UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

Implementação Paralela do Método de Resolução Frontal de Sistemas de Equações

Autor: **Gustavo Camargo Longhin**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

Implementação Paralela do Método de Resolução Frontal de Sistemas de Equações

Autor: **Gustavo Camargo LonghinCedric Marcelo Augusto Ayala Bravo**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

Curso: Engenharia Civil
Área de concentração: Estruturas

Dissertação de Mestrado apresentada à comissão de Pós Graduação da Faculade de Engenharia Civil, como requisito para obtensão do título Mestre em Engenheria Civil.

Campinas, Setembro de 2001
S. P. - Brasil

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

L861i

Longhin, Gustavo Camargo.
Implementação paralela do método frontal de resolução de sistemas de equações / Gustavo Camargo Longhin.--Campinas, SP: [s.n.], 2001.

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

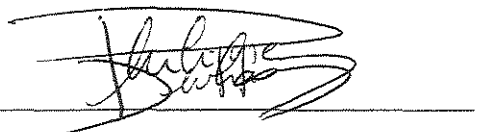
1. Programação orientada a objetos (Computação). 2. Programação paralela (Computação). 3. Método dos elementos finitos. 4. Álgebra. I. Devloo, Philippe Remy Bernard. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil. III. Título.

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

TESE DE MESTRADO

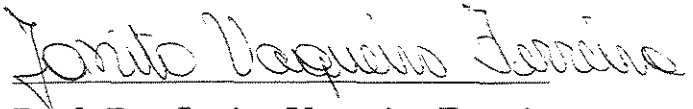
Implementação Paralela do Método Frontal de Resolução de Sistemas de Equações

Autor: **Gustavo Camargo Longhin**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**



Prof. Dr. Philippe Remy Bernard Devloo
instituição DES/FEC/UNICAMP

Prof. Dr. Mario Conrado Cavichia
instituição DES/FEC/UNICAMP



Prof. Dr. Janito Vaqueiro Ferreira
instituição DMC/FEM/UNICAMP

Campinas, 13 de Setembro de 2001

Dedico este trabalho à minha querida e amada esposa Adriana pelo amor incondicional, pela paciência, pelo apoio, pelo suporte e principalmente pela nossa filha Laura, que Deus nos abençoe.

Agradecimentos

Ao Departamento de Estruturas da Faculdade de Engenharia Civil - UNICAMP pela infraestrutura.

À FINEP, CNPQ e PETROBRAS pelo financiamento da infraestrura computacional.

À FAPESP pela bolsa de estudos e reserva técnica.

Ao amigo e orientador Prof . Dr. Philippe R. B. Devloo pela empolgante orientação e compartilhamento irrestrito de suas idéias.

À minha amada esposa Adriana, pela nossa filha Laura, pelo suporte, paciência, pelo amor e apoio incondicional.

Ao meu pai Adolfo e minha mãe Maria Lúcia pelo amor, carinho, educação e pela família sempre presente.

Aos amigos e professores do departamento de estruturas pelo suporte e interesse no desenvolvimento do trabalho.

Aos amigos do LabMeC em especial, ao Edivaldo pela Rapadura, Cesar pelas revisões, correções e pessimismo, Fábio por pilotar a churrasqueira, Erick pela diferença, Cedric e aos demais aqui não citados, que compartilharam no desenvolvimento do trabalho.

À Deus pela minha vida, e por iluminar meu caminho.

Resumo

Para obtenção da solução do sistema de equações, geralmente elabora-se um código que armazena a matriz dos coeficientes na memória RAM. Em seguida inicia-se o processo de decomposição desta matriz.

A matriz é formada pela contribuição de cada elemento do domínio aos graus de liberdade do problema. Quanto maior o problema, maior o tamanho desta matriz e consequentemente mais memória RAM será necessária para seu armazenamento. Portanto, um procedimento de montagem anterior à decomposição pode ser inviabilizado devido ao tamanho da matriz.

Com esta motivação, Bruce Irons desenvolve no início da década de 70, um método de resolução que não exige uma montagem inicial da matriz de rigidez global de equações. Neste método é definida uma estrutura onde uma equação totalmente adicionada é imediatamente decomposta e armazenada num dispositivo independente. A matriz que recebe as equações totalmente adicionadas é denominada matriz frontal e com isso o método também é denominado método frontal.

Sobre a estrutura frontal são aplicadas técnicas de optimização por paralelismo. São utilizados equipamentos com memória compartilhada e portanto, utilizão-se as bibliotecas oriundas da especificação posix (pthread no ambiente GNU & Linux) para desenvolvimento multi-threading. São apresentados resultados comparando o método frontal com outros métodos bem como as comparações entre os métodos seriais e os paralelos.

Aplica-se um tratamento orientado a objetos para desenvolvimento dos solvers. Nota-se com a orientação a objetos um excelente grau de modularidade, documentação, extendibilidade e manutenção no código elaborado. A utilização de UML (Unified Modeling Language) é também de grande valia no desenvolimento/planejamento do projeto.

Abstract

Obtaining the solution of a system of linear equations, generally results in a code elaboration which stores the matrix coeficients in the RAM memory and afterwards, some decomposition processes starts.

The matrix is assembled summing up the contributions of each element from the domain to the problem's degrees of freedom. The bigger the problem, the larger the assembled matrix, therefore a higher requirement regarding the RAM memory capacity. From this, a procedure which does not assemble the matrix of coeficients prior to its decomposition would be more interesting.

With that motivation, Bruce Irons developed in the beggining of the seventies a procedure which does not require an initial assembly of the global stiffness matrix. In this method a structure is defined where a totally added equation is immediatelly decomposed and the decomposition results are stored in an independent storage device. The matrix which receives the equations contribution was called frontal matrix and so was the method.

On that frontal structure parallel optimization techniques are applied. Shared memory equipments are the hardware basis for the implementation and accordingly, public domain multi-threading libraries based on the posix specification are used (pthread under GNU & Linux) for the multi-threading development. Results are shown comparing standart methods against the frontal solver as well as serial codes against parallel ones.

Object oriented techniques are applied for the solvers development and planning. As a result, excelent degrees of modularity, extendibility, documentation and management are observed. The Unified Modelling Language (UML) utilization as a helpping tool for object oriented development was also very important.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Introdução

Problemas físicos podem ser representados por modelos matemáticos, no entanto, nem todos os problemas possuem uma solução analítica para sua formulação matemática. Na maioria dos casos, aplicam-se as aproximações numéricas para obtenção de uma solução aproximada ao modelo. Uma destas técnicas de aproximação numérica é o Método dos Elementos Finitos (MEF).

O objetivo da aplicação de aproximação numérica num projeto é simular algum comportamento real, diminuindo, desta forma, as inconsistências entre produto idealizado e realidade do produto final. Equipes técnicas responsáveis por projetos dos mais variados tipos recorrem a simulações numéricas para garantirem a viabilização de seus projetos.

O MEF requer para sua aplicação, a realização de um volume alto de operações matemáticas. A popularização dos microcomputadores é o motivo pelo qual o MEF pôde ser difundido e consagrado em escala mundial.

Uma das simulações realizadas com elementos finitos requer a obtenção da solução de um sistema de equações do tipo $Ax = B$. Quanto maior o problema a ser resolvido, maior o número de incógnitas e equações envolvidas. Quanto maior o número de incógnitas e equações envolvidas, maior a precisão no resultado e maior a quantidade de operações matemáticas a ser executada pelo computador. A quantidade de operações se traduz em tempo de execução. Busca-se então um equilíbrio entre a precisão e a viabilidade computacional do modelo.

Durante um processo de solução por elementos finitos, o custo computacional para obtenção da solução do sistema de equações perfaz algo em torno de 80% do custo total do processo [6]. Por isso é fundamental utilizar a técnica mais eficaz para solução de um sistema de equações. Isto justifica a dedicação da comunidade científica na pesquisa de solvers especializados. Estes solvers são otimizados com respeito a morfologia do sistema de equações, ambiente operacional e arquitetura de processadores.

A utilização ineficiente de recursos computacionais ainda é comum. Máquinas com processamento paralelo são frequentemente sub-utilizadas, executando aplicativos escrito para arquiteturas não paralelas. Disponibilizar solvers especializados em arquiteturas paralelas seria, portanto, muito interessante.

Será desenvolvido neste projeto um solver Frontal, inicialmente idealizado por Bruce Irons [26] na década de 70, adequado para arquiteturas paralelas.

O método frontal teve muita aceitação na comunidade científica. Vários autores desenvolveram variações sobre a mesma estrutura de solução que podem ser verificadas nas citações [55], [6], [54], [19], [50], [15].

Desenvolvendo um solver frontal adequado à arquitetura da máquina pode-se estender os limites do equilíbrio computacional aumentando a performance do código.

Utilizando a filosofia de programação multi-linguagem o solver frontal é encapsulado num objeto *ActiceX* e estará assim disponível aos usuários *Visual BASIC*.

A utilização da filosofia *MLP* (*Mixed Language Programming*) será também assunto deste projeto. O solver implementado em *C++* será disponibilizado para programadores de linguagens tipo *RAD* (*Rapid Development Application*), entre elas o *Visual BASIC*.[24]

1.2 Objetivos

Pode-se dividir os objetivos deste projeto em duas partes, sendo uma focada em performance de execução, e uma segunda que trata de desenvolvimento. Sendo esta última não apenas centrada no tópico de codificação, mas também na disponibilização de tecnologias através de novas filosofias de programação multi-linguagem.

