (norma do vetor de resíduo), veja:

```
double gip = Grtr ->GetValue();
double res = sqrt(gip/fNumEq);
```

- PRo recebe o valor do ro calculado no ciclo e os coeficientes ro, ptq e rtr são zerados;

- Caso o valor do resíduo calculado seja menor que uma tolerância predeterminada, é atribuído o valor ECanStop para a flag GStatus, indicando que as demais tarefas podem ser finalizadas;

- Se o resíduo ainda não for tolerável a execução termina retornando EContinue, indicando que pelo menos mais um ciclo do algoritmo é necessário.

6.3.11 Classe TVecUpdateTask (tarefas 4 e 6)

Define uma tarefa para atualizar valores em um dado vetor num outro Data Set. Tal atualização pode ser de duas formas: atribuir valores a posições pré determinadas do vetor destino ou somar valores nessas posições. Essa característica se fez necessária dado ao fato de que o vetor gerado pela tarefa 5 que calcula o vetor q deve ser somado, o que difere da tarefa 3 que gera um vetor que deve ser atribuído em determinadas posições do vetor do Data Set que receberá a contribuição.

Seu construtor recebe os seguintes parâmetros:

- número do processador onde deve ser executada;

- número do Data Set que está enviando os dados, ou seja, o Data Set de origem;

- número do Data Set que receberá o vetor (Data Set de destino);

- id do vetor que será atualizado (vetor de destino);

- id do vetor NP;

- id do Map Manager;

- vetor com os valores a serem somados ou atribuídos;

- flag que indica o tipo da atualização (igual a zero = soma, diferente de zero = atribui).

Dados:

- long fDestObjId : id do vetor de destino;

- int fDestSetNumber: número do Data Set de destino;

- int fOriSetNumber: número do Data Set de origem;
- long fNPId : id do vetor NP, vetor que contém o número do Data Set dono de cada equação;
- long fMapId : id do Map Manager;
- TFMatrix fR : vetor com os valores a serem somados ou atribuídos;
- int fIsU : flag que indica se os valores deverão ser somados ou atribuídos.

Execução:

- São criados ponteiros para os dados necessários;
- A versão do vetor de destino é incrementada por intermédio do método IncrementVersion() do **DM**;
- O Map Manager é usado para conseguir-se os índices das posições do vetor de destino que serão somadas (ou preenchidas) com os valores do vetor fR. A atribuição ou soma é feita da seguinte forma: cada posição do vetor fR é somada ou atribuída a posição indicada no vetor de índices gerado pelo Map Manager, com o abaixo:

```
TFMatrix *U = (TFMatrix *)DM->GetObjPtr(fDestObjId);
TIntVec *idx;
if(fIsU)
      idx = MM->GetSendEqIdx(fDestSetNumber, fOriSetNumber);
else
      idx = MM->GetReceiveEqIdx(fOriSetNumber, fDestSetNumber);

if(fIsU) {
      for(int j = 0; j < idx ->Capacity(); j++)
      {
            (*U)((*idx)[j],0) = fR(j,0);
      }
} else {
      for(int j = 0; j < idx ->Capacity(); j++)
      {
            (*U)((*idx)[j],0) += fR(j,0);
      }
}
```

O vetor de índices idx é conseguido usando-se o método GetSendEqIdx() do Map Manager, se trata-se de uma atribuição. Isto pois, no caso de uma atribuição, assume-se que serão alteradas as posições do vetor destino pertencentes ao Data Set de origem, ou seja, aquelas posições cujas contribuições o Data Set destino da tarefa TVecUpdateTask envia ao Data Set de origem. Para o caso de tratar-se de uma soma, uma analise semelhante pode ser feita: quando será feita uma soma, assume-se que serão alteradas as posições do vetor destino pertencentes ao Data Set de destino, ou seja, aquelas posições cujas contribuições o Data Set destino da tarefa TVecUpdateTask recebe do Data Set de origem.

- A tarefa TVecUpdateTask é finalizada retornando ESuccess.

6.3.12 Classe TDblUpdateTask (tarefas 2, 7a e 9)

Provavelmente a classe mais simples do algoritmo, essa classe define uma tarefa para acumular um valor em um coeficiente global tal como Grtr, Gptq, etc. Diferente da tarefa para atualização de vetores, essa tarefa apenas soma o valor passado à ela ao coeficiente determinado.

Seu construtor recebe 3 parâmetros: o número do processador mestre ("dono" dos coeficientes) que é onde a tarefa será executada, o id do coeficiente a ser atualizado e o valor a ser somado.

Dados:

- long fGInProdId : id do coeficiente a ser atualizado;
- double fInProd : valor a ser somado.

Execução:

Usando o DM, consegue-se um ponteiro para o coeficiente a ser atualizado, um objeto da classe TSvDouble. O valor atual do coeficiente é conseguido usando-se o método GetValue() da classe TSvDouble. Esse valor é somado a fInProd e o resultado dessa soma passa a ser o novo valor do coeficiente, só então a versão do coeficiente é atualizada e a tarefa termina retornando ESuccess. Veja:

```
double v = GInProd ->GetValue();
v += fInProd;
GInProd ->SetValue(v);

DM->IncrementVersion(fGInProdId);

return ESuccess;
```

6.4 Dependência de dados

No caso do algoritmo de gradiente conjugado implementado, a primeira tarefa (TCGStartTask) é responsável pela criação de todas as demais tarefas de execução continua, sendo responsável também pelo registro das dependências de cada tarefa criada.

As tarefas do algoritmo de gradiente conjugado tem suas dependências de dados, para o primeiro ciclo, relacionadas no quadro abaixo:

Tabela 3 - Dependência das tarefas do algoritmo de gradiente conjugado

Tarefa	Dado	Acesso	Versão
TCGStartTask	DataIdVec	EReadAccess	
	DSetProc	EReadAccess	
	NP[1..n]	EReadAccess	
	DSetProc[1..n]	EReadAccess	
	GNodeNumber[1..n]	EReadAccess	
TCGTask1	DataIdVec	EReadAccess	
	NP	EReadAccess	
	DSetProc	EReadAccess	
	D	EReadAccess	
	r	EReadAccess	0
	z	EWriteAccess	
	Status	EReadAccess	0
TCGTask3	DataIdVec	EReadAccess	
	DataIdVec[1..n]	EReadAccess	
	NP	EReadAccess	
	DSetProc	EReadAccess	
	p	EWriteAccess	
	z	EReadAccess	1
	Ro	EReadAccess	NumDataSets[11]
	PRo	EReadAccess	0
	Status	EReadAccess	0
	mapas do Map Manager	EReadAccess	

[11] NumDataSets é o número total de Data Sets em que o problema foi dividido.

TCGTask5	DataIdVec	EReadAccess	
	DataIdVec[1..n]	EReadAccess	
	NP	EReadAccess	
	DSetProc	EReadAccess	
	p	EReadAccess	nwaits[12]
	A	EReadAccess	
	q	EWriteAccess	
	Status	EReadAccess	0
	mapas do Map Manager	EReadAccess	
TCGTask8 (type 0)	DataIdVec	EReadAccess	
	DataIdVec[1..n]	EReadAccess	
	NP	EReadAccess	
	DSetProc	EReadAccess	
	r	EWriteAccess	
	q	EReadAccess	nwaits1[13]
	Ro	EReadAccess	NumDataSets
	ptq	EReadAccess	NumDataSets
	Status	EReadAccess	0
TCGTask8 (type 1)	NP	EReadAccess	
	x	EWriteAccess	
	p	EReadAccess	nwaits
	Ro	EReadAccess	NumDataSets
	ptq	EReadAccess	NumDataSets
	Status	EReadAccess	0

[12] nwaits é o número de Data Sets para onde o Data Set corrente envia dados (informação obtida com o Map Manager, vide página 97).

[13] nwaits1 é o número de Data Sets que enviam dados para o Data Set corrente (informação obtida com o Map Manager, vide página 97).

TChkTask	rtr	EReadAccess	NumDataSets
	Ro	EReadAccess	NumDataSets
	PRo	EWriteAccess	
	ptq	EReadAccess	NumDataSets
	Status	EWriteAccess	
TCGTask7	NP	EReadAccess	
	p	EReadAccess	nwaits
	q	EReadAccess	nwaits1
TVecUpdateTask	NP	EReadAccess	
	DestObj	EWriteAccess	
TDblUpdateTask	DestObj	EWriteAccess	

Como acima está a descrição das dependências de dados de cada tarefa, a seguir será repetido o esquema de divisão do algoritmo da pagina 93, para que possamos associar um ao outro.

```
Tarefa 1
                M^k z^k = r^k
                ro^k = (r^k).(z^k)
                  ->    Tarefa 2
                              soma ro^k ao ro

        Tarefa 3
                beta = ro/ro(i-1)
                se i = 0
                        p^k = z^k
                senão
                        p^k = z^k + beta p^k(i-1)
                fimse
                  ->    Tarefa 4
                              atualiza p nos demais processadores

        Tarefa 5
                q^k = A^k p^k
                  ->    Tarefa 6
                              atualiza q nos demais processadores
                  ->    Tarefa 7
                              calcula contribuição ptq^k = (p^k).(q^k)
                        ->    Tarefa 7a
                                    soma ptq^k ao ptq global

        Tarefa 8
                alfa = ro/p^tq
                r = r(i-1) - alfa q
                  ->    Tarefa 9
                              soma (rtr = (r^k). (r^k)) a rtr

        Tarefa 10
                x^k = x^k(i-1) + alfa p^k

        Tarefa 11
                verificação da convergência
```

onde o índice k indica o número do Data Set e rtr, ptq, ro e $ro_{(i-1)}$ são coeficientes globais.

A seqüência das tarefas do algoritmo de gradiente conjugado pré condicionado é garantida pela dependência que cada uma delas apresenta de versões de dados. Por exemplo, a tarefa 3 só entra em execução se a tarefa 1 tiver preenchido o vetor z. Sendo assim, garantimos que a tarefa 3 será executada após a tarefa 1, registrando a dependência da tarefa 3 para a versão 1 do vetor z, e fazendo com que a tarefa 1 incremente a versão de z quando for executada. Veja:

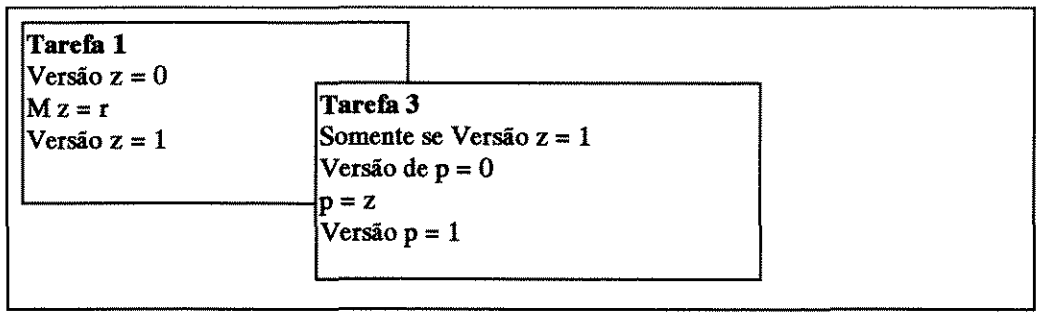


Figura 20- Exemplo de organização de tarefas por dependência de versão de dados

Na realidade a tarefa 3 depende não só da versão de z mas também de outros dados como os coeficientes globais Ro e PRo.

6.4.1 Dependências e controle das versões dos dados

Genericamente, cada tarefa do algoritmo de gradiente conjugado mantém uma variável fVersion que é incrementada a cada iteração. Com fVersion calcula-se a versão dos dados dos quais a tarefa depende para então passar este valor ao método AddDataDepend no final do método Execute, fazendo com que esta tarefa seja executada somente quando as novas versões dos dados estejam disponíveis, ou seja, na próxima iteração.

6.4.1.1 Classe TCGStartTask

A tarefa TCGStartTask depende de acesso a leitura aos dados necessários na criação das tarefas contínuas (vide Tabela 3, página 121). Ela é uma tarefa para que se possa escolher o processador onde ela será executada, se não fosse esse fato a mesma poderia ser uma função convencional.

A dependência das tarefas de acesso a leitura aos dados independente de versão, são registradas na execução da tarefa TCGStartTask e não são registradas novamente a cada execução.

As dependências de cada tarefa são alteradas por elas próprias, no momento de sua execução, somente quando deve ser registrado que a tarefa passa a depender de uma nova versão do dado, ou seja a tarefa que dependia da versão "x" do dado passa agora a depender da versão "x+1", por exemplo.

6.4.1.2 Classe TCGTask1

A tarefa 1 (classe TCGTask1) calcula o vetor do resíduo pré condicionado z. O momento da sua execução é determinado pelas seguintes dependências:

- acesso a leitura da versão nc do vetor de resíduo r, dependência que garante que a tarefa 1 só será executada novamente depois que a tarefa 8, que atualiza o resíduo, for executada;

- acesso a escrita do vetor z;

- acesso a leitura do versão nc do Status, dependência que garante que a tarefa 1 só será executada novamente no próximo ciclo do algoritmo;

onde:

nc = número do ciclo.