1.2.1 Performance Computacional

Busca-se nesta etapa obter melhoras na performance durante a solução de sistemas de equações de grande porte, utilizando os recursos oferecidos por máquinas paralelas com memória compartilhada *SMP* (*Shared Memory Parallel*)

Máquinas do tipo PC com múltiplos processadores estão cada vez mais accessíveis. Porém, apesar do desenvolvimento dos hardwares, poucos softwares são capazes de aproveitar a arquitetura paralela. Sendo assim, frequentemente máquinas com mais de um processador são sub utilizadas.

A performance atingida atualmente pela maioria dos softwares poderia ser melhorada fossem estes reescritos apropriadamente para operarem em uma arquitetura paralela. Um código serial (i. e. não paralelo) executando em uma máquina com dois processadores utiliza apenas 50% dos recursos da máquina.

Um dos objetivos deste trabalho é desenvolver e implementar códigos especializados, que se adaptam a arquitetura do hardware, aumentando a eficiência do sistema computacional. Espera-se que processos antes executados com 50% ou menos do potencial dos equipamentos, sejam agora executados com todo potencial disponível.

1.2.2 Desenvolvimento Multi-Linguagem

Observa-se no desenvolvimento das linguagens de programação uma crescente especialização. Diferentes linguagens tornam-se mais apropriadas para desenvolvimento de certos tipos de sistemas [22].

Propõe-se utilizar as vantagens de *C++* para criar um objeto solver frontal que poderá ser utilizado pelos usuários de *Visual BASIC* .

Performance Algébrica

As operações algébricas existentes no processo frontal envolvem operações entre matrizes e/ou vetores esparsos, que são estruturas de dados complexas.

A linguagem *C++* permite trabalhar com facilidade e eficiência com estruturas de dados complexas [33] sendo, consequentemente, uma das mais recomendadas para operações algébricas. *C++* será utilizada para a implementação dos códigos dos solvers [53].

Performance de Desenvolvimento

A linguagem *VB* (*Visual BASIC*) mostra-se imbatível no tempo gasto para desenvolvimento, sendo operada num ambiente de desenvolvimento amigável e com ótimos recursos de edição.

Com esta linguagem, um programador experiente pode desenvolver executáveis completos, com poderosas interfaces gráficas em questão de dias [34], [2]. Ressalva feita com relação à performance do aplicativo, caso este envolvesse grandes quantidades de operações com vetores e/ou matrizes.

Otimização do Desenvolvimento

Pretende-se utilizar as vantagens de duas linguagens para desenvolvimento do projeto. A linguagem *C++* com mais requintes estruturais será utilizada para todos os métodos e estruturas de dados necessárias para realização de operações algébricas. A linguagem *VB* com facilidades de desenvolvimento, codificação e edição avançadas, será utilizada para implementação das interfaces gráficas, bem como as entradas de dados e estruturação global do projeto.

A parte complexa é implementada utilizando a linguagem *C++*. Todos recursos algébricos/matemátic são inseridos numa espécie de caixa preta. O solver resultante será disponibilizado à programadores de *VB* através da tecnologia desenvolvida pela *Microsoft* chamada *COM* (*Component Object Model*) [34], [12].

Além do *VB*, outras linguagens que possuem interface para *COM* são *FoxPro*, *JAVA*, *Delphi* etc.

O aplicativo que utiliza o objeto desenvolvido para solução do sistema de equações será eficiente, e possuirá um curto período de implementação [22].

Capítulo 2

Conhecimentos Utilizados

2.1 Filosofia de Programação Orientada para Objetos

2.1.1 Linguagem C++

O *C++* é uma linguagem de programação baseada na filosofia de orientação para objetos. Os conceitos básicos deste enfoque foram já introduzidos pela linguagem de programação *Simula* desenvolvida na década de 60 por O. J. Dahl e Kristen Nygaard [14]. A linguagem *Simula* foi desenvolvido para realizar simulações computacionais de processos reais, e nesta, a elaboração de módulos é central. A construção destes módulos baseia-se nos objetos físicos a serem modelados [5][3], [43].

A linguagem *Simula* não obteve sucesso no desenvolvimento de programas de propósitos gerais, mas os conceitos utilizados nesta foram explorados por várias linguagens posteriores, entre as quais se destacam o *SmallTalk* e o *C++*. O *SmallTalk* foi desenvolvido pela *Xerox PARC* (Palo Alto Research Center, California) na década de 70, por uma equipe coordenada por Alan Kay conforme [21], [31], [29].

O desenvolvimento do *C++* iniciou-se em 1980 nos laboratórios da *Bell* por Bjarne Stroustrup [45]. A linguagem foi originalmente desenvolvida para solucionar algumas simulações motivadas por eventos, com restrições muito rigorosas, para as quais as considerações sobre eficiência impediam o uso de outras linguagens. O *C++* foi utilizado primeiramente por uma equipe não pertencente ao grupo de desenvolvimento coordenado por Stroustrup em 1983 e, no verão de 1987, a linguagem estava ainda em pleno desenvolvimento, passando por refinamentos e evoluções naturais.

Um objetivo chave do projeto *C++* era manter a compatibilidade com *C*. A idéia era preservar a integridade de milhões de linhas de código já escritas e depuradas, a integridade de muitas bibliotecas *C* existentes e a utilidade de ferramentas *C* já desenvolvidas. Devido ao alto grau de sucesso na obtenção deste objetivo, a transição de *C* para o *C++* mostrou-se muito simples. A melhoria mais significativa na linguagem *C++* é seu suporte à filosofia de programação orientada para objetos (OOP).

As linguagens *SmallTalk* e *C++* são as mais utilizadas atualmente, embora estes possuam diferentes enfoques para realizar a orientação para objetos. O *SmallTalk* é considerado como uma

linguagem pura, enquanto que o *C++* é híbrida, no sentido que em *C++* coexistem características de uma linguagem estruturada (o *C*) juntamente com características de orientação para objetos.

O fato de ser uma linguagem muito difundida faz de *C++* uma das linguagens que mais se desenvolve. A sintaxe de *C++* é muito rica, dando ao programador várias opções para escrever a mesma instrução. Comandos muitas vezes escritos com muitas linhas, podem ser reescritos com poucas linhas, o que se traduz não só em redução no tempo de implementação, mas também execução, pois o compilador criará um código mais eficiente[23], [33].

2.1.2 Características Básicas

A filosofia de orientação para objetos teve sua maior penetração na indústria de software. Porém, seu uso no âmbito de computação científica ainda é incipiente.

O que diferencia a orientação para objetos das demais filosofias de projeto de sistemas, é a abordagem ao problema. Abordando o problema sob um ponto de vista estruturado (i.e. não orientado para objetos) procura-se definir um conjunto de dados que definem o estado do sistema e uma posterior sequência de eventos que acarretará na transformação do estado do sistema.

Numa abordagem orientada para objetos definem-se estruturas abstratas, denominadas classes, responsáveis por partes da solução do problema. Cada classe incorpora tanto dados (forma) como métodos (comportamentos) sendo estes necessários e suficientes para tratar as responsabilidades da classe [4].

A classe sendo uma abstração, a simples definição da mesma não é suficiente para promover a solução do problema. Faz-se necessário criar uma instância desta classe, chamado objeto. Este é uma entidade auto-suficiente, incorpora forma e comportamento conforme sua classe.

A correta interação entre vários objetos, de classes responsáveis por diferentes partes do problema global, resultará na solução do problema [10], [4], [40], [28], [42].

Classes e Encapsulamento

Segundo Furlan [18], a definição de uma classe é "uma coleção de objetos que podem ser descritos com os mesmos atributos e as mesmas operações." Determina-se os objetos que compartilham as mesmas necessidades e comportamentos definindo a responsabilidade da classe.

Esta precisa definição de escopos e responsabilidades torna o planejamento e gerenciamento da elaboração do sistema simples e de fácil entendimento. Visto que cada classe é responsável por uma porção da solução e uma classe incorpora abstratamente forma e comportamento, não é necessário que outras classes tenham acesso ao esquema de funcionamento desta classe. Não importa como é realizado tal tarefa, o essencial é que tal tarefa é simplesmente realizada. Define-se assim encapsulamento.

Objetos de uma classe são para objetos de uma outra classe uma caixa preta, reforçando a abordagem que não interessa como tal tarefa seja realizada e apenas que ela é realizada.

Herança

Na abstração de classes, há a possibilidade de ocorrer uma conexão semântica entre mãe e filho na qual uma classe filha (sub-classe) herda as propriedades de sua mãe (super-classe) direta ou indiretamente. Cada classe pode ter suas propriedades particulares herdadas diretamente da classe mãe ou substituídas/mascaradas nesta transição, assim somente propriedades diferentes serão declaradas na classe filha. Quando é criada uma nova classe a partir de uma classe existente, a nova classe automaticamente herda as características da classe base.

A abstração de uma classe base estabelece uma interface comum para um grupo de classes semelhantes, a herança é a capacidade de um novo objeto tomar atributos e operações de um objeto existente, permitindo criar classes complexas sem repetir código.

Polimorfismo

Duas situações caracterizam o polimorfismo, estas são diferenciadas pelo escopo em que ocorrem. De acordo com Page-Jones em [36] dois casos são definidos sucintamente como segue:

1. Polimorfismo é a habilidade pela qual uma única operação ou nome de atributo pode ser definido em mais de uma classe e assumir implementações diferentes em cada uma destas classes.

2. Polimorfismo é a propriedade por meio da qual um atributo ou variável pode apontar para (ou manter o identificador de) objetos de diferentes classes em instantes diferentes.

O polimorfismo baseado na definição 1 é o mais intuitivo, onde um mesmo método pode ser declarado mais de uma vez, sendo que, a diferença em cada uma das implementações são os parâmetros fornecidos. Por exemplo:

- void TMatrix::Multiply(TMatrix & m);
  Um método onde o próprio objeto da classe é multiplicado pelo parâmetro "m" e o resultado da multiplicação armazenado no próprio objeto.

- void TMatrix::Multiply(TVector & v);
  Um método onde o próprio objeto da classe é multiplicado pelo parâmetro "v" e o resultado armazenado em "v".

- TMatrix * TMatrix::Multiply(TMatrix & m);
  Um método onde o próprio objeto da classe é multiplicado pelo parâmetro "m" e o resultado retornado para um ponteiro de TMatrix.

O compilador decide qual das implementações utilizar baseado no contexto da chamada para a função.

O polimorfismo baseado na definição 2 não apresenta-se de forma tão trivial, mas é sem dúvida a mais completa definição de polimorfismo. Tal característica está diretamente ligado como o conceito de abstração em orientação para objetos. Métodos abstratos numa classe não possuem implementações nesta classe, tornando-a uma classe abstrata. Uma implementação para estes

métodos deve ser fornecida por uma classe filha não abstrata. A redefinição de um método numa classe filha como abstrato invalida este método na classe filha.

Como resultado prático para estas afirmações tem-se:

Suponha uma classe que implementa um método que recebe como parâmetro um ponteiro para esta classe. Neste caso, quando da chamada para este método, poderá ser fornecido como parâmetro um ponteiro para qualquer objeto de uma classe filha desta classe. Qualquer operação sobre o ponteiro no escopo interno ao método, será executada baseada nas implementações da classe filha. Cabe ao compilador identificar os tipos e executar as corretas implementações.

Nota-se com a utilização de polimorfismo em C++ dois pontos favoráveis para desenvolvimento em larga escala. Um primeiro ponto diz respeito a imposição de interfaces em classes filhas, ou seja, a definição de atributos abstratos (*virtual*) numa classe, força que as classes filhas necessariamente implementem estes atributos abstratos. Promove-se com isso um código mais robusto e de mais fácil manutenção.

Um segundo ponto está relacionado com o escopo de implementação, ou seja, uma operação abstrata permite que um método seja chamado sem que se saiba precisamente qual a classe do objeto alvo, sabendo-se apenas que o a classe é filha de uma classe abstrata e portanto implementa o método sendo chamado.

As referências [45, 46, 33] trazem detalhadas explicações sobre as propriedades relacionadas à abstração em orientação para objetos através de *C++*.

Os conceitos de polimorfismo são largamentes utilizados nos módulos do ambiente PZ, destacando-se o módulo matricial onde, a classe base TPZMatrix é abstrata, forçando em qualquer classe filha a implementação de sua interface básica. Em `http:\\labmec.fec.unicamp.br\~pz` obtem-se a documentação online do pacote.

2.2 UML

A linguagem *UML* (*Unified Modelling Language*) tem como objetivo deixar todas as informações importantes ao projeto explícitas, e em seguida, transmitir de maneira clara e precisa estas informações às equipes encarregadas da sua elaboração.

Para desenvolvimento de software, a linguagem UML é um formalismo para descrever um projeto orientado para objetos. É estruturada como um conjunto de diagramas, contendo cada um o seu propósito específico.

UML é também fruto de um esforço de unificação de diversos diagramas desenvolvidos por diferentes autores. Em particular, as seguintes metodologias existiam antes da UML. Como é descrito por Furlan [18].

1. Booch - Desenvolvido por Grady Booch, sua proposta era um método baseado em desenho orientado à objeto.

2. OMT - Idealizado por Rumbaugh, desenvolvido na *GE Corporation*, também conhecido como técnica de modelagem de objetos (*OMT - Object Modelling Technique*). Baseado na modelagem semântica de dados, este método tornou-se um enfoque testado e maduro, cobrindo as diversas fases do desenvolvimento orientado a objeto.

3. OOSE - Jacobson, com sua técnica, criou as bases para os métodos *Object Oriented Software Engineering (OOSE)* e *Objectory*. Este método é diferenciado de outros pelo seu foco em casos de uso e a categorização de pessoas e equipamentos dependendo do seu papel no sistema global.

4. Shlaer/Mellor - Sally Shlaer e Stephen Mellor criaram um método orientado à objeto que pode usar ferramentas tradicionais, utilizando até mesmo diagramas de fluxo de dados para representar o comportamento do objeto.

5. Coad/Yourdon - Peter Coad e Edward Yourdon com seu enfoque simples, porém eficaz, dividiram a análise orientada a objeto como sendo classes e objetos. Objetos são abstrações de um domínio de problema que reflete a habilidade de sistema em reter dados e encapsular valores de atributos e serviços.

6. Martin/Odell - Propuseram, na visão de muitos observadores do mercado, uma extensão orientada a objeto da Engenharia da Informação.

7. Wirfs/Brock - Rebecca Wirfs - Brock propõe uma metodologia dirigida a responsabilidades cujo maior diferencial frente aos demais métodos é o uso de cartões CRC - *Class-Responsibility-Collaboration*, desenvolvidos por Ward Cunningham. Tais cartões permitem um mapeamento de classes para execução de certas responsabilidades através da colaboração mútua.

8. Embley/Kutz - Da universidade de Brigham Young, enfatizaram uma forte fundamentação na modelagem de dados, o que inclui um modelo objeto/relacionamento que suporta vários relacionamentos.

Duas destas metodologias de documentação destacavam-se pela aceitação e crescimento a nível mundial, os métodos de Booch e OMT. Seus autores, Grady Booch e James Rumbaugh juntaram forças através da Rational Corporation para forjar uma unificação completa de seus trabalhos. Em 1995 lançaram a primeira proposta do Método Unificado. Tinha sido dado então o primeiro passo da UML. Ainda em 1995, Ivar Jacobson juntou-se aos dois primeiros e fundiu à linguagem unificada, o seu método *OOSE* (*Object Oriented Software Engineering*). A motivação destes era a criação de uma linguagem de modelagem unificada [18], [1].

A UML desenvolveu-se então, a partir da necessidade que equipes multidisciplinares tinham de se comunicar. A UML traduz para a mesma linguagem, as diversas linguagens técnicas envolvidas no desenvolvimento do projeto.

Outro ponto importante inerente à UML, é a qualidade da documentação e do planejamento que um projeto baseado nesta tecnologia pode alcançar. Neste trabalho, utiliza-se a UML para realização da especialização das classes através dos diagramas de classes, simplificando bastante o entendimento das responsabilidades, relacionamentos e heranças entre as classes envolvidas neste projeto.

Num projeto de grande escala onde vários programadores participarão do desenvolvimento, tais características referentes a documentação são fundamentais, e muitas vezes fazem a diferença entre um bem e um mal sucedido projeto.

Existem diversos softwares para desenvolvimento baseado na UML. No pacote Visual Studio da Microsoft encontra-se o Visual Modeler. Neste trabalho utilizou-se o software Together, desenvolvido sob coordenação de Peter Coad na TogetherSoft. Together é distribuído gratuitamente para instituições de ensino. Num dos tópicos que seguem, será abordado com mais detalhe o software Together.

UML é um formalismo para descrever um projeto orientado para objetos. É estruturada como um conjunto de diagramas, contendo cada um o seu propósito específico, conforme [18].

2.2.1 Diagramas de Classes

Os diagramas de classes estão contidos no grupo de diagramas, que representam a base para a UML, fazem parte das representações estáticas dos sistemas. Os elementos estáticos de um modelo, são os conceitos significativos para um dado aplicativo, incluindo conceitos de um mundo real, conceitos abstratos, conceitos de implementação, conceitos computacionais, ou seja, todo tipo de conceito encontrado em sistemas. Por exemplo, um sistema para venda de ingressos para um teatro possui os conceitos de: ingressos; reservas; planos de assinatura; algoritmos para escolha de assentos; páginas interativas na internet para compra de ingressos e etc.

A representação estática do sistema captura a estrutura de objetos do sistema. Os diagramas de classes representam de maneira gráfica, o estado estático do sistema. Entidades gráficas denotam as características de um desenvolvimento orientado para objetos. Representa-se as classes, sua genealogia, e todo relacionamento com as demais classes envolvidas no projeto. A Figura A.1 (pag 73) ilustra um diagrama de classes.

Classes

No diagrama de classes representa-se uma classe com um retângulo. Subregiões neste retângulo são delimitadas, separadas por linhas horizontais. A região mais acima contém o nome da classe, logo abaixo encontra-se a região destinada às propriedades da classe, abaixo das propriedades encontra-se a região alocada aos métodos.

Quando uma classe é uma classe com templates, o retângulo é dividido em quatro regiões e a porção mais acima contém o parâmetro *template* para *template class.*

Associações

Uma associação descreve uma conexão discreta entre objetos ou outras instâncias em um sistema. Uma associação relaciona uma lista ordenada de duas ou mais entidades, com repetições permitidas. O mais comum tipo de associação é uma associação binária entre um par de entidades. Uma instância de uma associação é um *link*, que é representado por uma linha sólida entre as classes correspondentes.