A classe TCGTask1 possui um dado fVersion onde é armazenado o número da execução da tarefa. fVersion é usado para registrar, ao final de cada execução, as dependências das novas versões de r e de Status.

6.4.1.3 Classe TCGTask3

A tarefa 3 calcula o vetor de busca p. Sua execução tem as seguintes dependências:

- acesso a leitura da versão nc+1 do vetor z, dependência que garante que a tarefa 3 será executada depois da tarefa 1;

- acesso a escrita do vetor p;

- acesso a leitura da versão "(nc+1) * NumDataSets"[14] do coeficiente Ro, essa dependência garante que a tarefa 3 será executada somente após terem sido computadas as contribuições de todos os Data Sets para o coeficiente global Ro. Cada tarefa 1 calcula o z do Data Set e a participação do Data Set para o coeficiente Ro (r . z), essa participação é somada ao coeficiente global Ro por uma tarefa

[14] nc + 1 pois nc começa em zero.

TDblUpdateTask que incrementa a versão de Ro, daí o valor da versão ser "(nc+1) * NumDataSets". Com essa dependência garante-se um sincronismo entre todos os Data Sets.

- acesso a leitura da versão nc do coeficiente PRo, dependência que garante que a tarefa 3 será executada depois que o coeficiente PRo for atualizado;

- acesso a leitura do versão nc do Status, dependência que garante que a tarefa 3 será executada novamente apenas no próximo ciclo do algoritmo, pois a versão do Status só muda uma vez por ciclo.

A tarefa 3 também possui um dado fVersion que é usado na atualização das dependências acima descritas.

A tarefa 3 cria e submete uma tarefa TVecUpdateTask para atribuir os valores de p das equações desse Data Set nos demais Data Sets que precisem desses valores (vide item "Classe TCGTask3", página 105).

6.4.1.4 Classe TCGTask5

A tarefa 5 calcula o vetor q que é o resultado do produto da matriz A pelo vetor p. O momento da sua execução é determinado pelas seguintes dependências:

- acesso a escrita ao vetor q;

- acesso a leitura a versão "(nc+1) * nwaits" de p, onde nwaits é o número de Data Sets para onde este Data Set envia dados (número de espera), ou seja, a tarefa 5 deve rodar somente depois que todos os Data Sets, que são donos de alguma das equações que esse Data Set trabalha, tenham enviado suas contribuições para o vetor p. Essas contribuições são enviadas pelas tarefas TVecUpdateTask criadas e submetidas pela tarefa 3, e cada uma dessas tarefas incrementam a versão de p;

- acesso a leitura do versão nc do Status, dependência que garante que a tarefa 5 será executada novamente apenas no próximo ciclo do algoritmo.

A tarefa 5 cria e submete uma tarefa TVecUpdateTask para adicionar o vetor q aos vetores q dos demais Data Sets que precisam dessa contribuição (vide item "Classe TCGTask5", página 109).

Também é na tarefa 5 que é criada a tarefa 7 que calcula "p . q".

6.4.1.5 Classe TCGTask7

É a tarefa 7 que calcula a contribuição do Data Set para o coeficiente global ptq (produto "p . q"). A tarefa 7 tem o seu momento de execução determinado pelas seguintes dependências:

- acesso a leitura da versão "(nc+1) * nwaits" de p, (segunda dependência da tarefa 5, vide página 127);

- acesso a leitura da versão "(nc+1) * nwaits1" de q, onde nwaits1 é o número de Data Sets de onde este Data Set recebe dados, ou seja, a tarefa 7 deve rodar somente depois que todos os Data Sets, que trabalham com alguma das equações que esse Data Set é dono, tenham enviado suas contribuições para o vetor q. Essas contribuições são enviadas pelas tarefas TVecUpdateTask criadas e submetidas pela tarefa 5, e cada uma dessas tarefas incrementam a versão de q;

A tarefa 7 cria e submete ao **TM** uma tarefa TDblUpdateTask (tarefa 7a do esquema) para enviar a contribuição desse Data Set para ser somada ao coeficiente global ptq (produto "p.q").

6.4.1.6 Classe TCGTask8 (type 0)

A tarefa 8 (TCGTask8, type 0) é responsável pelo cálculo do resíduo r. O momento da sua execução é determinado pelas seguintes dependências:

- acesso a leitura da versão "(nc+1) * NumDataSets" do coeficiente Ro, essa dependência garante que a tarefa 8 será executada somente após terem sido computadas as contribuições de todos os Data Sets para o coeficiente global Ro (como a terceira dependência da tarefa 3, vide página 126);

- acesso a leitura da versão "nc * NumDataSets" do coeficiente ptq, como a dependência anterior, essa garante que a tarefa 8 será executada somente após terem sido computadas as contribuições de todos os Data Sets para o coeficiente global ptq;

- acesso a leitura da versão nc do Status, dependência que garante que a tarefa 8 será executada novamente apenas no próximo ciclo do algoritmo;

- acesso a leitura da versão "(nc+1) * nwaits1" de q, onde nwaits1 é o número de Data Sets de onde este Data Set recebe dados (como a segunda dependência da tarefa 7, vide página 128);

A tarefa 8 calcula a contribuição do Data Set para o produto "r . r" e cria uma tarefa do tipo TDblUpdateTask para que essa contribuição seja acumulada no coeficiente global rtr.

6.4.1.7 Classe TCGTask8 (type 1)

A tarefa 10 (TCGTask8, type 0) é responsável pelo cálculo da solução aproximada x. Além das três primeiras dependências da tarefa 8, o momento da execução da tarefa 10 é determinado pela seguinte dependência:

- acesso a leitura da versão "(nc+1) * nwaits1" de p, onde nwaits1 é o número de Data Sets de onde este Data Set recebe dados (como a segunda dependência da tarefa 5, vide página 127).

6.4.1.8 Classe TChkTask

A tarefa 11 (Classe TChkTask) faz a verificação da convergência e atualiza os valores dos coeficientes globais (vide item **Erro! A origem da referência não foi encontrada.**, página **Erro! Indicador não definido.**). Ela é executada ao final de cada ciclo e o momento de sua execução é determinado pelas seguintes dependências:

- acesso a escrita à flag Status;

- acesso a escrita ao coeficiente PRo;

- acesso a escrita da versão "(nc+1) * NumDataSets", do coeficiente rtr;

- acesso a escrita da versão "(nc+1) * NumDataSets", do coeficiente ptq;
- acesso a escrita da versão "(nc+1) * NumDataSets", do coeficiente Ro.

Capítulo 7

7. Paralelização do cálculo de elementos finitos

Uma das metas desse trabalho é o desenvolvimento de um programa de elementos finitos que fosse o mais paralelo o possível, ou seja, que cada processador trabalhasse em paralelo desde a leitura dos dados e montagem das malhas de elementos finitos, até a solução usando o método do gradiente conjugado pré condicionado e pós processamento. Para tanto foi incorporada ao ambiente PZ uma classe de análise parcial de problemas de elementos finitos, além de classes de malhas parciais (geométrica e computacional).

Para efetivar a leitura dos dados e montagem das malhas em paralelo, foram criadas várias classes/tarefas com base no ambiente OOPAR, algumas de execução local em cada processador e outras que são transmitidas de processador para processador para atualizar informações sobre a interface entre as malhas parciais (informações de fronteira).

7.1 Introdução

Como uma das metas desse trabalho era expandir o ambiente PZ de forma que ele pudesse tratar problemas de elementos finitos com processamento paralelo desde a leitura dos dados até a geração dos resultados, foi necessária a criação de uma sistemática que pudesse determinar como as malhas parciais se relacionam, em tempo de execução e, principalmente, em paralelo.

No ambiente PZ, a partir de descrições em arquivos de entrada é criada a malha geométrica, e a partir desta, a malha computacional é criada. No ambiente PZ paralelo isto não poderia ser diferente, cada malha parcial geométrica lê suas informações de entrada e cria seus nós e elementos para depois dar origem a malha parcial

computacional. Com as malhas parciais geométricas e computacionais geradas (em paralelo) surge uma questão: cada malha computacional deve saber com quais outras malhas se relaciona, para que, quando da solução iterativa do sistema de equações, as malhas parciais possam trocar os resíduos calculados para os nós na interface entre as malhas. Nos referiremos às informações necessárias para a troca de dados sobre os nós na interface entre as malhas como "*informações de fronteira*".

7.2 Determinação das informações de fronteira

A determinação das informações de fronteira começa pelas malhas parciais geométricas para, a partir destas, determinar as informações de fronteira das malhas computacionais. Isto deve-se ao fato de que cada nó geométrico possui uma identificação única (*id*), independente da malha parcial (ou das malhas parciais) da qual ele faça parte. Um nó computacional, por sua vez, é criado em tempo de execução partindo-se da geometria da malha parcial, independente do arquivo de entrada. Disso conclui-se que um nó computacional na interface entre duas malhas é criado em ambas com ids diferentes (Figura 21).

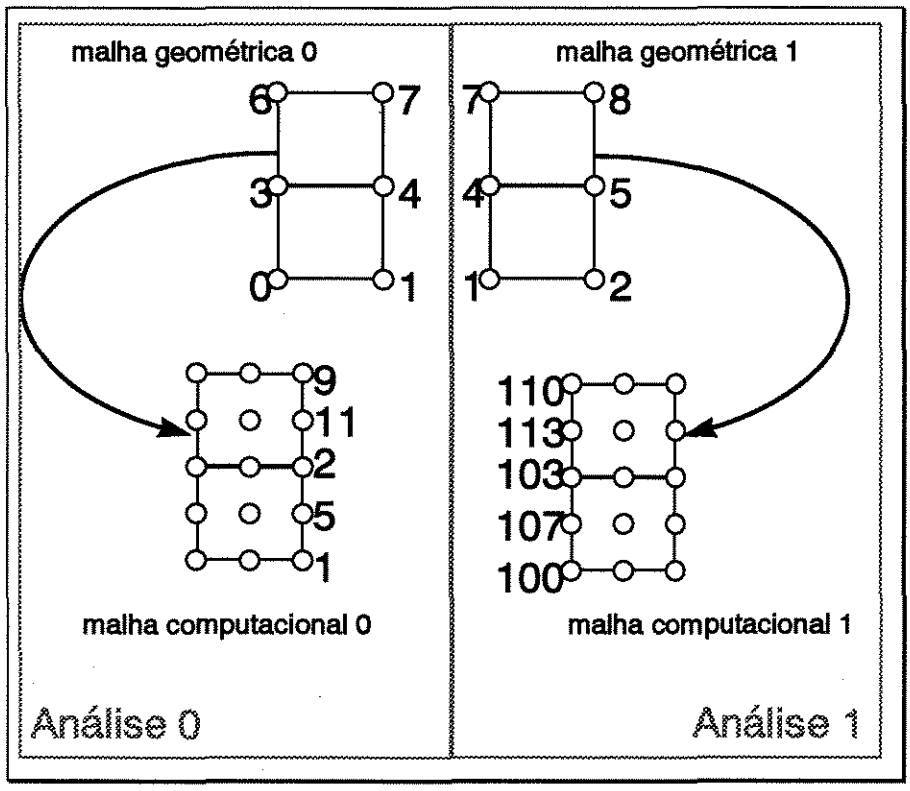


Figura 21 - Análises parciais antes da determinação das informações de fronteira

7.2.1 Malha geométrica

A determinação das informações de fronteira da malha parcial geométrica é feita em dois passos:

1) **BuildGeoElCorrespondence**

Determinação dos nós de interface, montagem de estruturas com os elementos e lados onde estão esses nós. Com essas informações a malha vizinha monta a correspondência (vizinhança) entre os elementos de uma malha parcial e elementos de outras malhas (BuildExternalConnectivity).

2) **BuildGeoNodCorrespondence**

Determinação da correspondência entre os nós de uma malha parcial (GetBoundaryNodes) e nós de outras malhas (SetBoundaryNodes).

7.2.1.1 BuildGeoElCorrespondence

A malha parcial geométrica vasculha sua lista de elementos e separa todos os elementos que, em pelo menos um dos lados, não possuam nem um apontador para o elemento vizinho nem uma condição de contorno, pois isso indica que, na verdade, esse lado do elemento está conectado a um elemento pertencente a uma outra malha parcial.

Para cada lado de elemento que não está conectado com um outro elemento conhecido (elemento interno) nem está com uma condição de contorno, cria-se uma estrutura com os seguintes dados:

```
struct TNeighbourInfo {
long fElementId; //Id do elemento geométrico
LongVec fNodeId; //Vetor com os ids dos nós do lado
int fSide; //Número do lado
};
```

Cada malha passa a sua lista de estruturas TNeighbourInfo para todas as demais. Cada malha parcial geométrica, ao receber uma lista de estruturas TNeighbourInfo executa o seguinte procedimento para montar a conectividade externa (método BuildExternalConnectivity):

- Com sua lista de estruturas TNeighbourInfo a malha parcial compara os lados dos seus elementos de interface (descritos nas estruturas TNeighbourInfo), com os lados dos elementos de interface da malha que enviou a lista de TNeighbourInfo.

- Para cada par de estruturas TNeighbourInfo com ids de nós coincidentes cria-se uma estrutura descrevendo a conexão, estrutura TExternElementList:

```
struct TExternElementList {
long fElementSide;  // Lado do elemento conectado ao elemento vizinho
long fNeighbourSide;// Lado do vizinho conectado ao elemento
long fNeighbour;    // Id do elemento vizinho
long fGridId;       // Id da malha do elemento vizinho
TExternElementList *fNext;
};
```

Esta estrutura é na verdade uma lista ordenada contendo informações sobre os elementos externos conectados a um elemento interno da malha parcial geométrica.

Ao final deste procedimento, a malha parcial geométrica estará com uma lista de estruturas TExternElementList para cada um dos seus elementos de interface, descrevendo dessa forma todas as conexões de um lado de um elemento com um lado de um elemento externo.

7.2.1.2 BuildGeoNodCorrespondence

A malha parcial geométrica monta um vetor com todos os ids dos seus nós de interface (método GetBoundaryNodes). Esse vetor é enviado para as demais malhas parciais.

Uma malha parcial, ao receber os ids dos nós de interface de uma outra malha, vasculha sua lista de nós e, caso encontre um nó cujo id conste no vetor de ids recebido, este é então adicionado a lista de nós externos da malha (método SetBoundaryNodes).

7.2.2 Malha computacional

A determinação das informações de fronteira da malha parcial computacional é feita também em dois passos, a partir das informações de fronteira da malha parcial geométrica, são eles:

1) **BuildCompElCorrespondence**

Usando as estruturas TExternElementList montadas pela malha geométrica, a malha computacional monta estruturas com os ids dos nós computacionais e geométricos. Estas estruturas são enviadas à malha vizinha que alterará no nó computacional, as informações id do nó e id da malha computacional responsável por este nó.

2) **BuildDofNodCorrespondence**

Cada malha envia às demais vetores com os ids dos seus nós geométricos e computacionais. Ao receber estes vetores, a malha computacional verifica altera no nó computacional, as informações id do nó e id da malha computacional responsável por este nó.

7.2.2.1 BuildCompElCorrespondence

A malha parcial computacional varre a lista de estruturas TExternElementList da sua malha geométrica e, para cada malha vizinha, é enviada uma lista de estruturas como a seguinte:

```
struct TCompElSideDefinition {
long fTargetGeoElId;     // Id do elemento destino (na malha vizinha)
long fTargetSide;        // lado do elemento vizinho
LongVec fGeonodId;       // Ids dos nós geométricos no lado
LongVec fDofnodId;       // Ids dos nós computacionais no lado
};
```

Essa lista de estruturas TCompElSideDefinition é enviada para a malha vizinha que, ao recebe-la, executa os seguintes passos (Figura 22):

- Localiza-se o elemento geométrico de destino, identificado por fTargetGeoElId;
- A partir do elemento geométrico, o elemento computacional é localizado, pois o elemento geométrico conta com um apontador para seu respectivo elemento computacional;
- Para cada um dos nós do elemento computacional no lado indicado por fTargetSide as informações Id (id do nó) e GridId (id da malha computacional responsável por este nó) são alteradas com o seguinte critério: A malha com o menor id ficará responsável pelo nó computacional. Dessa forma o id do nó será alterado para ser o id

em fDofnodId, dado inicialmente na malha que ficará sendo responsável por este nó, malha origem.

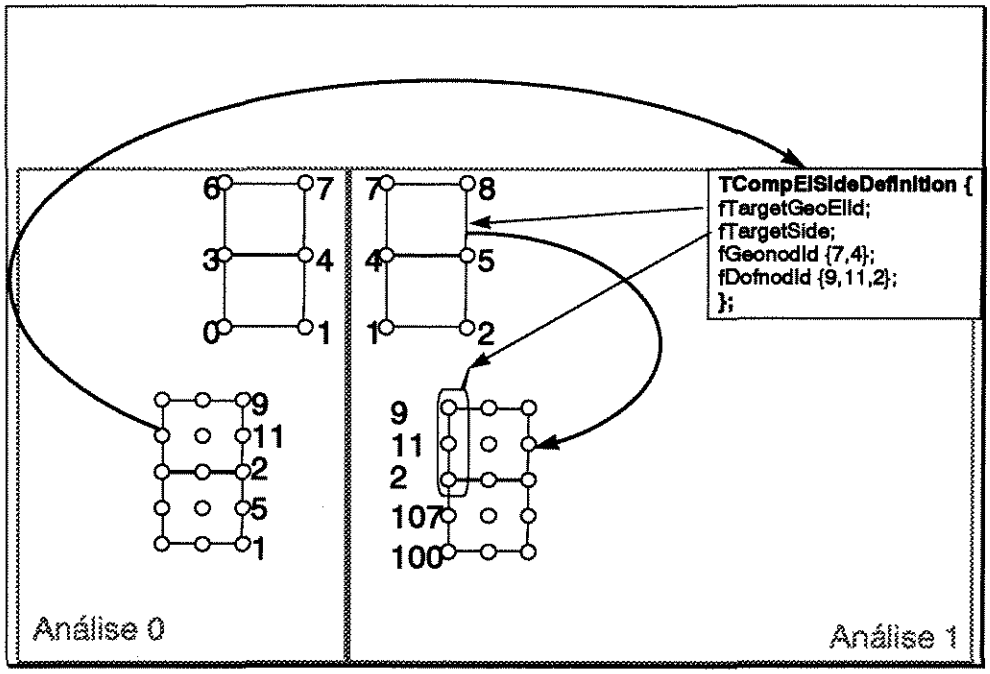


Figura 22 - Alteração dos nós computacionais na fronteira

- É criada uma estrutura composta por um vetor por malha vizinha, sendo que cada vetor contém os ids dos nós da respectiva malha vizinha, presentes na fronteira (estrutura fBoundaryNodeIdByGrid);

- É criada uma outra estrutura composta por um vetor por malha vizinha, sendo que cada vetor contém os ids dos elementos presentes na interface com a respectiva malha vizinha (estrutura fBoundaryElementIdByGrid).

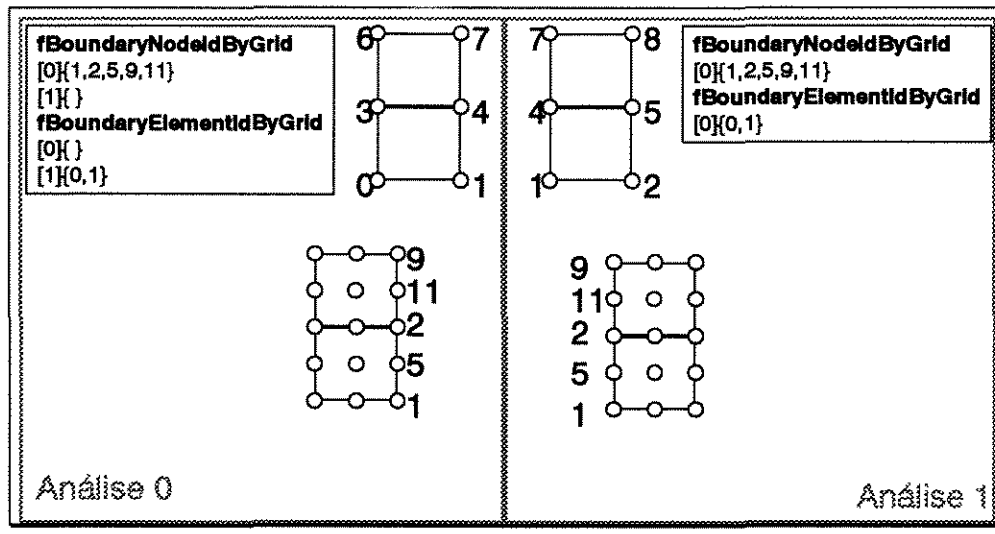


Figura 23 - Malhas parciais e informações de fronteira (configuração final)

7.2.2.2 BuildDofNodCorrespondence

Em configurações de malhas muito irregulares os procedimento descritos em BuildCompElCorrespondence não bastariam para montar, de forma precisa, o mapa fBoundaryElementIdByGrid. Existem configurações de malhas possíveis onde procurar uma malha vizinha apenas pelos lados dos elementos não basta. A configuração da figura abaixo é um bom exemplo:

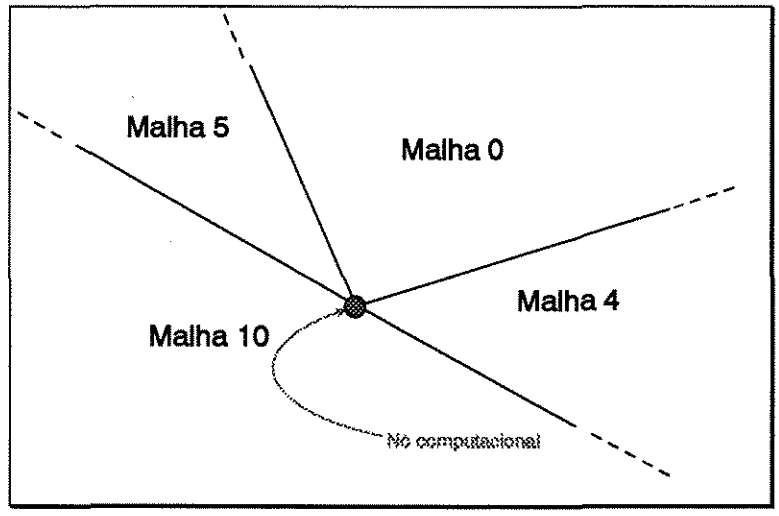


Figura 24 - Configuração irregular de malhas parciais

Numa configuração como a da Figura 24, após o acerto de nós de interface pelos lados (BuildCompElCorrespondence), o mapa fBoundaryElementIdByGrid da malha 10 indicaria a malha 4 como malha responsável pelo nó mostrado na figura, o que é um erro pois as malhas 4 e 5 indicariam a malha 0 como responsável pelo nó. Daí vem a

necessidade de uma malha enviar os ids dos seus nós de interface para que as demais procedam um segundo acerto nos seus mapas fBoundaryElementIdByGrid, o que é descrito a seguir.

Usando o mapa fBoundaryElementIdByGrid, cada malha monta dois vetores com os ids dos nós computacionais e geométricos, e envia a cada uma das demais malhas computacionais. Cada malha computacional, ao receber estes vetores, executa o seguinte para cada id nos vetores:

- Localiza o nó geométrico cujo id está no vetor e verifica se o nó computacional referenciado por eles ainda possui um GridId maior que o id da malha que enviou o vetor (GridId = id da malha responsável pelo nó computacional);

- Se o GridId do nó localizado ainda é maior, altera-se este para o id da malha que enviou o vetor de ids, indicando dessa forma, que a malha que enviou o vetor será a responsável pelo nó. Depois disso o id do nó computacional é mudado para o id constante do vetor recebido;

- Se o nó alterado ainda não constava do mapa fBoundaryNodeIdByGrid, este é então incluído. O id desse nó entra para um vetor que é retornado para a malha que desencadeou o processo para que esta repita o processo de verificação.

Todo o procedimento acima descrito foi concebido para garantir que, por mais irregular que seja a configuração de malhas, todos os nós computacionais de interface tenham suas informações alteradas de modo a garantir a consistência das informações de fronteira.

Para uma configuração regular de malhas, o procedimento acima não vai acarretar nenhuma mudança ao mapa fBoundaryElementIdByGrid. No entanto, por mais irregular que a configuração seja, espera-se que, executando o procedimento acima duas vezes como o descrito, o mapa fBoundaryElementIdByGrid alcance a forma correta.

7.3 Solução por gradiente conjugado pré condicionado paralelo

Com o problema de elementos finitos dividido em malhas parciais, a solução é feita usando o algoritmo de gradiente conjugado pré condicionado paralelo descrito no

capítulo 6. Cada malha parcial computacional é equivalente a um Data Set do algoritmo. O número do Data Set é trocado pelo id da malha parcial e o número global de uma equação é substituído pelo id do nó computacional que dá origem à esta equação. Dessa forma temos uma organização para um problema de elementos finitos equivalente à organização proposta no capítulo 6, que é muito mais genérica. Seguindo esta padronização são criadas as estruturas TCGGlobalData e os seus dados, além dos mapas do Map Manager. Com estes objetos montados, as iterações do algoritmo correm normalmente como foi descrito no capítulo 6.

Cada malha parcial executa a montagem da sua matriz de coeficientes, bem como do seu vetor de carga, em paralelo.

As informações de fronteira são usadas na montagem em paralelo dos mapas do Map Manager. Este é então usado na montagem do vetor de carga global (primeiro resíduo global) e na matriz diagonal global D, ambos usados no algoritmo de gradiente conjugado pré condicionado.