Associações carregam informações sobre relacionamentos entre objetos num sistema. Na execução de um aplicativo, ligações entre módulos do sistema são criadas e destruídas. A associação é o agente responsável pela aglutinação do sistema como um todo. Sem associações não há a definição de um sistema, e sim uma série de classes que não trabalham juntas.

Dependências

Uma dependência indica uma relação semântica entre dois ou mais modelos elementares. Relaciona os modelos elementares por si só, e não necessita de uma série de instâncias para seu significado. Indica uma situação na qual uma mudança no servidor pode necessitar uma mudança no elemento cliente, ou indicar uma mudança de significado para o mesmo na dependência.

Dependências são graficamente representadas por linhas tracejadas, com uma seta que aponta para o elemento servidor.

Generalizações

A relação de generalização é uma relação classificatória entre uma descrição mais genérica e uma descrição mais específica. A descrição específica está baseada na descrição genérica sendo esta estendida pela descrição específica. A descrição mais específica é totalmente consistente com a descrição genérica (possui todas as propriedades, membros e relacionamentos) e a maioria contém informações adicionais.

Generalizações são as representações gráficas das heranças de classes, uma linha sólida liga as duas classes envolvidas no relacionamento por generalização, uma seta sólida aponta para a superclasse.

2.2.2 Diagramas de Sequência

Na primeira fase do projeto utilizavam-se os diagramas de classes para planejamento e documentação das classes envolvidas no projeto. Nesta segunda fase estende-se o uso da UML para incluir os diagramas de sequência. Estes representam dinamicamente o comportamento do sistema.

Os diagramas de sequência estão ligados com a visão comportamental do código. Os objetos tratados no diagrama trocam mensagens dentro de um intervalo de tempo predeterminado [1].

O aplicativo Together de desenvolvimento e documentação orientado para objetos, baseado na UML , traduz um diagrama de sequência para o código fonte de uma linguagem especificada. Sendo possível também o processo inverso, onde o citado aplicativo interpreta o código e cria com isso um diagrama de sequência que representa o método passado como parâmetro. Este processo é conhecido como "engenharia reversa".

Um diagrama de sequência utiliza cinco conceitos. Os tópicos a seguir tratam destes conceitos.

Interações

Interações são entidades que definem sequências de troca de mensagens ou modelos de sequência de troca de mensagens entre outras entidades envolvidas no cumprimento de determinada tarefa. Interações são utilizadas para modelar comunicações entre entidades. São graficamente representadas pelas setas nos diagramas. A Figura 2.2 (pag 20) apresenta exemplos de interações.

Atribuições das Classes

São entidades que definem papéis ou funções específicas executadas pelas classes envolvidas nas interações ou colaborações. São utilizadas para modelar papéis que entidades executam dentro

de interações ou colaborações. São os métodos utilizados nas interações, encontram-se acima das setas representantes das interações. A Figura 2.2 (pag 20) apresenta exemplos das Atribuições das Classes.

Linhas de Existência

São entidades gráficas que representam a existência de atribuições de classes durante um período de tempo. Modelam a existência de entidades ao longo tempo. A Figura 2.2 (pag 20) apresenta exemplos de Linhas de Existência.

Ativações

São entidades gráficas que representam o espaço de tempo durante o qual um atributo de classe está executando uma tarefa, ou quando está ativo e possui o foco (*thread*). São utilizados para representar um espaço de tempo durante o qual entidades estão ativas ou executando alguma operação, e também para modelar relações de controle entre entidades. A Figura 2.2 (pag 20) apresenta exemplos de Ativaçoes.

Mensagens

São entidades que definem a informação que é trocada em interações e colaborações. São utilizadas para modelar o conteúdo da comunicação entre entidades. São utilizados para fornecer informações das entidades e habilitar entidades a requerer informações de outras entidades. Atributos de classes comunicam-se entre si através de mensagens.

Uma requisição é uma especificação de comunicação, e uma mensagem mostra uma requisição em uma interação. Quando duas instâncias comunicam-se, uma instância de uma mensagem é passada entre elas. Uma mensagem possui um remetente e um destinatário e possivelmente mais informações com relação ao tipo de requisição sendo passado. Um evento no destinatário sinalizará o formato da mensagem no remetente.

As mensagens são os parâmetros passados pelas Atribuições de Classes (métodos) nas interações. A Figura 2.2 (pag 20) apresenta exemplos de Mensagens.

A Figura 2.2 (pag 20) ilustra um diagrama de sequência gerado a partir do método Compress da classe TPZFront.

Os diagramas de sequência, bem como os de classes, mostram-se ferramentas muito úteis no desenvolvimento de projetos de grande escala. Facilitam sobremaneira a comunicação entre as equipes envolvidas no projeto. Aumentam a focalização sobre o problema específico.

As Figuras 2.1, 2.3 e 2.4 (pags 19, 20 e 21) trazem diagramas de sequência de alguns dos métodos envolvidos no processo frontal, alguns foram criados com engenharia reversa (Together interpreta o código e monta o diagrama) e outros foram criados graficamente, sendo o código gerado posteriormente.

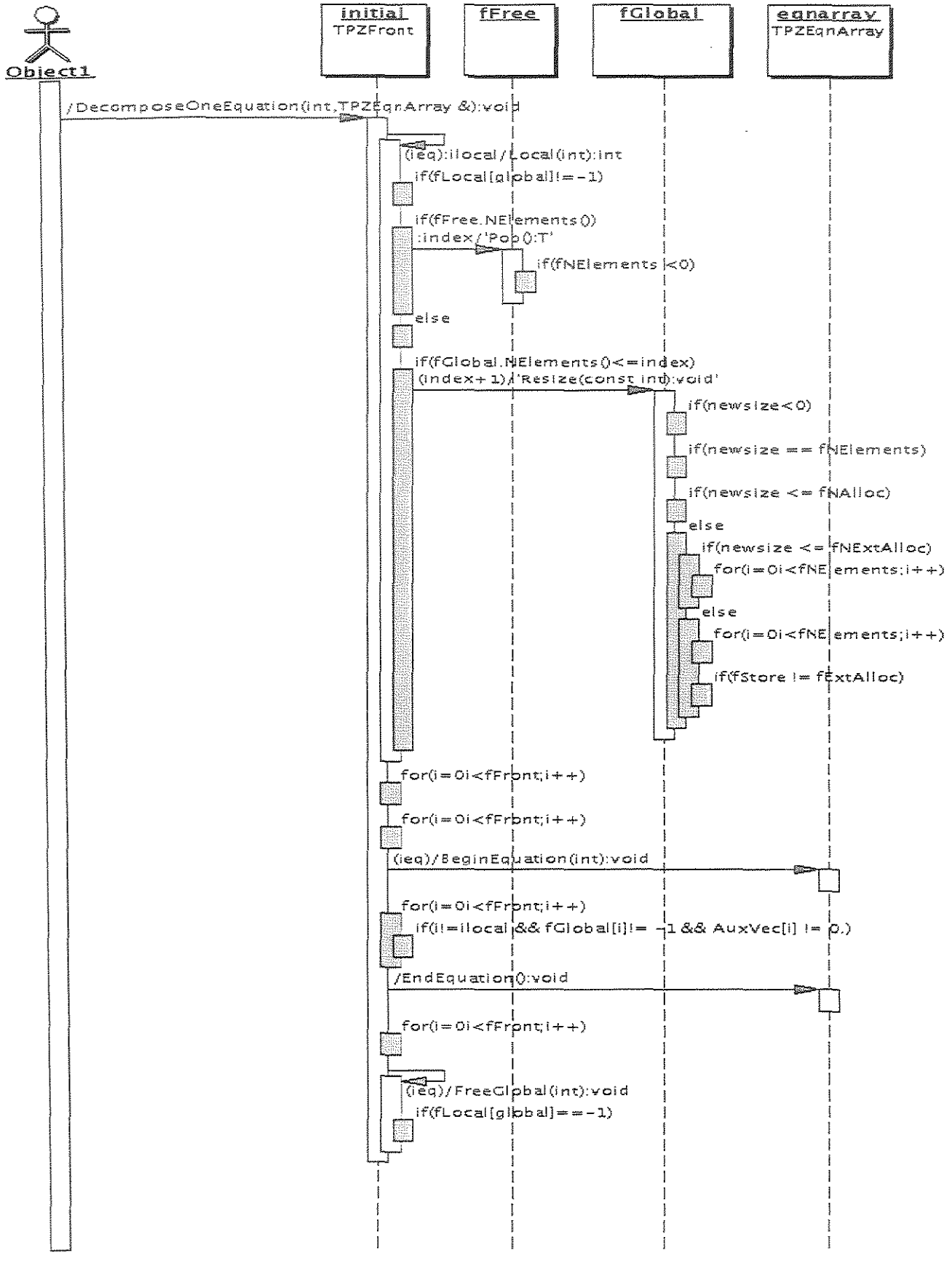


Figura 2.1: Diagrama de sequência DecomposeEquations da classe TPZFrontMatrix.