7.3.1 Montagem dos dados do algoritmo PCG paralelo

Como primeiro passo cada análise parcial monta, em paralelo, seu vetor de carga e sua matriz de rigidez. A solução do sistema global é conseguida pela soma dos sistemas locais das análises parciais.

$$Kx = f \Rightarrow \sum_{i=1}^{n} K_i x_i = \sum_{i=1}^{n} f_i$$

onde i é o número da análise parcial e n é o total de análises parciais envolvidas.

Em seguida cada análise parcial envia o número de equações sob sua responsabilidade para ser acumulado no processador 0. O número global de equações é usado na verificação da convergência do algoritmo de gradiente conjugado.

Nesse ponto os dados locais de cada análise parcial, necessários para o algoritmo PCG paralelo, já estão parcialmente disponíveis. Considerando que cada análise parcial compõe um Data Set, já estão disponíveis os seguintes dados (estrutura TCGGlobalData, página 99):

- z — parte do resíduo pré condicionado tratada pelo Data Set, inicialmente com zeros.
- r — parte do resíduo tratada pelo Data Set. Trata-se do vetor de carga, que já está montado para os nós internos, mas ainda precisa da contribuição das malhas vizinhas (nós de interface).
- p — parte de p tratada pelo Data Set, inicialmente com zeros.
- x — parte da solução tratada pelo Data Set, inicialmente com zeros.
- A — parte da matriz de coeficientes tratada pelo Data Set. Matriz de rigidez K_i já disponível.
- q — parte de q tratada pelo Data Set, inicialmente com zeros.
- NP — vetor com tamanho igual ao número de equações do seu Data Set, onde é armazenado o número do Data Set "dono" de cada equação, ou seja, o id da malha computacional que é responsável pela equação.
- GNodeNumber — vetor com o mesmo tamanho de NP, onde é armazenado o id do nó computacional referente à equação. Esse número é importante para a troca de dados entre os Data Sets (montagem do Map Manager).

Ainda precisam ser montados os seguintes dados:

- r — parte do resíduo tratada pelo Data Set, ainda precisa da contribuição das malhas vizinhas (nós de interface).
- D — parte da diagonal da matriz de rigidez global K tratada pelo Data Set.

Cada análise parcial cria e/ou submete cada um dos vetores e matrizes citados acima, sendo criada uma estrutura TCGGlobalData com os ids de cada dado.

A parte do resíduo inicial referente aos nós de interface, bem como a matriz diagonal D, dependem da montagem do Map Manager, pois é através do Map Manager

que a contribuição de um Data Set chega aos demais, ou em outras palavras, é através do Map Manager que a contribuição de uma análise parcial chega até as análises vizinhas.

Por depender da criação do Map Manager, r e D só serão completamente montados na primeira iteração do algoritmo PCG paralelo.

7.3.2 Montagem dos mapas do Map Manager em paralelo

Cada análise parcial guarda, nas suas informações de fronteira, um vetor para cada malha vizinha com os ids dos nós na interface entre uma e outra malha (mapa fBoundaryNodeIdByGrid). O Map Manager por sua vez é mais genérico, independendo da configuração do problema de elementos finitos, pois o mesmo não está diretamente associado a idéia de nós.

Para utilizar o algoritmo de gradiente conjugado pré condicionado descrito no capítulo 6, cada análise parcial utiliza seu mapa fBoundaryNodeIdByGrid e monta o mapa de envio (fSendMap) do Map Manager. Para cada posição do mapa fSendMap é enviada uma tarefa para montar no processador do Data Set destino a mesma posição do mapa de recebimento fReceiveMap. Dessa forma, os dois mapas do Map Manager, únicos para todo o ambiente, são montados em paralelo.

Sendo assim, os procedimento descritos em "6.2.1 Criação dos Mapas", página 97, são executados para o problema de elementos finitos da seguinte forma:

Sendo G a malha montando suas ocorrências de fSendMap e N cada uma das suas vizinhas, são executados os seguintes passos:

- Recupera-se um vetor dofvec que é o vetor do mapa fBoundaryNodeIdByGrid com os ids dos nós de N presentes na malha G.

- Monta-se o vetor fSendMap(G, N) com os índices locais de G para os nós representados em dofvec.

- Envia-se uma tarefa para ser executada no processador de N que, a partir do vetor dofvec, monte o vetor fReceiveMap(G, N) com os índices locais de N das equações de cada nó representado em dofvec.

Os índices locais de um nó são conseguidos com métodos das classes TDofNod e TBlock, do ambiente PZ.

Como cada análise parcial monta a sua parte do mapa fSendMap em paralelo e recebe tarefas para montar partes de fReceiveMap que também são executadas em paralelo, podemos dizer que todos mapas do Map Manager são montados de forma distribuída pelos vários processadores envolvidos, diferente do descrito inicialmente no item 6.2.1.

7.3.3 Início do algoritmo PCG paralelo

Diferente do descrito no capítulo 6, a criação e submissão das tarefas de execução contínua do algoritmo PCG paralelo é feita de forma paralela e não mais por uma TCGStartTask, ou seja, cada análise parcial (Data Set) cria e submete suas tarefas de execução contínua. Isto só é feito depois que os mapas do Map Manager estejam completamente montados, e as iterações do algoritmo começam tão logo todas as tarefas tenham sido submetidas. A tarefa de verificação da convergência continua sendo criada no início do programa.

7.3.3.1 Outras alterações no algoritmo PCG paralelo descrito no capítulo 6

Para trabalhar com um problema de elementos finitos (ambiente PZ), foram feitas as seguintes alterações no algoritmo PCG paralelo:

- As tarefas 8 e 10 foram unificadas, ou seja, não se criam mais duas instâncias da classe TCGTask8 por Data Set. Isto foi feito pois ambas as instâncias deveriam ser executadas na mesma iteração, o que passou a não ocorrer para um programa PZ.

- A tarefa 1 (Classe TCGTask1) cria tarefas TVecUpdateTask para enviar a contribuição do vetor de cargas e da diagonal. Na forma inicial, tanto a diagonal como o vetor de cargas eram montados a priori, o que não é possível devido a montagem paralelizada do problema de elementos finitos.

- Como o Map Manager é criado vazio e montado em paralelo, as tarefas dependem da versão 1 dos mapas.

7.4 Geração dos resultados

Quando o algoritmo PCG converge, cada Data Set envia sua contribuição do vetor de solução x, para os Data Sets vizinhos. Sendo assim, a tarefa que gera arquivos de pós processamento usando o ambiente PZ só é executada quando todos as atualizações de x tenham sido feitas.

Cada análise parcial gera um arquivo de pós processamento com os dados da sua malha parcial. O problema global pode ser visualizado combinando-se todos os arquivos parciais.

7.5 Análise de elementos finitos em paralelo (implementação)

A análise parcial de elementos finitos é criada sem saber, a princípio, com quais outras análises parciais se relaciona, ou seja, quais são as malhas vizinhas. Seguindo uma seqüência lógica de procedimentos, cada análise parcial troca informações com todas as demais e monta suas informações de fronteira.

Dado a extensão do ambiente PZ, não se fez a transformação de todas as classes em derivações de TSaveable para que fosse possível colocar todos os dados da análise parcial (e ela própria) sob o controle do **DM**, e efetuar uma transferência desses dados de um processador para outro quando fosse conveniente. Diferente disso, optou-se por criar uma análise parcial e tarefas que fossem fixas no processador onde foram criadas e interagissem com o ambiente, e as demais análises parciais, por intermédio de outras classes e tarefas, estas sim transferíveis de um processador para outro.

Partindo das classes e da estrutura já existente no ambiente PZ, o que se fez foi:

- estender a classe de análise de forma a trabalhar com um problema parcial e cooperar na solução do problema global;
- criar classes e tarefas para comunicar dados sobre as análises parciais para as demais análises;
- criar tarefas para garantir a execução sincronizada da montagem e análise do problema de elementos finitos.

7.5.1 Objetos transmitidos de um processo para os demais

Para a montagem das informações de fronteira não são enviados de um processador para outro nenhum objeto PZ do tipo nó ou do tipo elemento. Ao invés disso cada processo monta e envia aos outros processadores estruturas mais simples com as informações sobre os elementos e nós presentes no contorno da malha parcial.

7.5.1.1 Classe TNeighbourInfo

Derivada de TSaveable, essa classe leva de uma malha geométrica para as demais a lista dos nós em um dos lados de um elemento geométrico do contorno.

Dados:

- long fElementId — id do elemento ao qual a estrutura TNeighbourInfo se refere;
- int fSide — número do lado do elemento ao qual a estrutura TNeighbourInfo se refere;
- LongVec fNodeId — vetor contendo os ids dos nós presentes no lado fSide do elemento em questão;

7.5.1.2 Classe TCompElSideDefinition

Também derivada de TSaveable, essa estrutura é montada a partir das informações de fronteira da malha parcial geométrica e leva os ids dos nós (tanto geométricos como computacionais) conectados a um lado de um elemento do contorno de uma malha parcial para as demais.

Dados:

- long fTargetSide — lado do elemento destino;
- LongVec fGeonodId — vetor com os ids dos nós geométricos presentes no lado em questão;
- LongVec fDofnodId — vetor com os ids dos nós computacionais presentes no lado em questão;

7.5.1.3 Classe TPAnalysisGlobalData

Estrutura com dados do problema global de elementos finitos que cada análise parcial precisa para montar as informações de fronteira e solucionar o problema com o algoritmo de gradiente conjugado.

Dados:

- long fGridId — id dado à malha parcial computacional referente à estrutura TPAnalysisGlobalData;
- int fProcNumber — número do processador onde a análise deve residir;
- long fNPId — id do vetor com o número do processador onde cada análise parcial reside;
- long fGGIdsId — id do vetor com os ids dados à cada malha parcial geométrica;
- long fCGIdsId — id do vetor com os ids dados à cada malha parcial computacional;
- long fDiagStructId — id do vetor com os ids do vetor diagonal de cada malha parcial;
- long fGNumEqsId — id do inteiro onde será acumulado o número de equações do problema global de elementos finitos (o número de equações é usado pela tarefa *TChkTask* do algoritmo de gradiente conjugado que verifica a convergência da solução);

7.5.2 Objetos locais de cada processo

Os objetos de análise parcial, malha geométrica e malha computacional, são inerentes a um único processo não podendo então serem transmitidos de um processador para outro.

7.5.2.1 Classe TExternElementList

Trata-se de uma lista ordenada dos vizinhos conectados a um elemento de uma malha parcial geométrica.

Dados:

- long fElementSide — lado do elemento referente a presente estrutura que está conectado a um elemento de outra malha parcial (vizinho);
- long fNeighbourSide — lado do elemento vizinho conectado ao elemento em questão;
- long fNeighbour — id do elemento vizinho;
- long fGridId — id da malha parcial responsável pelo elemento vizinho;
- TExternElementList *fNext — ponteiro para a próxima estrutura TExternElementList da lista.

7.5.2.2 Classe TPartialGeoGrid

Trata-se de uma derivação da classe TGeoGrid do ambiente PZ, que acrescenta dados e métodos para determinação e controle das informações de fronteira da malha geométrica, mantendo toda a funcionalidade de TGeoGrid.

Dados:

- long fId — id da malha geométrica parcial;
- LongVec fGridIds — vetor com os ids de todas as malhas vizinhas;
- TVoidPtrMap fBoundElements — lista onde são armazenadas estruturas do tipo TExternElementList para cada elemento do contorno da malha parcial;
- LongVec fBoundNodes — vetor com os ids de todos os nós do contorno.

Principais métodos :

- void GetBoundaryNodes(LongVec &nodeIds)

  constrói uma lista com os ids dos nós que estão na fronteira entre uma ou mais malhas;

- void SetBoundaryNodes(long gridid, LongVec &nodeids)

  recebe informações sobre os nós de contorno de outra malha parcial e, caso encontre um nó cujo id conste no vetor de ids recebido, este é então adicionado a lista de nós externos da malha;

- void GetExternalElements(VoidPtrVec &neighbour)

  Retorna em neighbour ponteiros para estruturas TNeighbourInfo com informações sobre os elementos que possuem conexão com elementos de outras malhas parciais. Estas estruturas são enviadas para as demais malhas parciais;

- void BuildExternalConnectivity(long fromgridid, VoidPtrVec &Neighbours)

  Recebe um vetor de ponteiros para estruturas TNeighbourInfo e constrói uma lista de estruturas TExternElementList (lista fBoundElements).

7.5.2.3 Classe TPartialCompGrid

Uma derivação da classe TCompGrid do ambiente PZ, que mantém toda a funcionalidade desta, acrescentando dados e métodos para determinação e controle das informações de fronteira da malha parcial computacional.

Dados:

- long fId — id da malha parcial computacional;
- VoidPtrVec fExternalNodes — lista de ponteiros para os nós da malha que são responsabilidade de uma outra malha parcial, ou seja, "pertencem" à outra malha;
- LongVec fGridId — vetor contendo os ids das malhas parciais computacionais que fazem fronteira com a malha parcial computacional em questão;
- VoidPtrVec fBoundaryNodeIdByGrid — vetor de ponteiros para listas (tipo LongVec) contendo os ids dos nós do contorno de uma dada malha. Cada vetor dessa lista contém os ids dos nós

de uma dada malha presentes no contorno da malha parcial computacional em questão;

- VoidPtrVec fBoundaryElementIdByGrid — vetor de ponteiros para listas (tipo LongVec) contendo os ids dos elementos do contorno que possuem ao menos um nó pertencente à outra malha parcial computacional.

Tanto fBoundaryNodeIdByGrid como fBoundaryElementIdByGrid são organizadas de forma que em cada ocorrência haja uma lista com os ids dos nós e dos elementos, respectivamente, referentes a malha com o id armazenado na mesma posição da lista fGridId, o que resulta na organização ilustrada no seguinte exemplo:

Supondo que a malha com id 1 tenha no seu contorno 2 elementos com nós pertencentes à malha 0 (Figura 23, página 137), os elementos 0 e 1. Considere que os elementos 0 e 1 possuem em um dos seus lados os nós 1,2,5,9 e 11, da malha 0. A malha 1 ficaria então com seus dados de fronteira organizados da seguinte forma:

fGridId	fBoundaryNodeIdByGrid	fBoundaryElementIdByGrid
0	{1,2,5,9,11}	{0,1}

Principais métodos:

- void SetIdCorrespondenceBySide(long TargetGeogridId, VoidPtrVec &sideinfo)

  constrói uma estrutura TCompElSideDefinition para cada elemento do contorno conectado a TargetGeogridId e armazena um ponteiro para a estrutura criada na lista *sideinfo*;

- void GetIdCorrespondenceBySide(long FromCompGridId, VoidPtrVec &sideinfo)

  recebe uma lista de estruturas TCompElSideDefinition e acerta o id dos nós computacionais de contorno bem como a malha parcial computacional que é responsável por estes;

- void SetIdCorrespondenceByNode(LongVec &geonodid, LongVec &dofnodid)

  constrói a lista fBoundaryNodeIdByGrid com os ids dos nós computacionais de contorno conhecidos;

- void GetIdCorrespondenceByNode(long FromCompGridId, LongVec &geonodid, LongVec &dofnodid, LongVec &newgeonod, LongVec &newdofnod)

  recebe um vetor com os ids dos nós computacionais no contorno de uma malha diferente e estabelece qual malha será responsável por cada nó bem como os ids dos nós. Se for detectado a presença de um nó de uma outra malha que não tenha sido detectado antes, os ids destes nós são retornados em *newgeonod* e *newdofnod.*

- void CreateCGData(LongVec &DSetNumber, LongVec &GNodeNumber)

  Monta os vetores *DSetNumber* (número do Data Set responsável por cada equação) e *GNodeNumber* (número do nó responsável por cada equação) usados pelo algoritmo do método de gradiente conjugado pré condicionado.

7.5.2.4 Classe TPartialGridAnalysis

A classe TPartialGridAnalysis é uma derivação da classe TAnalysis do ambiente PZ e foi criada de forma a acrescentar dados e métodos para gerar e controlar as informações de fronteira das malhas parciais. Cada objeto TPartialGridAnalysis é associado a um Data Set do algoritmo de gradiente conjugado e é responsável tanto pela montagem dos mapas do Map Manager em paralelo como pela submissão das tarefas e conseqüente início da solução do sistema em paralelo. Todo o processo desenvolvido para a solução de problemas de elementos finitos em paralelo será descrito em detalhes mais adiante.

Dados:

- TPartialCompGrid *fPCGrid — ponteiro para a malha parcial computacional;
- TIntVec fNP — vetor de inteiros com os números do processador onde cada

malha parcial é tratada;

- LongVec fGGIds vetor de inteiros com os ids dados a cada malha geométrica;
- LongVec fCGIds vetor de inteiros com os ids dados a cada malha computacional;
- LongVec fPAGDIds vetor de inteiros com os ids das estruturas TPAnalysisGlobalData com dados de cada análise parcial;
- long fDataIdVecId id do vetor DataIdVec que contém os ids de cada estrutura TCGGlobalData;
- long fDiagId id do vetor com os valores da diagonal global do problema de elementos finitos, referentes às equações da malha parcial.

Os dados fNP, fCGIds, fGGIds, fPAGDIds e fDataIdVecId são criados logo no início do programa e submetidos ao **DM**, sendo passados a cada objeto TPartialGridAnalysis no seu construtor sendo este criado já no processador onde a análise vai residir. A malha computacional (fPCGrid) também é passada no construtor da classe mas não é um dado do **DM**.

Principais Métodos:

- virtual void BuildGeoElCorrespondence()

  construção da correspondência entre um elemento geométrico e o seu vizinho nas demais malhas parciais;

- virtual void BuildGeoNodCorrespondence()

  construção da correspondência entre um nó geométrico e o seu vizinho nas demais malhas parciais;

- virtual void BuildCompElCorrespondence()

  construção da correspondência entre um elemento computacional e o seu vizinho nas demais malhas parciais;

- virtual void BuildDofNodCorrespondence()

  construção da correspondência entre um nó computacional e o seu vizinho nas demais malhas parciais;

- virtual void CreateDiagonal(long DiagId)

  redimensiona o vetor onde será armazenado a diagonal referente as equações da malha parcial. Tal vetor é criado com dimensão igual a 0 no início do programa e passado a cada análise parcial via **DM**;

- void CreateCGData(long cgdId)

  cria e submete ao **DM** os vetores e matrizes do algoritmo de gradiente conjugado, partindo dos dados da análise parcial;

- void BuildMaps(long MMId)

  cria os vetores de envio do Map Manager e submete tarefas ao **DM** para que os vetores de recebimento do Mapa Manager sejam criados nos processadores de destino;

- void StartCG(long MMId, long cgdId)

  cria e submete ao **TM** todas as tarefas de execução contínua do algoritmo de gradiente conjugado;

- void Solution(void)

  Carrega as soluções da malha parcial computacional na mesma e gera arquivo de saída no formato do programa de visualização desejado.

Cada método da classe TPartialGridAnalysis deve ser executado seguindo uma ordem lógica. Para garantir essa ordem, cada método só pode ser chamado por uma tarefa, submetida ao **TM,** que tem o seu momento de execução determinado pela dependência de versão de dados e pela dependência da execução dos passos anteriores envolvidos na análise.

7.5.3 Procedimentos e Tarefas

Todo o processamento do problemas de elementos finitos é executado de forma totalmente paralelizada, desde a leitura dos dados e montagem das malhas até a solução do sistema de equações e impressão dos resultados. Cada processador usa um objeto da classe TPartialGridAnalysis para o controle das malhas e demais dados do problema. Todavia, os métodos envolvidos devem ser executados de forma harmônica em todos os processadores, o que impede que os métodos da análise sejam chamados sem tarefas sob o controle do OOPAR. Segue abaixo os passos a serem executados por cada análise parcial e os seus respectivos métodos:

1. Leitura da malha e construção da correspondência entre os elementos geométricos. Método BuildGeoElCorrespondence();

2. Construção da correspondência entre os nós geométricos. Método BuildGeoNodCorrespondence();

3. Construção da correspondência entre os elementos computacionais. Método BuildCompElCorrespondence();

4. Construção da correspondência entre os nós computacionais. Método BuildDofNodCorrespondence();

5. Criação do vetor de armazenamento da diagonal do problema parcial. Método CreateDiagonal(fDiagId);

6. Montagem da matriz de coeficientes Método Assemble();

7. Montagem dos mapas do Map Manager do algoritmo de gradiente conjugado. Método BuildMaps();

8. Criação dos dados e das tarefas do algoritmo de gradiente conjugado. Métodos CreateCGData() e StartCG();

9. Geração do arquivo de saída com o resultado do problema. Método Solution().

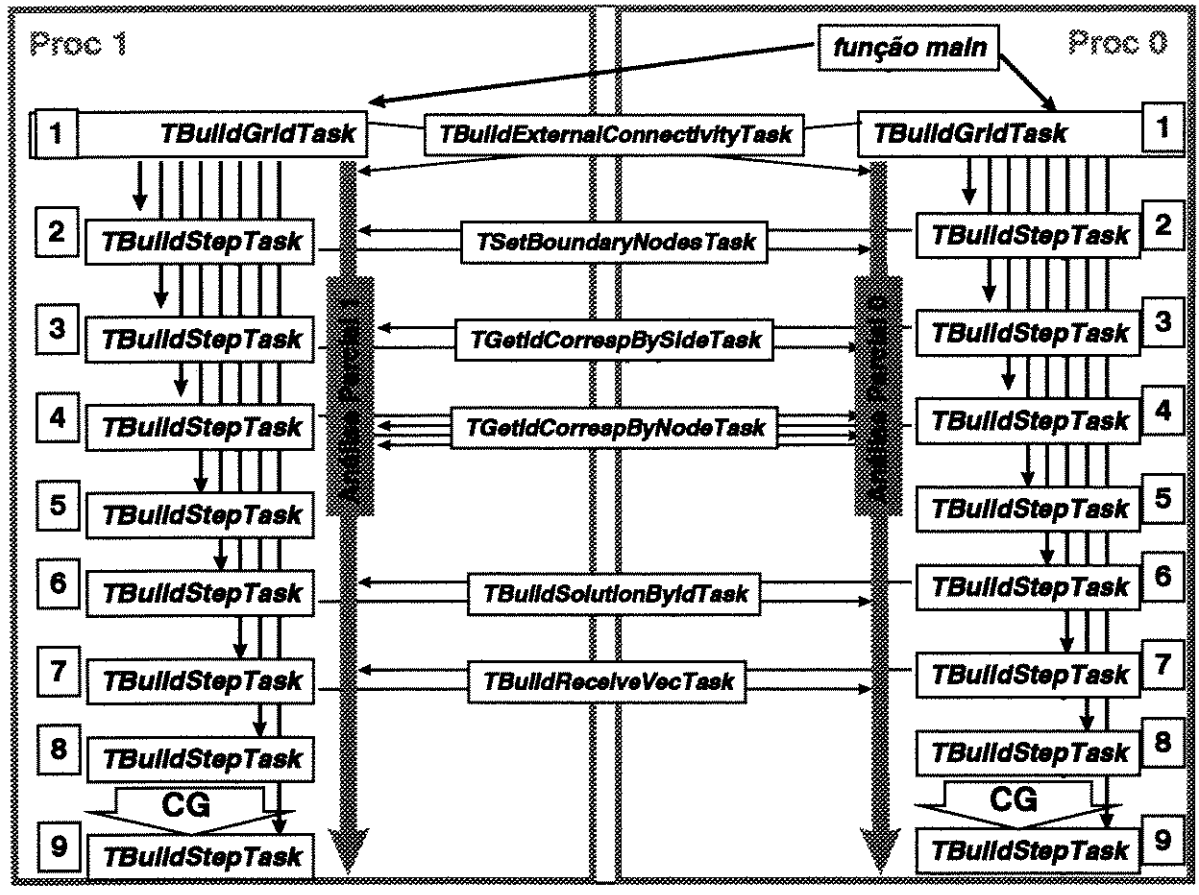


Figura 25 - Esquema de seqüência de tarefas para dois processadores

Para ativar os métodos da análise parcial do processador em momento oportuno, foi criada uma tarefa TBuildStepTask que ativa, no seu método execute, os métodos da classe TPartialGridAnalysis. TBuildStepTask recebe no seu construtor um objeto tipo TPartialGridAnalysis e um parâmetro step que definirá quais os métodos de TPartialGridAnalysis que serão chamados.

Dos passos citados acima, apenas o primeiro não é executado com o auxilio de um objeto TBuildStepTask. Tudo ocorre da seguinte forma (Figura 25):

- O processo mestre, cria e submete um objeto TBuildGridTask para cada malha parcial. Esta tarefa executa o primeiro passo, cria a análise parcial e submete tarefas TBuildStepTask para cada um dos outros passos.

Como todos os dados de TBuildGridTask são derivados de TSaveable, as tarefas são criadas no processador 0 e transferidas para o processador onde serão executadas. Por sua vez as tarefas TBuildStepTask criadas no método execute de TBuildGridTask

possuem um ponteiro para a análise, o que impede que as tarefas TBuildStepTask trafeguem de um processador para outro.

A maioria dos passos da análise transfere e recebe dados das demais malhas parciais, através de tarefas construídas exatamente para este fim (Figura 25). São elas:

- **TBuildExternalConnectivityTask**: Leva estruturas TNeighbourInfo com dados sobre os nós de fronteira e chama o método BuildExternalConnectivity() da malha de destino para atualizar as informações de fronteira;

- **TSetBoundaryNodesTask**: Leva um vetor com os ids dos nós de fronteira e chama o método SetBoundaryNodes() da malha geométrica de destino para atualizar as informações de fronteira;

- **TGetIdCorrespondenceBySideTask**: Leva estruturas TCompElSideDefinition com informações de fronteira e chama o método GetIdCorrespondenceBySide() da malha computacional de destino para atualizar as informações de fronteira;

- **TGetIdCorrespondenceByNodeTask**: Leva vetores com os ids dos nós de interface e chama o método GetIdCorrespondenceByNode() da malha computacional de destino. Também cria e submete outro objeto TGetIdCorrespondenceByNodeTask para atualizar as informações na malha origem, caso tenha sido alterado o id de algum nó computacional;

- **TBuildSolutionByIdTask**: Leva uma matriz com valores para serem atualizados num vetor de destino (no caso o vetor diagonal) e um vetor com os ids dos nós a serem atualizados. Chama o método BuildSolutionById() da malha computacional de destino. Esta tarefa seria utilizada na montagem da diagonal D, porém optou-se por usar o Map Manager para este fim, haja visto que a utilização do Map Manager acarreta comunicação apenas entre uma malha e as malhas vizinhas, ao passo que e as tarefas TBuildSolutionByIdTask são enviadas de uma malha para todas as demais;

- **TBuildReceiveVecTask**: Leva um vetor com os ids dos nós de interface e com ele monta o vetor com os índices locais da malha parcial de destino (posição "(origem, destino)" do mapa fReceiveMap do Map Manager).