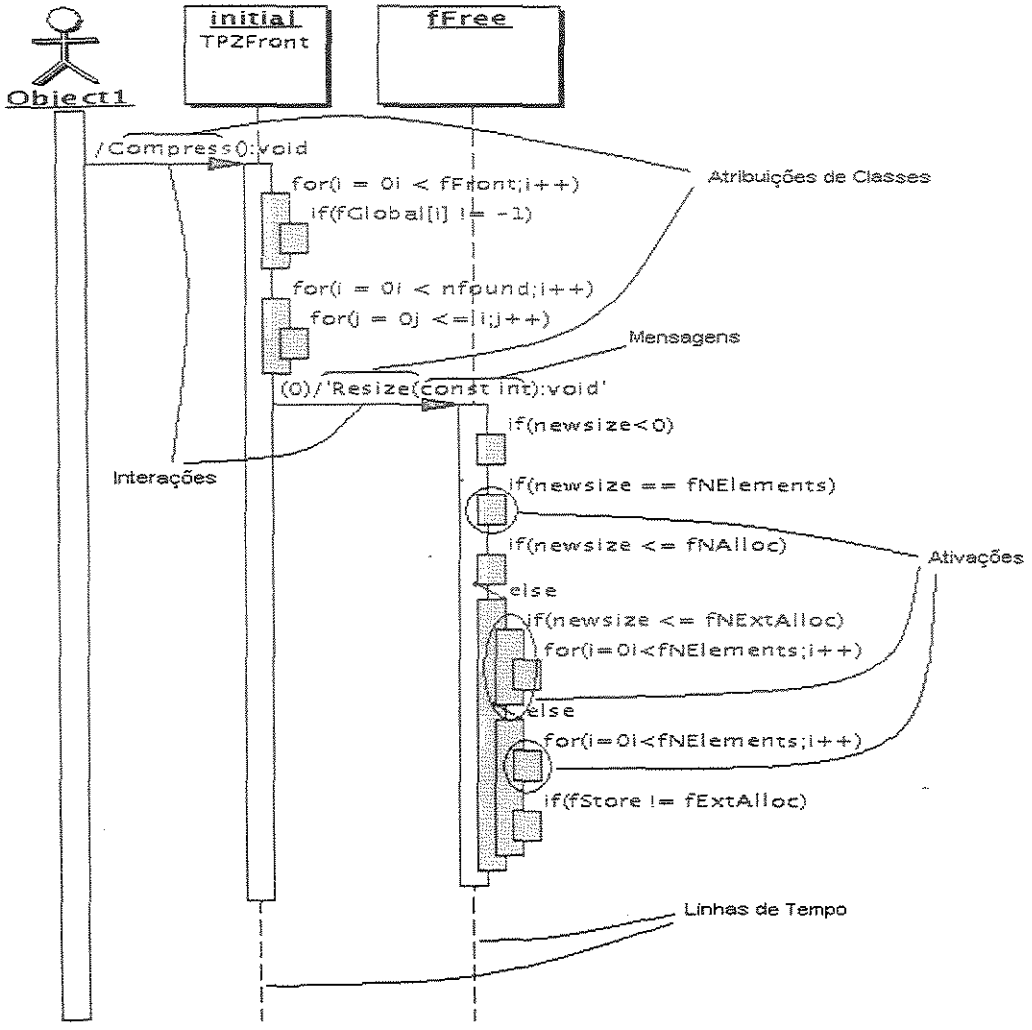


Figura 2.2: Diagrama de sequência do método Compress

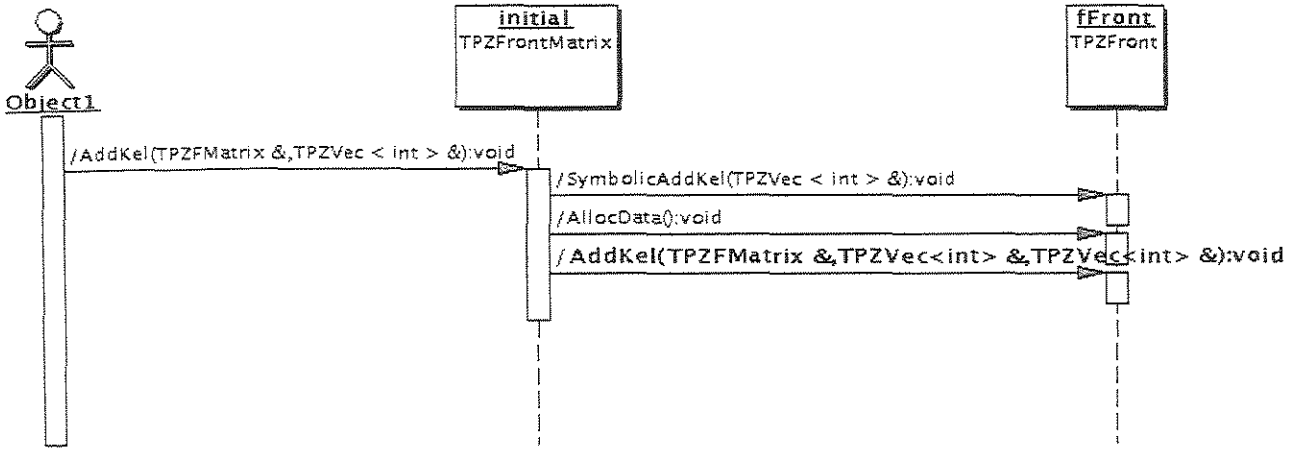


Figura 2.3: Diagrama de sequência do método AddKel.

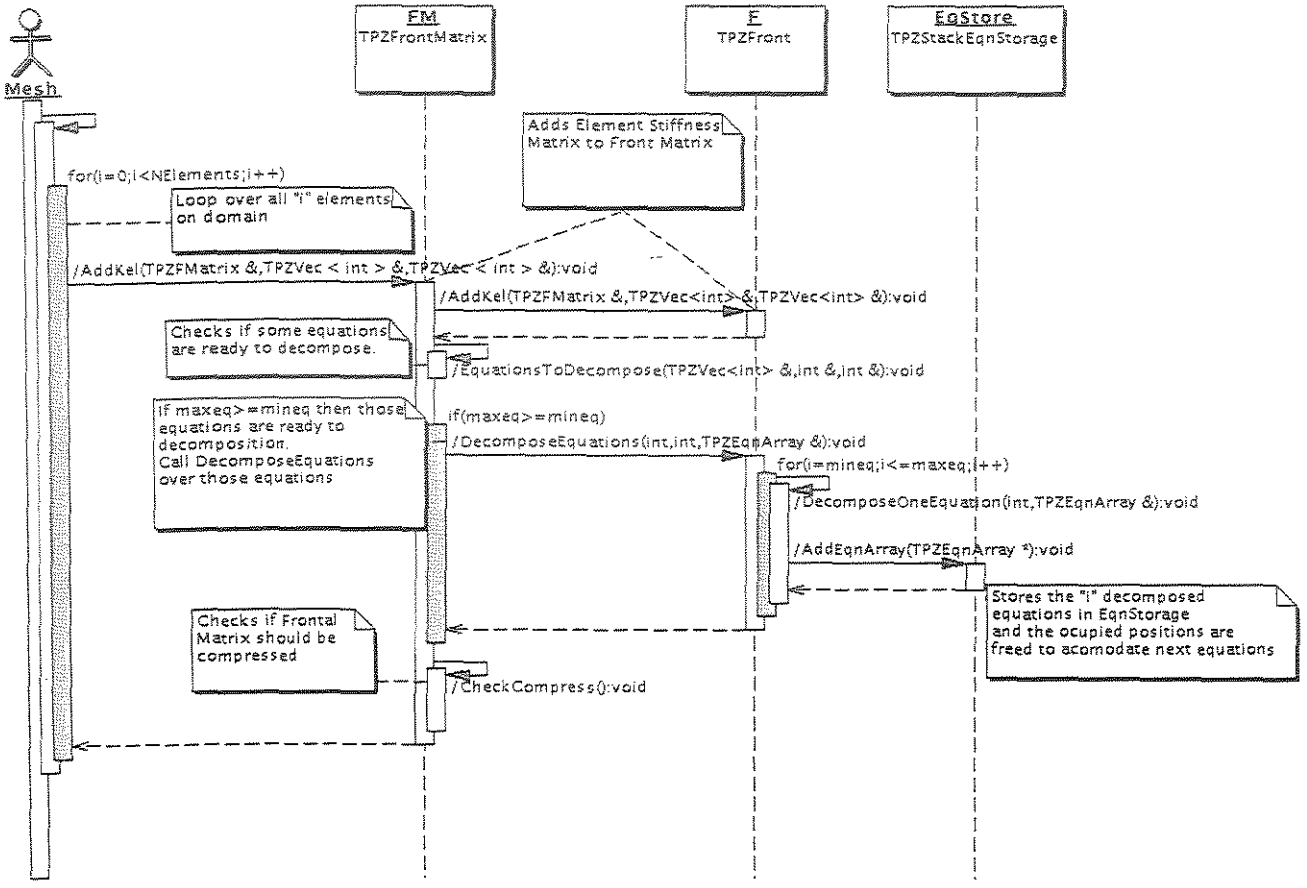


Figura 2.4: Diagrama de sequência do método frontal.