Cada uma das tarefas citadas acima são criadas em um processador e executadas no processador onde a análise parcial a ser atualizada reside, haja visto que essas tarefas atuam diretamente sobre as malhas computacionais e geométricas da análise de destino. Exceto a tarefa TBuildReceiveVecTask que age sobre os mapas do Map Manager do algoritmo de gradiente conjugado.

7.5.3.1 TBuildGridTask

Por ser a primeira tarefa a ser executada, a tarefa TBuildGridTask é criada no processador 0, na função *main* do programa, e é executada no processador onde sua análise parcial deve residir. É responsabilidade de TBuildGridTask:

- a leitura e criação das malhas parciais e da análise parcial;
- a montagem da correspondência entre os elementos geométricos (primeiro passo envolvido na criação das informações de fronteira);
- a criação e submissão das tarefas TBuildStepTask (que recebem um ponteiro para a análise parcial criada no seu construtor).

Dados:

- long fPAGDId — id da estrutura TPAnalysisGlobalData referente a analise parcial a ser trabalhada;
- long fPAGDIdsId — id do vetor com os ids das estruturas TPAnalysisGlobalData de todas as análises parciais;
- long fMMId — id do Map Manager, que é criado na função *main*;
- long fCGGDId — id da estrutura TCGGlobalData referente a analise parcial a ser trabalhada;
- long fDataIdVecId — id do vetor com os ids das estruturas TCGGlobalData de todas as análises parciais;
- long fCGChkTaskId — id da tarefa de verificação da convergência do algoritmo de gradiente conjugado, que é criada na

função *main*. Esse id é usado para fazer com que o último *build step* seja executado somente após a convergência do algoritmo CG, pois trata-se da geração do arquivo de saída.

- TPartialGridAnalysis *fPAnalysis — ponteiro para a análise parcial criada no método Execute();

Todos os dados são passados no construtor, exceto o ponteiro para a análise parcial.

Método Execute

O método Execute de TBuildGridTask segue os seguintes passos:

1. Criação e construção das malhas parciais geométrica e computacional (respectivamente TPartialGeoGrid e TPartialCompGrid). O id de cada malha é dado segundo o dado fGridId da estrutura TPAnalysisGlobalData da análise parcial em questão.

2. A malha parcial computacional é registrada na lista de malhas computacionais do processador. O mesmo é feito analogamente para a malha parcial geométrica.

3. Os dados do material são lidos.

4. O objeto TPartialGridAnalysis que tratará da análise parcial é criado.

5. A matriz de coeficientes é criada e é chamado então o método BuildGeoElCorrespondence() da análise parcial. Esse método cria a correspondência entre um elemento geométrico e o seu vizinho em uma outra malha parcial.

6. São criados e submetidos cada um dos 8 *build steps* (TBuildStepTask).

7. A tarefa TBuildGridTask termina retornando ESuccess, o que causará sua exclusão da fila de tarefas do **TM**.

7.5.3.2 TBuildStepTask

A classe TBuildStepTask é usada para chamar os métodos da classe TPartialGridAnalysis em momento oportuno. Como cada método envolvido na criação das informações de fronteira precisa ser executado seguindo uma ordem lógica, um método após o outro e sempre depois que todas as malhas tenham enviado suas contribuições. Todos as tarefas TBuildStepTask são criadas e submetidas ao **TM** no método Execute() de TBuildGridTask, porém, cada tarefa TBuildStepTask só é executada quando todas as dependências de cada passo são satisfeitas.

Apenas o ultimo passo é executado duas vezes: uma para determinar de qual versão do vetor solução do gradiente conjugado é necessária para gerar o arquivo de saída, e uma segunda para gerar o arquivo de saída com base no vetor solução já com as contribuições das malhas vizinhas.

Dados:

- unsigned short int fSolDependOk — flag usada no último passo para indicar se a geração do arquivo de saída já pode ser efetuada;
- unsigned int fNumGrids — número de malhas parciais em que o problema foi dividido;
- const int fStep — número do passo;
- TPartialGridAnalysis *fPAnalysis — ponteiro para a análise parcial em questão;
- long fDiagId — id do vetor com os valores da diagonal global referente as equações da análise parcial;
- long fCGGDId — id da estrutura TCGGlobalData referente à análise parcial.

Método Execute:

O método Execute é responsável pela chamada do método do objeto TPartialGridAnalysis, referente ao valor de fStep, ou seja, para cada valor de fStep, é

chamado um ou mais métodos do objeto fPAnalysis (vide item 7.1.3). Logo após é retornado ESuccess e a tarefa sai da fila de execução do **TM**.

O último passo é executado uma primeira vez, logo após a montagem do Map Manager, para determinar o número de atualizações do vetor de solução que deve-se esperar para que o arquivo de saída possa ser gerado. Isto ocorre pois cada análise precisa receber as contribuições com respeito aos nós das malhas vizinhas, e no momento da criação do último passo ainda não se sabe quantas são as malhas vizinhas. Com o número de malhas vizinhas, é registrada a dependência do último passo para o vetor de solução e é retornado EContinue. Dessa forma o último passo só é executado novamente depois que as malhas vizinhas tenham enviado suas contribuições.

7.5.4 Dependência de dados

Cada tarefa da classe TBuildGridTask entra em execução assim que todos os dados dos quais ela depende já tenham sido transferidos para o seu processador.

A tarefa TBuildGridTask cria e submete cada um dos 8 passos do algoritmo paralelo de elementos finitos (tarefas TBuildStepTask). Para cada um dos passos, a dependência de dados é registrada antes da submissão ao **TM**.

Devido ao fato de que cada *build step* agir diretamente sobre a análise parcial, que não é um objeto sob controle do **DM**, seria impossível determinar o momento da execução de cada *build step* usando a versão da análise ou de uma das malhas. Para solucionar essa questão foi utilizado a seguinte técnica: como cada tarefa usava diretamente a estrutura TPAnalysisGlobalData, optou-se por alterar a versão dessa estrutura a cada execução de uma tarefa que viesse a contribuir para a formação das informações de fronteira da malha parcial. A versão da estrutura TPAnalysisGlobalData é incrementada sempre que uma tarefa que age sobre a análise é executada, seja esta tarefa um *build step*, seja uma tarefa que traz informações de outra análise parcial. Deve-se ressaltar que estas tarefas não mudam a estrutura TPAnalysisGlobalData, que possui dados fixos, mas incrementa a versão desta estrutura para que o momento da execução de cada *build step* possa ser determinado, haja visto que a estrutura TPAnalysisGlobalData está sob o controle do **DM**.

O número de atualizações da análise parcial pode ser facilmente determinado graças ao fato de que cada análise parcial envia e recebe de dados de todas as demais, independente da outra malha ser vizinha ou não. Sendo assim, os passos que devem ser executados depois que a análise tenha recebido a contribuição de todas as demais malhas, depende da seguinte versão da estrutura TPAnalysisGlobalData:

```
      dSi = dS(i-1) + NPartAnalysis
      onde:
            dSi = dependência do passo i
            dS(i-1) = dependência do passo i-1
            NPartAnalysis = número total de análises parciais
envolvidas na solução do problema.
```

Capítulo 8

8. Avaliação de desempenho

Para ilustrar o funcionamento do ambiente PZ paralelizado, foram executados testes, com problemas da ordem de 10.000 equações, variando-se o número de processadores e/ou o modelo das máquinas.

Em todos os testes foram utilizados os recursos de hardware e software do ambiente CENAPAD-SP:

Hardware:

- **IBM 9076 SP - Scalable POWERparallel System 2**

  08 processadores IBM RISC 6000 modelo 370

  Arquitetura POWER2

  Ambiente Operacional: AIX V4.1.4

  Velocidade de clock: 66 MHz

  Desempenho teórico de 200 Mflops por nó, totalizando 1,6 Gflop

  Memória de 256 MB por nó, totalizando 2 GB

  Discos por nó: 2 GB, totalizando 16 GB

  High Performance Switch interligando os processadores (até 48,3 MB/s)

- **Estações IBM RISC 6000 39H**

  Arquitetura POWER2

  Ambiente Operacional: AIX V4.1.4

  Velocidade de clock: 67 MHz

Placa Gráfica POWER Gt4e

Memória:

256 MB(máquina: thira)

128 MB(máquinas: delos, paros, psara)

Disco :

4,5 GB (máquina: thira)

9,0 GB (máquinas: delos, paros e psara)

- **Estações IBM RISC 6000 25T**

Arquitetura PowerPC

Ambiente Operacional: AIX V4.1.4

Velocidade do clock: 66,7 MHz

Memória de 32 MB

Disco : 1 GB

Software:

- PVM (Parallel Virtual Machine) versão 3.3.11;
- XPVM (Ambiente gráfico interativo para acompanhamento de aplicações PVM);
- IBM Data Explorer(Software de visualização científica usado no pós processamento).

Sabe-se que são comuns problemas de ordem superior a 10.000 equações, todavia, buscou-se avaliar o ambiente PZ paralelo (desenvolvido neste trabalho), de forma rápida e simples. Problemas de grandes dimensões poderão ser resolvidos com o PZ paralelo em trabalhos futuros.

8.1 O problema de teste

A classe de material usada no programa de teste calcula esforços e deformações em uma placa de 15x15, engastada em todo o contorno, utilizando a teoria de Reissner-Mindlin. Ao todo são seis graus de liberdade por nó, tendo-se utilizado interpolação cúbica para solução do problema. O domínio é sempre dividido em p malhas parciais de dimensões "(n/p) x n" elementos, sendo p o número de processadores e n o número de elementos em cada dimensão do domínio.

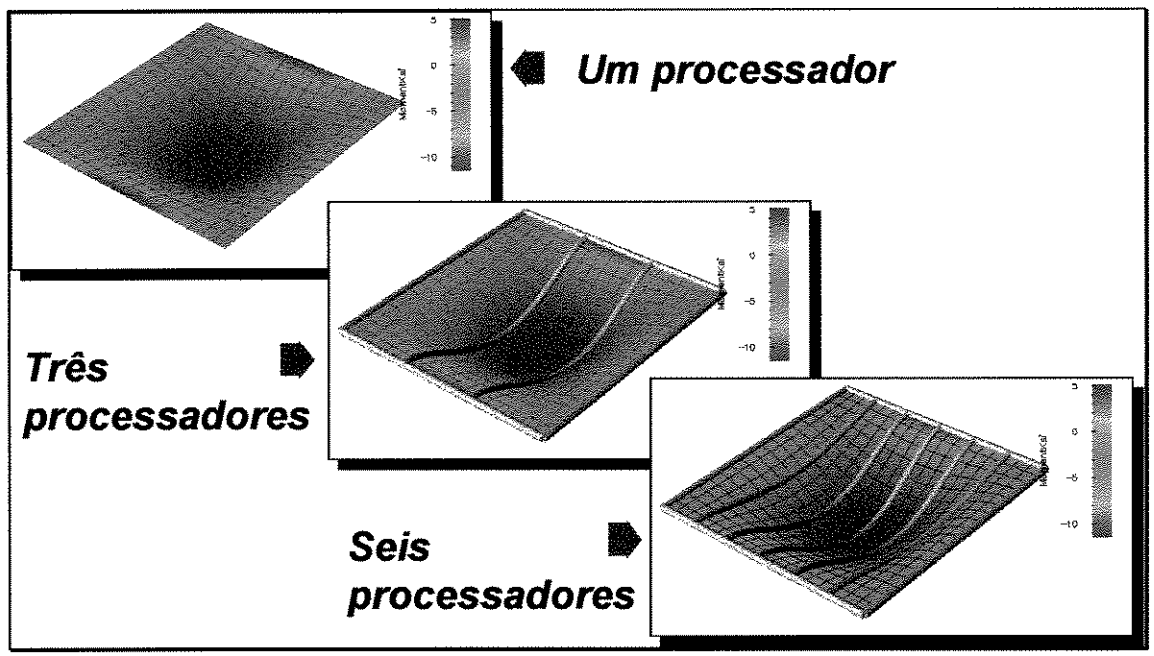


Figura 26 - Visualização DX do momento e da deformação da placa de teste (calculados com 1, 3 e 6 procs.)