2.3 Tecnologia *COM (Component Object Model)*

No mercado das linguagens de programação, fabricantes buscam isolar um nicho e focar o desenvolvimento para aquela fatia do mercado. Sendo bem distintos alguns nichos:

- Desenvolvimento de aplicativos de banco de dados - São aplicativos totalmente concebidos a partir de uma necessidade de gerenciamento ou obtenção de informações dissolvidas em várias tabelas de dados, que se correlacionam gerando tal informação. Uma linguagem que é especialmente desenvolvida para este tipo de aplicação é o FoxPro da Microsoft.
- Desenvolvimento de aplicativos para internet - O principal mercado é o dos desenvolvedores de *sites* (www). Um aplicativo desenvolvido para internet deve ser perfeitamente multi-plataforma, ou seja, executará da mesma maneira em diferentes padrões de sistemas operacionais. A linguagem *JAVA* é um bom exemplo, os fabricantes garantem que um aplicativo desenvolvido em *JAVA* funcionará em qualquer sistema operacional.
- Desenvolvimento científico - A linguagem *FORTRAN* é muito eficiente para cálculos numéricos com estruturas de dados simples.
- Interfaces Gráficas - O ambiente de desenvolvimento do *Visual BASIC* permite num curto espaço de tempo, a criação de sofisticadas interfaces gráficas para Windows, de maneira simples.
- Complexidade de Dados - A solução concebida para resolver um problema pode envolver algoritmos e estruturas de dados complexas. *C++* é muito eficiente no gerenciamento desta complexidade.

Estas linguagens são especializadas em seus nichos. Não recomenda-se desenvolver um projeto com uma linguagem que não se encaixe no nicho da mesma.

A execução do software não é o único tópico que se especializa. A principal característica do *Visual BASIC* é a facilidade e a velocidade com a qual um programador cria um aplicativo inteiro. *Visual BASIC* é uma linguagem *RAD* (*Rapid Application Development*) [22], especializada em desenvolvimento rápido. A Figura 2.5 (pag. 23) representa o ambiente de desenvolvimento para *Visual BASIC*.

Visto esta especialização das linguagens, torna-se essencial para desenvolvimento corporativo, a utilização de várias linguagens, onde cada equipe de programadores utiliza a linguagem mais apropriada à sua função. São desenvolvidos e testados separadamente as partes integrantes do sistema global.

Uma equipe conectará todas as partes desenvolvidas gerando o software final. Esta conexão será realizada com a tecnologia *COM* (*Component Object Model*). Estes objetos são também conhecidos como *Controles ActiveX*. [12], [2] .

ActiveX é uma tecnologia de modularização desenvolvida e de propriedade da *Microsoft*. Os *Controles ActiveX* podem ser utilizados por qualquer linguagem de programação com suporte à *COM*. Podem também ser incorporadas em arquivos tipo *Microsoft Office*. Por exemplo, quando uma planilha *MS EXCEL* é inserida em um arquivo *MS WORD*, esta planilha é, para o *MS WORD*, um *controle ActiveX*.

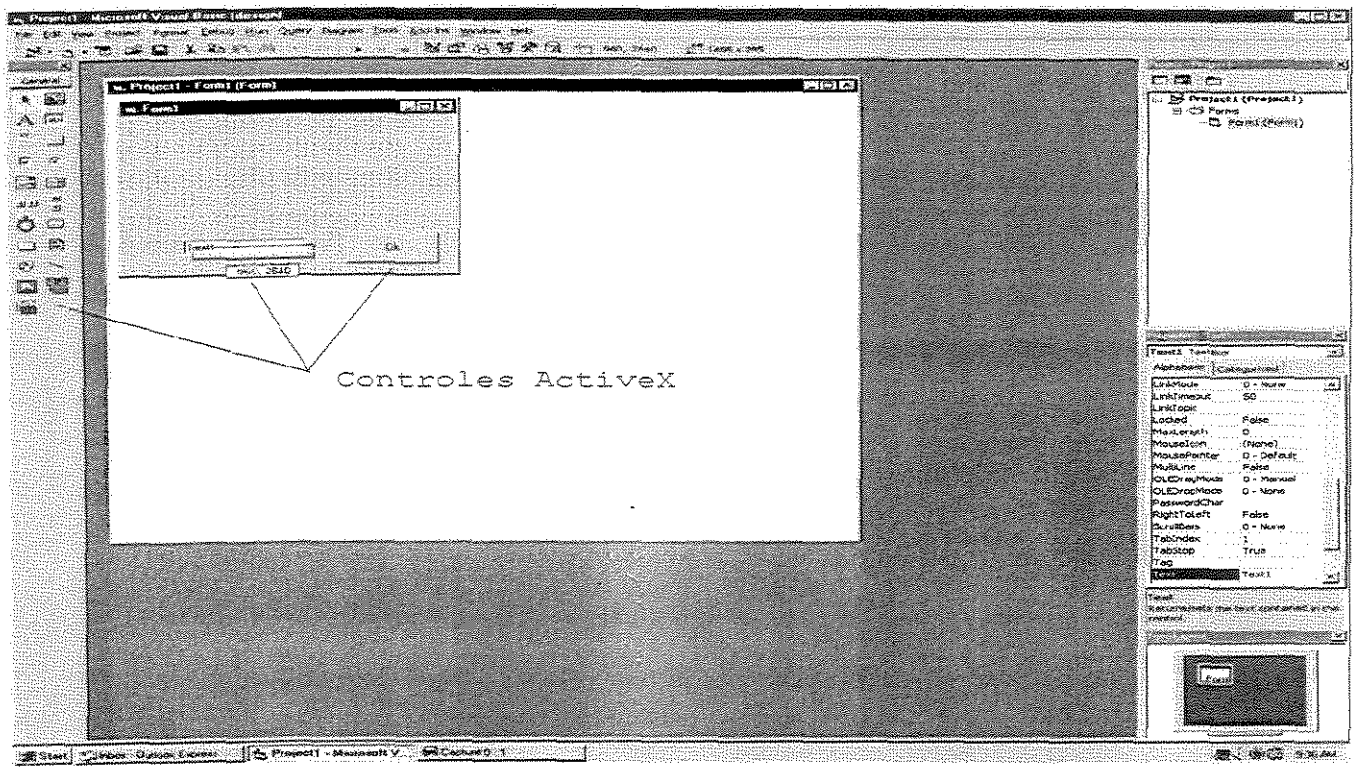


Figura 2.5: Desenvolvimento no Visual Basic

Este desenvolvimento utilizando várias linguagens, é também conhecido como *MLP* (*Mixed Language Programming*).

O motivo pelo qual pretende-se encapsular o solver frontal como *Controle ActiveX* é para poder disponibilizar o mesmo para outras linguagens de programação.

Um Solver implementado em *C++* é eficiente quanto a estruturação e gerenciamento de dados. Sua posterior disponibilização em formato *COM* "populariza" à comunidade *Visual BASIC* um solver desenvolvido em *C++*. As características das diferentes linguagens combinam-se formando um produto final de alta qualidade.

2.3.1 *Microsoft Foundation Classes* MFC

Juntamente com o pacote *Visual Studio* da *Microsoft* é distribuída a MFC "*Microsoft Foundation Classes*". A MFC é uma biblioteca de classes contendo todas as classes envolvidas na elaboração do ambiente gráfico e as facilidades do sistemas operacionais Windows. Contém desde classes que facilitam o gerenciamento de strings até classes para programação de redes com seus vários protocolos e formatos.

As classes MFC apresentam uma programação para o ambiente Windows orientada para objetos. As classes MFC encapsulam as tradicionais API do Windows (*Application Programming Interface*) [24], [2].

Sob uma filosofia orientada para objetos, especializam-se as classes fornecidas pela microsoft buscando a funcionalidade desejada no software sendo desenvolvido. Particularmente neste projeto

utiliza-se a MFC para elaboração de *Property Pages.*

2.3.2 *Property Pages*

Em alguns *controles ActiveX* há a necessidade de uma configuração inicial para obtenção da funcionalidade esperada. Esta configuração muitas vezes deve ser feita em *Design Time* e não em *Run Time.* Para tanto o programador que desenvolve o controle (em *C++* por exemplo) deve disponibilizar uma janela para execução desta configuração por parte do programador que estará futuramente utilizando o controle em outra linguagem (*Visual Basic* por exemplo).

Esta janela é denominada *Property Page*, é essencialmente uma interface gráfica para configuração em *Design Time* de um *controle ActiveX*. Entende-se por *Design Time* o período de edição de um programa e *Run Time* pelo período em que este está sendo executado. Configurações podem ser executadas no momento da edição do programa (*Property Pages*), ou então enquanto este está rodando através de instruções em seu código fonte.

Para o solver gradiente conjugado pré condicionado disponibilizado em um controle ActiveX, uma janela *Property Page* foi implementada. Através desta possibilita-se a escolha do erro admissível na resolução do sistema de equações. A Figura 2.6 (pag 25) mostra uma property page de um controle ActiveX num projeto Visual Basic.

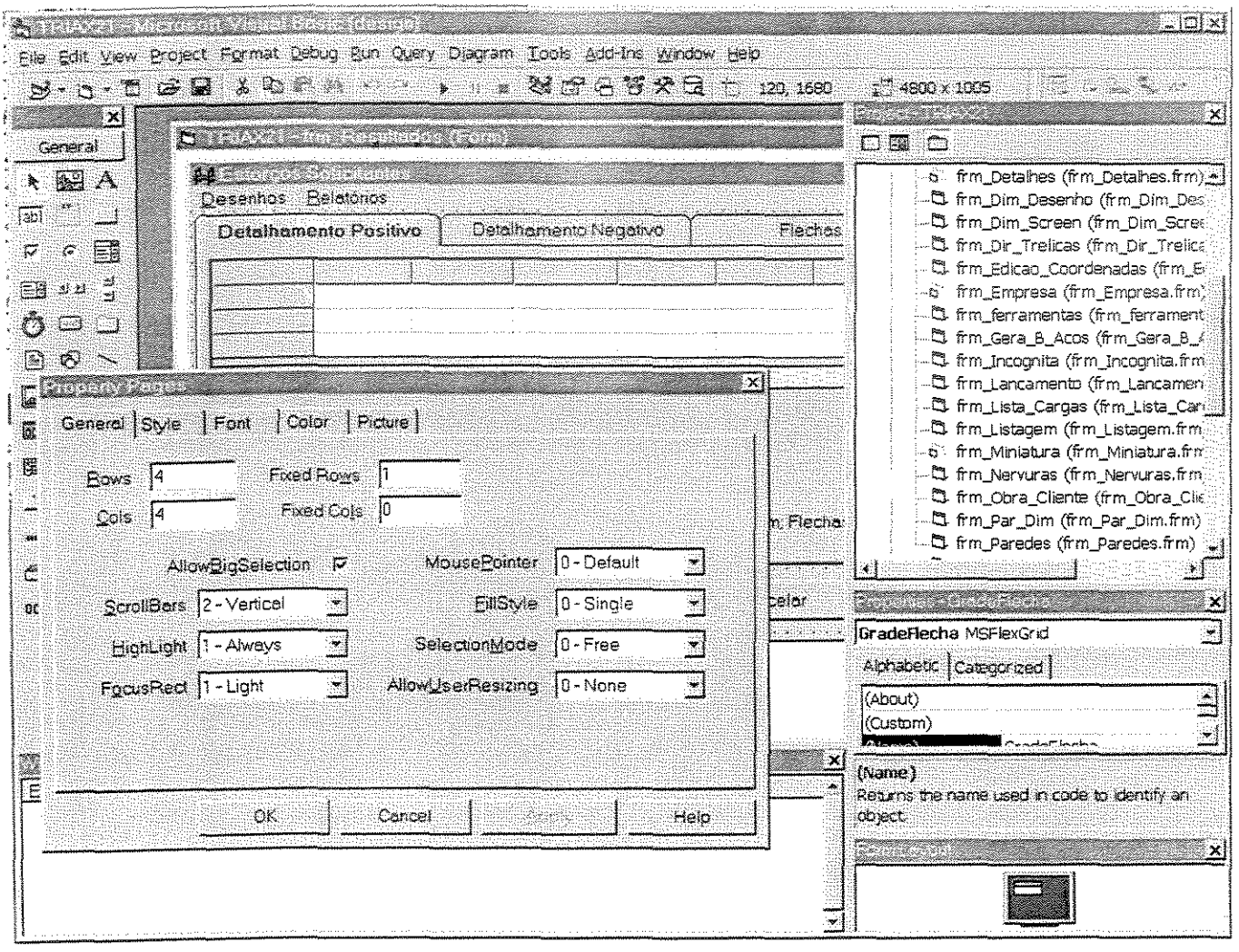


Figura 2.6: Property Page de um controle ActiveX.

Capítulo 3

Programação Paralela

3.1 Programação Paralela

Existem duas abordagens distintas para se trabalhar com computação paralela. Esta diferença decorre do tipo de Hardware sendo utilizado.

Um hardware possibilita o paralelismo com memória distribuída. Ocorre quando mais de uma máquina estão envolvidas no processo computacional. Cada computador possui seu banco de memória e este não está disponível às outras máquinas presentes no processo. A comunicação entre os equipamentos é feita com um dispositivo específico (switch de alta performance).

Uma segunda configuração de Hardware possibilita o paralelismo com memória compartilhada. Uma máquina possui mais de um processador, que acessam o mesmo banco de memória. Os processadores compartilham o mesmo banco de memória. Estas máquinas biprocessadas já são bem accessíveis.

Neste projeto utiliza-se memória compartilhada, pois o laboratório de mecânica computacional da Faculdade de Engenharia Civil - UNICAMP, LabMeC já possui equipamentos com estas características.

3.1.1 Multi-tarefa (*Multi-threading*)

Num processo multi-tarefa, existem mais de uma sequência de instruções sendo executadas ao mesmo tempo, havendo ou não compartilhamento dos dados. Um programa com estas características será multi-tarefa.

Segundo o manual da SUN de programação multi-tarefa [47], define-se uma tarefa como sendo uma única sequência de instruções (processo) que serão executadas por um ou mais programas. Durante este processo uni-tarefa, existe apenas um ponto de execução a cada instante.

A simples utilização de uma máquina com recursos paralelos não significa que estejam ocorrendo processos multi-tarefa. É necessário que o código gerencie tanto as instruções aos processadores, quanto os dados sendo compartilhados.

Foram utilizadas neste projeto duas plataformas; Microsoft WINDOWS NT e GNU&Linux. Para ambas as plataformas encontram-se as bibliotecas com as ferramentas *multi-thread*. Estas possuem basicamente as mesmas funções, variam quanto a nome e sintaxe.

As bibliotecas multi-tarefa utilizados estão disponíveis sob formato freeware, a biblioteca pthread do sistema Linux e biblioteca multiplataforma OmniThread desenvolvido pela Oliveti [39], sendo esta última a escolhida para o desenvolvimento na plataforma Windows.

Como a parte algébrica deste trabalho opera em duas plataformas, uma descrição para cada uma delas segue nos próximos tópicos. As referências [25] e [8] tratam da programação paralela com multi-tarefa. Na referência [25] encontra-se uma extensão dos manuais POSIX direcionado aos threads, enquanto que [8] apresenta uma introdução à programação paralela utilizando multi-threading. Os tópicos a seguir tratam das duas bibliotecas utilizadas.

3.1.2 Biblioteca *OMNI Thread*

Apesar de ser uma biblioteca multiplataforma, esta foi utilizada apenas no desenvolvimento para WINDOWS NT, visto que o próprio Linux traz uma biblioteca específica. Desenvolvida por uma união da *Tristan Richardson Olivetti & Oracle Research Laboratory* em Cambridge em 1997 [39], `http://www.uk.research.att.com`.

A abstração OMNI Thread foi desenvolvida para disponibilizar ferramentas multi-tarefa para serem usadas com *C++* possuindo a mesma sintaxe; a chamada é a mesma em qualquer plataforma. A interface foi desenvolvida para ser semelhante à interface *C* para os pthreads (POSIX) [8], [25], [38].

Abstração OMNI Threads

A abstração OMNI consiste num encapsulamento das diferentes bibliotecas para as diferentes plataformas, numa classe multi-tarefa independente de plataforma. Diretivas de compilação são as definições necessárias para uso deste pacote, com elas defini-se em qual ambiente se está trabalhando.

Na versão utilizada estão implementadas as interfaces para as seguintes plataformas: [39].

- Solaris 2.x utilizando pthreads versão 10
- Solaris 2.x utilizando threads solaris (mas a versão pthreads é agora comum)
- Alpha OSF1 utilizando pthreads versão 4
- Windows NT utilizando NT threads
- Linux 2.x utilizando Linuxthread 0.5 (que é baseado em pthreads versão 10)
- Linux 2.x utilizando MIT pthreads (que é baseado na versão 8)
- ATMos utilizando pthreads versão 6

Objetos de Criação de Linha de Execução

Para a criação de uma nova tarefa é necessário a declaração de um objeto do tipo omni_thread, onde no seu construtor um dos parâmetros fornecidos é a função a ser executada na nova linha de execução e os dados que esta utiliza.

Objetos de Sincronização

Objetos de sincronização são utilizados para gerenciar (sincronizar) tarefas dentro do mesmo processo. A biblioteca não disponibiliza sincronização entre processos diferentes. Os objetos de sincronização disponíveis são as travas de exclusão mútuas *mutex*, variáveis de condição (*condition variables*) e semáforos contadores (*counting semaphores*).

1. Mutex (*Mutual Exclusion Locks*)
   Um objeto do tipo omni_mutex é utilizado para exclusão mútua. Impede que dois threads acessem o mesmo dado ao mesmo tempo. Possui duas operações, *lock()* (travar) e *unlock()* (destravar). Pode ser utilizado também os nomes alternativos *acquire()* e *release()*.

2. Variáveis de Condição (*Condition Variable*)
   Uma variável de condição é representada por uma *omni_condition* e é usada para sinalização entre threads. Uma chamada para *wait()* faz com que um thread espere pela sinalização da variável de condição. Caso, devido a uma chamada a *wait()*, existam threads esperando , uma chamada para *signal()* reinicia ao menos um thread. Uma chamada para *broadcast()* reinicia todos os threads que estão esperando, e não apenas um deles como *signal()*.
   Quando da construção de um *omni_condition*, deve ser fornecido um ponteiro para *omni_mutex*. Uma chamada ao método *wait*() executa encapsuladamente uma chamada de *unlock*(). Quando um processo recebe um *signal()* ou *broadcast()* uma chamada a *lock*() é executada. A conexão entre um mutex e uma variável de condição existe enquanto existir a variável de condição (ao contrário de pthreads onde a conexão dura apenas o tempo de espera). O mesmo mutex pode ser usado com diferentes variáveis de condição.
   Uma chamada para *wait*() com um temporizador pode ser obtida utilizando *timed_wait*(). Passa-se como parâmetro um tempo absoluto a ser esperado. O método *omni_thread::get_time*() pode ser usado para transformar um tempo relativo em um tempo absoluto. É retornado de *timed_wait*() a constante *ETIMEDOUT* caso acabe o tempo de espera antes que a variável de condição seja sinalizada.