8.2 Acompanhamento da execução

Para auxiliar na análise da execução do programas paralelos, associada ao PVM existe uma ferramenta com interface gráfica que permite ao usuário um amplo acompanhamento da execução da aplicação paralela, o XPVM. Com o XPVM, o programador pode avaliar a quantidade de espera de um processador por uma certa mensagem, ajudando na detecção de pontos onde a lógica do programa torne-o ineficiente, pouco paralelizado.

São duas as informações passadas pelo XPVM sobre a execução do problema teste, que serão comentadas:

Gráfico de barras: Cada barra representa a execução de um processo no decorrer do tempo. Quando há comunicação entre os processos, uma linha vermelha é traçada entre o transmissor e o receptor da mensagem. As barras são coloridas segundo o seguinte critério:

- Branco: O processador está esperando por uma mensagem;
- Verde: O processador está computando;
- Amarelo: O processador está ocupado com atividades do sistema operacional.

Entre uma cor e outra o XPVM traça uma linha preta, o que acarreta a aparição de regiões pretas na barra, quando o processador muda de estado rapidamente.

Gráfico de utilização: A utilização da maquina paralela (conjunto dos processadores) é ilustrada através de um gráfico onde aparece, a cada momento t, uma linha vertical dividida em p (número de processadores) partes coloridas de acordo com o tipo de atividade do processador referente. Cada parte é colorida com as seguintes cores:

- Vermelho: O processador está esperando por uma mensagem;
- Verde: O processador está computando;
- Amarelo: O processador está ocupado com atividades do sistema operacional.

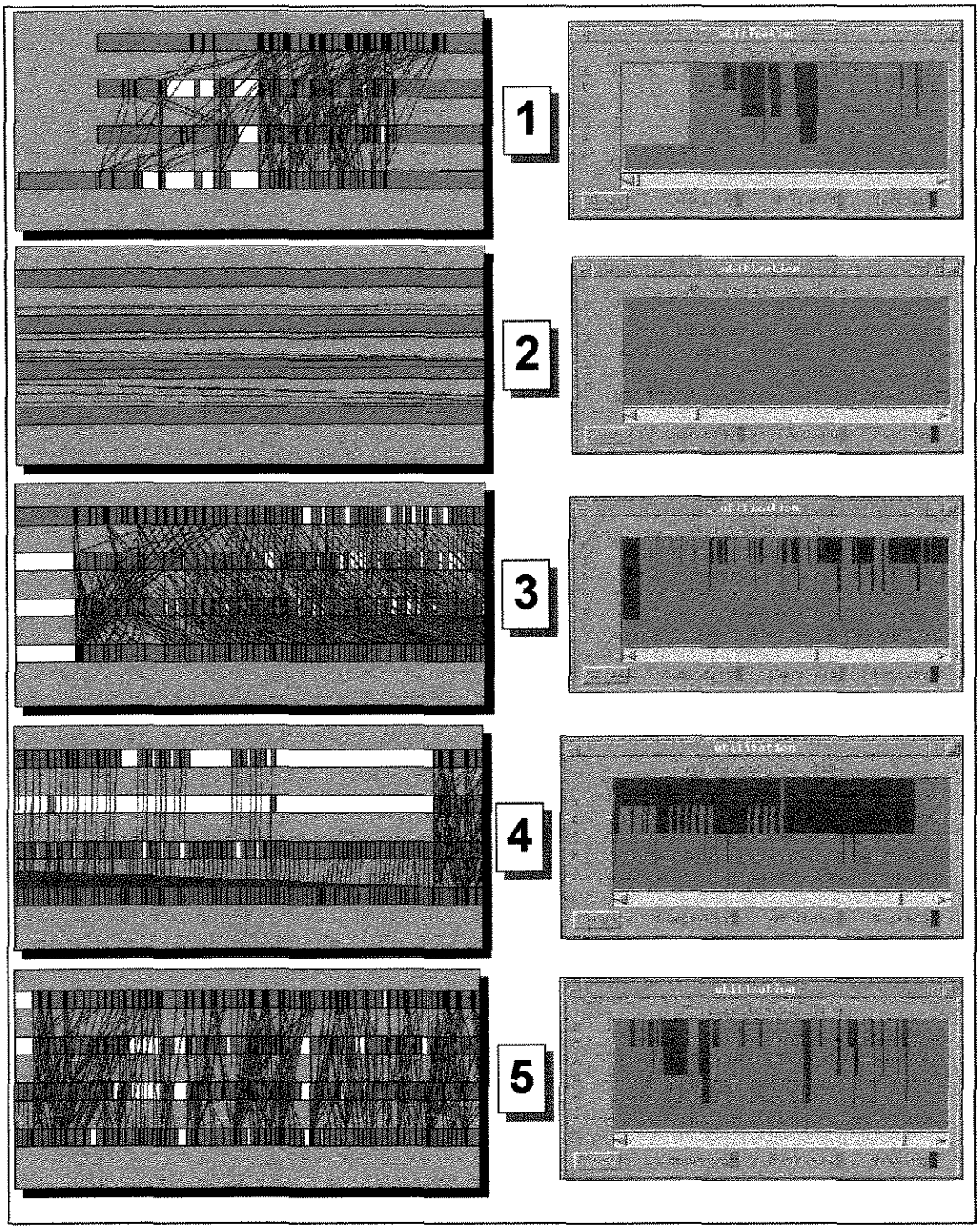


Figura 27 - Análise do programa de teste com o XPVM

O programa de teste foi executado utilizando-se quatro das máquinas do ambiente CENAPAD-SP, 3 estações "rápidas" IBM tipo 39H (Thira, Paros e Psara) e uma estação "lenta" IBM tipo 25T (Leros, primeira barra de cima para baixo). Optou-se por utilizar uma máquina relativamente mais lenta para poder-se avaliar o dispositivo de sincronismo por dependência de versão de dado do OOPAR. A seguir são apresentados os gráficos de barra e de utilização, gerados pelo XPVM quando o mesmo foi utilizado

para análise do programa de teste. Tanto o gráfico de barras como o de utilização são referentes a um mesmo intervalo de tempo. Toda a execução do programa foi dividida em 5 intervalos (Figura 27):

1) **Início - Determinação das informações de fronteira - Assemble:**

Nota-se pelo gráfico de barras que os trabalhos são dividido entre os processadores logo de início. A presença de regiões vermelhas no gráfico de utilização (regiões brancas nas barras) é claramente devido a lentidão da máquina Leros, referente à primeira barra de cima (completamente verde). Os pontos de sincronismo são bem claros no gráfico de barras.

2) **Assemble:**

O Assemble é completamente independente de qualquer comunicação entre os processos, portanto este intervalo é 100% paralelizado.

3) **Assemble - Montagem dos dados do CG:**

A montagem dos dados do algoritmo CG demanda muita comunicação entre os processadores para montagem do Map Manager e para criação dos dados inerentes ao algoritmo. Os dados do CG são submetidos ao **DM** do OOPAR e, para cada dado submetido ao **DM**, é desencadeado um processo de troca de mensagem entre todos os processadores. Está claro pelos gráficos, que neste intervalo há muita comunicação.

4) **Montagem dos dados do CG - Início do CG:**

Os trechos de espera surgem pelo fato de todos os coeficientes do algoritmo CG residirem no processador 0, o que gera uma carga de processamento maior para o este processador, que para os demais.

5) **CG - Final:**

Os pontos de sincronismo do algoritmo CG são claros no gráfico de barras. Nota-se também que as tarefas estão sendo realmente executadas

em paralelo. Os tempos de espera caracterizados nos gráficos são por três motivos:

- O maior número de nós do processador 0;
- O tempo que o processador 0 gasta com as tarefas para acumular os coeficientes do algoritmo CG;
- A "lentidão" do processador 3 (maquina Leros).

O quinto intervalo se repete até o algoritmo CG convergir, sendo que ao final do algoritmo os arquivos para pós processamento são gerados em paralelo (Figura 28).

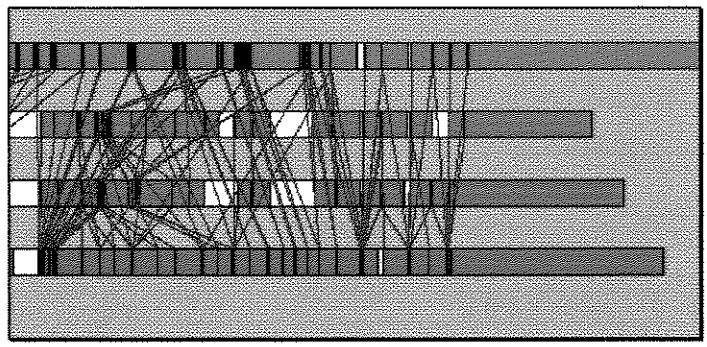


Figura 28 - Geração dos resultados / Finalização do programa.

8.3 Escalabilidade e Eficiência relativa

Eficiência relativa é a razão entre o tempo gasto para a resolução de um problema em um processador e o tempo gasto para a resolução do mesmo problema em p processadores , multiplicado por p[33]:

$$\varepsilon_{rel} = T_1/(p\ T_p)$$

A escalabilidade relativa é obtida comparando-se o tempo de processamento com o número de processadores envolvidos.

Em um gráfico de log(T) contra log(P), a escalabilidade ideal é dada por uma reta decrescente. Nesse mesmo gráfico, a escalabilidade de um programa é dada por uma curva que se afasta da escalabilidade ideal, à medida que p aumenta.

O programa teste foi usado na solução de uma placa de 12x12 elementos, resultando em 8214 equações. Este problema foi executado com uso exclusivo em 1 a 4 estações IBM 25T do ambiente CENAPAD-SP, resultando em uma análise de eficiência e escalabilidade segundo tabela e gráficos a seguir:

P	1	2	3	4
T (min)	43.4	25.4	17.2	14.9
ε_{rel}	1	0.854331	0.841085	0.728188

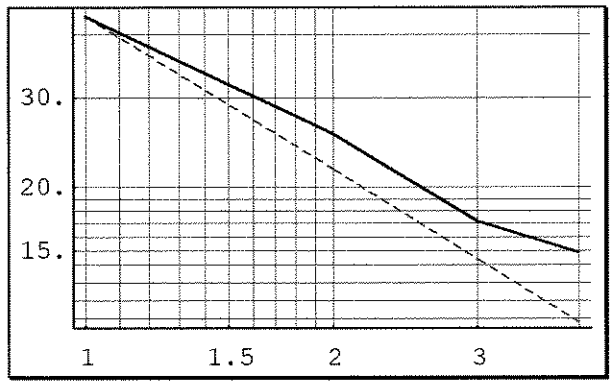


Figura 29 - Análise de escalabilidade com tamanho fixo: Tempo de execução

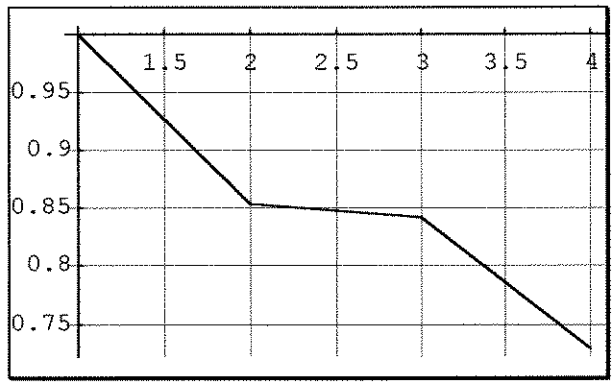


Figura 30 - Análise de escalabilidade com tamanho fixo: Eficiência

Nota-se que a eficiência é praticamente mantida quando aumenta-se o número de processadores de 2 para 3. Conclui-se então que, para que a eficiência seja mantida, o tamanho do problema precisa ser aumentado.

Além dos testes com uso exclusivo das estações 25T do ambiente CENAPAD-SP, foram feitos testes utilizando o SP. Porém, o SP do CENAPAD-SP não pode ser

usado de forma exclusiva, isto é, os processos sempre concorrem por CPU com mais um (ou dois) processos pesados, pois estão associadas a cada nó do SP, duas filas de execução de jobs. Considerando que a utilização do HPS (*High Performance Switch*, o comunicador veloz) implica num uso interativo do SP, os processos dos testes podem ter concorrido com até dois outros processos, haja visto que nenhuma fila de execução *batch* foi interrompida.

No SP, foi utilizado o mesmo problema dos testes na rede de estações 25T, ou seja, uma malha de 12x12 elementos resultando em 8214 equações. Seguem abaixo os resultados da análise de eficiência e escalabilidade no SP:

P	1	3	4
T (min)	70.37	31.53	26.43
ε_{rel}	1	0.743948	0.665626

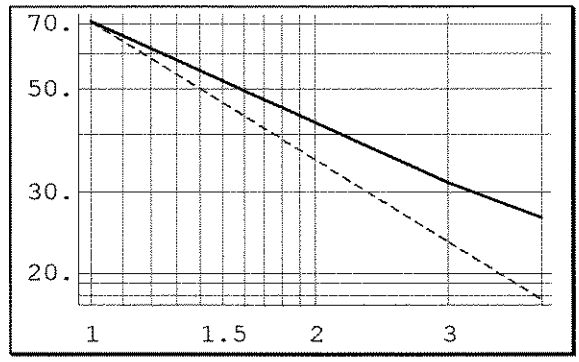


Figura 31 - Análise de escalabilidade com tamanho fixo: Tempo de execução (SP)

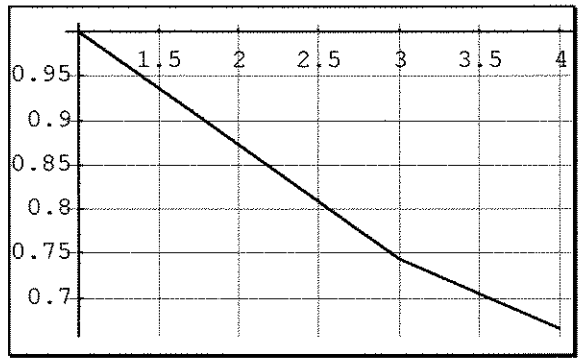


Figura 32 - Análise de escalabilidade com tamanho fixo: Eficiência (SP)

Mesmo sendo inconstante a carga nos nós do SP, observa-se nos gráficos que os testes apresentaram as mesmas características de escalabilidade e eficiência. Também foram executados testes com tamanho superior a 72.000 equações, porém os resultados não puderam ser aproveitados para uma análise de escalabilidade de tamanho fixo, devido ao fato de não haver hardware com memória suficiente para resolver seqüencialmente ($p = 1$) este problema.

Capítulo 9

9. Conclusão

Na primeira parte deste trabalho, utilizou-se o ambiente OOPAR no projeto e desenvolvimento de um algoritmo paralelo de gradiente conjugado pré condicionado (PCG). Graças às vantagens da programação orientada a objeto, uma vez definido o algoritmo paralelo, os trabalhos resumiram-se a criar classes para cada tarefa derivando a classe TTask do OOPAR. Uma vez que todas as tarefas foram desenvolvidas, implementou-se o Map Manager, classe responsável pelo mapeamento das equações locais de um processador para as equações locais dos demais, sendo responsável também pelo gerenciamento das informações com respeito ao relacionamento entre os vários processadores.

Cabem as seguintes observações sobre o ambiente OOPAR:

- Embora de depuração complexa, o mecanismo de dependência de tarefas a versões de dados se mostrou muito eficiente, garantindo a ordem de execução das tarefas implementadas, bem como todos os pontos de sincronismo previstos.

- O ambiente de memória compartilhada emulado pelo Data Manager se mostrou uma ferramenta muito útil e teve importância marcante, propiciando sobretudo a centralização dos coeficientes do PCG paralelo, evitando o cálculo redundante destes por todos os processadores.

- Em ambientes onde os processos precisam concorrer por CPU com os demais processos, o OOPAR pode ser de grande utilidade, pois as tarefas podem ser remanejadas facilmente de forma a diminuir a carga dos processadores mais carregados.

- O Communication Manager que trabalha com arquivos, emulando memória distribuída em PCs mostrou-se uma ferramenta indispensável na fase de implementação e depuração do código.

A flexibilidade do ambiente OOPAR permitiu o desenvolvimento do código de forma que a depuração do programa pudesse ser feita em três passos a saber:

1. Depuração um Data Set em um único processador. Testa-se o algoritmo sem preocupar-se com paralelismo.

2. Depuração com mais de um Data Set em um único processador. Testa-se o algoritmo paralelo sem preocupar-se com sincronismo ou comunicação entre processos.

3. Depuração com mais de um Data Set e mais de um processador. Testa-se o algoritmo paralelo, o sincronismo e a comunicação entre os processos.

O ambiente OOPAR foi depurado, sendo suas classes bem como procedimentos para o desenvolvimento de algoritmos documentados como parte deste trabalho.

A segunda parte deste trabalho teve como meta a paralelização do ambiente PZ de elementos finitos. Com base no ambiente OOPAR desenvolveu-se as classes TPartialGeoGrid, TPartialCompGrid e TPartialGridAnalysis, que estendem respectivamente a malha geométrica, a malha computacional e a classe de análise do ambiente PZ, para trabalhar com processamento paralelo. Como os métodos da classe TPartialGridAnalysis devem ser chamados de forma sincronizada pelos diversos processadores, foram criadas tarefas derivadas de TTask que utilizam os mecanismos do OOPAR para garantir que cada método seja chamado em momento oportuno.

Este trabalho visou sobretudo, o desenvolvimento de um ambiente onde os problemas de elementos finitos fossem resolvidos com processamento paralelo desde a criação das malhas até a geração dos arquivos de pós processamento. Esta meta foi alcançada. O ambiente PZ pode ser utilizado na solução, em ambientes de memória distribuída, de problemas de dimensões tais que sequer a malha caberia na memória de máquinas convencionais.

Em trabalhos futuros, o ambiente PZ pode ser alterado para que todas as suas classes possam ser transmitidas de um processador para outro, proporcionando uma maior flexibilidade pois poder-se-ia transferir uma malha para um processador mais conveniente, caso fosse necessário.

O ambiente OOPAR também pode ser alterado para que as tarefas e dados do ambiente possam ser gravadas em disco, e lidas posteriormente, para que dessa forma seja possível parar e reiniciar em outro momento, um processo que demande muito tempo de CPU.

A seqüência natural para este trabalho é a aplicação do pré condicionador definido em [13] na solução do algoritmo PCG paralelo.

Finalizando, uma grande característica deste trabalho é que ambas as partes do mesmo podem ser independentes, ou seja, o algoritmo PCG paralelo desenvolvido pode ser aplicado em qualquer problema que resulte em um sistema linear simétrico positivo definido, por sua vez, o ambiente PZ paralelo pode ser alterado para acionar um outro método iterativo paralelo para resolver o sistema de equações resultante.

10. Referências

[1] ALMASI, G.S., GOTTLIEB, A. **Parallel computing**. Redwood City: The Benjamin/Cummings Publishing Company, 1989. 489 p.

[2] BARRET, R. et al. **Templates for the Solution of linear systems: building blocks for iterative methods**. Philadelphia: SIAM, 1994. 105 p.

[3] LEA, D. et al. **User's guide to the GNU C++ Library**. Cambridge: Free Software Foundation, 1992. 124 p.

[4] AXFORD, T. **Concurrent programming**. West Sussex: John Wiley & Sons, 1989. Cap. 11: Parallel processing of Sets of data. p. 157-175

[5] DEVLOO, P.R.B., MENEZES, F.A., DA SILVA, E.C. OOPAR: The development of an environment for parallel computing using the object oriented programming philosophy. **Advances in Computational Structures Technology**. Edinburgh, p. 151-156, 1996.

[6] CAREY, G.F. **Parallel supercomputing: methods, algorithms and applications**. West Sussex: John Wiley & Sons, 1989. 287 p.

[7] BERTSEKAS, D.P., TSITSIKLIS, J.N. **Parallel and distributed computation: numerical methods**. Englewood Cliffs: Prentice Hall, 1989. 713 p.

[8] DONGARRA, J.J., POZO, R., WALKER, D.W. **LAPACK++: A design overview of object oriented extensions for high performance linear algebra**.

[9] DONGARRA, J.J., POZO, R., WLAKER, D.W. An object oriented design of high performance linear algebra on distributed memory architectures. In: OONSKI'93 - Annual Object-Oriented Numerics Conference, 1st, 1993, SunRiver. **Proceedings...** Oregon: SIAM.

[10] DEVLOO, P.R.B. On the development of a finite element program based on the object oriented programming philosophy. In: OONSKI'93 - Annual Object-Oriented Numerics Conference, 1st, 1993, SunRiver. **Proceedings...** Oregon: SIAM.

[11] SKACEL, M. **PESIM 1.2 - manual**. Brno: Technical University of Brno. 1994. (skacel@dcse.fee.vutbr.cz)

[12] WIENER, R.S., PINSON, L.J. **An introduction to object oriented programming and C++**. Addison Wesley, 1988.

[13] SANTANA, M. L. M. **Desenvolvimento de algoritmos de subestruturação para elementos finitos**. Campinas: FEC, UNICAMP, 1997. Dissertação (mestrado) - Faculdade de Engenharia Civil, Universidade Estadual de Campinas, 1997. 171 p.

[14] LIPPMAN, S.B. **C++ primer**. New York: Addison-Wesley Publishing Company, 1991. 614 p.

[15] SIMULOG. **MUDULEF users guide nº 1**. 180 p. 1995. (modulef@simulog.fr).

[16] ANSYS, Inc. **ANSYS 5.3 Complete User's Manual Set**. 1996.

[17] IBM CORPORATION. **IBM Visualization Data Explorer User's Guide**. Thomas J. Watson Research Center, 1994.

[18] CENPES-PETROBRÁS/PUC-RIO. **MVIEW Bidimensional Mesh View - Manual do Usuário**. Grupo de tecnologia em computação gráfica - TecGraf/PUC-Rio e Grupo de Geotecnia do SEDEN/DIPREX., 1995. 36p.

[19] RIBEIRO, F. L. B .**Introdução à computação gráfica**. COPPE/UFRJ, 1996. 136 p.

[20] PRESS, W.H. et al. **Numerical Recipes**. Cambridge: Cambridge University Press, 1986. 818 p.

[21] JENNINGS, A. **Matrix computation for engineers and scientists**. Surrey: John Wiley & Sons, 1977. 330 p.

[22] GOLUB, G. H., VAN LOAN, C. F. **Matrix computations**. London: The Johns Hopkins University Press, 1989. 640 p.

[23] BECKER, E. B., CAREY, G. F., ODEN, J. T. **Finite Elements; An Introduction**. Englewood Cliffs: Prentince-Hall, 1981. 258 p.

[24] COOK, R.D., MALKUS, D.S., PLESHA, M.E. **Concepts and applications of finite element analysis**. New York: John Wiley & Sons, 1989. 630 p.

[25] ULBIN, M. et al. Object oriented programming of engineering numerical applications. **Advances in Computational Structures Technology**. Edinburgh, p. 137-142, 1996.

[26] VALKENBERG, G. W. et al. Parallel iterative solvers for finite element analysis. **Advances in Computational Structures Technology**. Edinburgh, p. 373-382, 1996.

[27] KINCAID, D.R., OPPE, T.C. Some parallel algorithms on the four processor Cray X-MP4 supercomputer. In: CAREY, G.F. **Parallel supercomputing: methods, algorithms and applications**. West Sussex: John Wiley & Sons, 1989. p. 124-128.

[28] MACIEL, P.R.M., LINS, R.D., CUNHA, P.R.F. **Introdução às redes de petri e aplicações**. Campinas: Instituto de computação - UNICAMP, 1996. 187 p.

[29] IBM CORPORATION. **Introduction to parallel processing and Scalable POWERparallel Systems 9076 SP1 and 9076 SP2**. IBM International Technical Support Center. 1994. 107 p.

[30] GROPP, W., LUSK, E., SKJELLUM, A. **Using MPI : portable parallel programming with the message-passing interface**. Cambridge: Massachusetts Institute of Technology, 1996. 307 p.

[31] HWANG, K., BRIGGS, F.A. **Computer arquitetures and parallel processing**. San Francisco: McGraw-Hill Book Company, 1984. 846 p.

[32] GEIST, A. et al. **PVM parallel virtual machine : a users' guide and tutorial for networked parallel computing**. Cambridge: Massachusetts Institute of Technology, 1994. 279 p.

[33] TOLEDO, E.M., SILVA, R.S. **Introdução à computação paralela**. Rio de Janeiro: Laboratório Nacional de Computação Científica, 1997. 226 p.

[34] SNIR, M. et al. **MPI : the complete reference**. Cambridge: Massachusetts Institute of Technology, 1996. 336 p.

[35] HSIEH, S., SOTELINO, E. D. **PPI++: An object-oriented parallel portability interface in C++**. West Lafayette: Purdue University.

[36] BECKMAN, P., GANNON, D., SUNDARESAN, N. pC++ meets multi-threaded computation. In: WORKSHOP ON ENVIRONMENTS AND TOOLS FOR PARALLEL SCIENTIFIC COMPUTING, 2nd, 1994, Townsend. **Proceedings...** Philadelphia: SIAM, 1994. p. 76-85.

[37] KESSELMAN, C. Implementing parallel programming paradigms in CC++. In: WORKSHOP ON ENVIRONMENTS AND TOOLS FOR PARALLEL SCIENTIFIC COMPUTING, 2nd, 1994, Townsend. **Proceedings...** Philadelphia: SIAM, 1994. p. 86-97.

[38] KEYES, D. E., SAAD, Y., TRUHLAR, D. G. (Ed.) **Domain-based parallelism and problem decomposition methods in computational science and engineering**, Philadelphia: SIAM, 1995. 323 p.

[39] KARYPIS, G., KUMAR, V. **Metis - Unstructured graph partitioning and sparse matrix ordering**. Minneapolis: University of Minnesota, 1995. 16 p.

[40] DONGARRA, J. J., TOURANCHEAU, B. (Ed.) **Environment and tools for parallel scientific computing**. Philadelphia: SIAM, 1994. 291 p.