3. Semáforos Contadores (*Counting Semaphores*)
   Quando criado, um *omni_semaphore* recebe um valor inteiro inicial. Quando *wait*() é chamado o valor do inteiro é decrementado (caso este não seja zero), caso contrário o *thread* é suspenso como numa chamada *wait*(). Quando *post*() é chamado, caso haja algum thread suspenso em *wait*(), exatamente um *thread* é reiniciado. Caso não existam *threads* suspensos, então o valor do contador é incrementado.
   Se um *thread* chama *try_wait*(), então o *thread* não irá bloquear (suspender) se o valor do semáforo for igual a zero, retornando neste caso a constante *EAGAIN*.

No desenvolvimento deste projeto foram utilizados os métodos *lock()* e *unlock()* da classe *omni_mutex*. Da classe *omni_thread* os métodos *wait()*, *signal()* e *broadcast()*.

De maneira geral, trabalhar com as ferramentas *multi-threading* encapsuladas em uma biblioteca orientada para objetos, se mostrou vantajoso. Uma vez definidas as variáveis de sistema necessárias para o funcionamento do pacote, as chamadas aos métodos são bem mais suscintas.

3.1.3 Biblioteca *pthread*

A implementação na plataforma Linux foi realizada utilizando a biblioteca paralela *pthread* (*POSIX*) que acompanha o sistema Linux [38], [27], [37], [58], [25] e [8]. Os tópicos seguintes tratam das particularidades desta biblioteca e de sua aplicação.

Por não ser uma biblioteca orientada para objetos, encontram-se as funções para criação, gerenciamento e controle das tarefas e também a estrutura de dados e inicialização das mesmas. As funcionalidades das rotinas da biblioteca pthread são as mesmas da biblioteca OMNI

Uma tarefa na biblioteca pthread é representado por uma estrutura de dados do tipo pthread_t. Sua declaração é feita da seguinte maneira:

- pthread_t my_thread;

Um mutex na biblioteca pthread é um objeto do tipo pthread_mutex_t. Sua declaração é feita da seguinte maneira:

- pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
  A inicialização de my_mutex é feita com o macro
  PTHREAD_MUTEX_INITIALIZER

Uma variável de condição na biblioteca pthread é um objeto tipo pthread_cond_t. Sua declaração é feita da seguinte maneira:

- pthread_cond_t my_cond = PTHREAD_COND_INITIALIZER;
  A inicialização de my_cond é feita com o macro
  PTHREAD_COND_INITIALIZER

As funções utilizadas para as atuais implementações na plataforma Linux foram:

- *pthread_create*(*pthread*, pthread_attr_t *, void *(*start_routine)(void *), void *arg) - cria uma nova tarefa (*thread*). Neste o thread já começa a sua execução utilizando o ponteiro para a função fornecido pelos argumentos.
- *pthread_join*(*pthread*) - força a tarefa que está chamando a função *pthread_join* a esperar pela conclusão do *pthread* passado como parâmetro.
- *pthread_cond_broadcast*(*pthread_cond_t *Cond*) - A variável de condição *Cond* é o objeto de sincronização. Quando alguma condição predeterminada é satisfeita durante o processamento, esta função sinaliza a variável *Cond*.
- *pthread_cond_wait*(*pthread_cond_t *Cond*, *pthread_mutex * mutex*) - Automaticamente destrava o *Mutex* (*Mutual Exclusion Lock*) e espera a variável de condicionamento ser sinalizada. A execução da tarefa é suspensa e não há consumo de tempo de CPU aguardando a variável de condição

- *pthread_mutex_lock(pthread_mutex_t * mutex)* Um dispositivo de exclusão mútua é necessário para proteger dados compartilhados de manipulação concorrente. Um *Mutex* possui dois possíveis estados, destravado (nenhuma tarefa retém a posse do *Mutex*) e travado (uma tarefa tem posse sobre o *Mutex*).
  Nunca duas tarefas poderão possuir o mesmo *Mutex* simultaneamente. Uma tarefa tentando travar um *Mutex* já travado será suspensa até que este sejá destravado. Um *thread* que possui um *Mutex* tem direito de manipular os dados compartilhados.

3.1.4 Implementação e Depuração do Código Paralelo

Ao contrário de implementações não paralelas (seriais), onde existe apenas um caminho para chegar do ponto a até o ponto b, quando uma rotina é concebida para funcionar em paralelo, haverá em algum ponto do código uma bifurcação dos processos, e caberá ao programador implementar os meios de sincronização de todos os processos.

Suponha uma máquina com dois processadores e que um processador executa a tarefa 1. Uma chamada à uma função de criação de threads feita pela tarefa 1 inicia uma nova tarefa. Tarefa 2. A tarefa 2 é executa paralelamente à tarefa 1. Cada uma delas é executada em um processador. Sabe-se que um processador executa a tarefa 1 e outro a tarefa 2 mas não qual processador executa qual tarefa.

Durante a execução de um código paralelo, duas tarefas podem estar manipulando os mesmos dados. Então, a implementação de um código em paralelo requer um maior rigor no planejamento/desenvolvimento do código. O programador deve ater-se não só com a funcionalidade esperada pela rotina, mas sim com os dados que esta irá acessar. Qual parte do processo será paralelizada, quais dados serão protegidos e etc.

Além da dificuldade inerente à concepção da rotina paralelizada, há também complicações relacionadas à depuração destas.

Na depuração de um código simples o programador acompanha a execução do único *thread* existente em seu código através do depurador. O mesmo trabalho em uma rotina com paralelismo é algo mais complexo.

Num processo paralelo há, além do *thread* sendo depurado, um segundo (ou terceiro ...) *thread* que pode não estar sendo acompanhado pelo depurador. Isto aumenta o escopo da análise. Esta não mais limita-se ao que o depurador apresenta, e sim a possíveis tarefas concorrentes sendo executadas em *threads* não visualizados pelo depurador.

Um dos artifícios para depuração de códigos são os pontos de parada (*break points*). O código é executado até a linha onde foi inserido um ponto de parada. Desta linha em diante o código pode ser executado passo a passo ou prosseguir até um próximo ponto de parada ou o final do código. É utilizado para inspecionar valores contidos nas variáveis do programa naquele instante do processo.

O funcinamento de um código paralelo pode ser mascarado pela posição escolhida pelo programador para inclusão de um ponto de parada. Há situações em que a inclusão de um ponto de parada faz com que o código funcione. Na realidade o sucesso se deve não ao correto funcionamento da rotina, e sim a uma mudança de comportamento devido a escolha do ponto de parada.

Notou-se que no sistema linux a interrupção de um código paralelo durante sua execução deixa

informações pendentes no sistema operacional. Estas informações pendentes acarretam um comportamento anormal da máquina. Este descuido pode acarretar em várias horas de depuração na procura de erros que não existem. A solução foi a reinicialização periódica do sistema operacional.

3.2 Experimento com Programação Paralela

Foi desenvolvido na fase inicial do projeto, um solver para matrizes tipo Skyline, com fatoração por Cholesky. O objetivo desta implementação foi servir de experimento e introdução às técnicas de otimização por paralelismo em máquinas com memória compartilhada. Esta experiência foi muito produtiva e de fato prestou um ótimo trabalho na introdução aos conceitos utilizados.

3.2.1 Solver Cholesky para Matriz Skyline com Fatoração Paralela

Quando a matriz dos coeficientes das equações for positiva definida e simétrica, um método bastante apropriado para fatoração da matriz dos coeficientes do sistema é o método direto de *Cholesky* [26]. Segundo Serra [44], este método fornece indicações quanto à montagem correta da matriz de rigidez, pois caso esta não seja definida positiva e simétrica, o método falha, indicando problema com a matriz.

O procedimento é dividido em duas fases, uma primeira que é a fatoração da matriz dos coeficientes no produto de duas matrizes triangulares - inferior e superior - e a posterior que é a solução do sistema, usando a matriz dos coeficientes já decomposta. Implementa-se neste solver a decomposição da matriz dos coeficientes com ferramentas paralelas, onde a fatoração de diferentes equações é executada concorrentemente por mais de uma tarefa.

Decomposição da Matriz Simétrica Positiva Definida.

A implementação não paralela da decomposição da matriz é realizada de acordo com a sequência abaixo:

Seja $[S]$ uma matriz simétrica positiva definida. Esta matriz pode ser fatorada no produto de duas matrizes triangulares (superior e inferior), $[U]^t$ e $[U]$, sendo uma a transposta da outra. Assim

$$[S] = [U]^t.[U]$$

onde $[U]$ é a matriz triangular superior (upper).

Desenvolvendo, tem-se:

$$\begin{bmatrix} S_{11} & S_{12} & S_{13} & \cdots & S_{1n} \\ S_{21} & S_{22} & S_{23} & \cdots & S_{2n} \\ S_{31} & S_{32} & S_{33} & \cdots & S_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ S_{41} & S_{42} & S_{43} & \cdots & S_{nn} \end{bmatrix} = \begin{bmatrix} U_{11} & 0 & 0 & \cdots & 0 \\ U_{21} & U_{22} & 0 & \cdots & 0 \\ U_{31} & U_{32} & U_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ U_{41} & U_{42} & U_{43} & \cdots & U_{nn} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} & U_{13} & \cdots & U_{1n} \\ 0 & U_{22} & U_{23} & \cdots & U_{2n} \\ 0 & 0 & U_{33} & \cdots & U_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{nn} \end{bmatrix}$$

Daí

$$S_{ii} = U_{1i}^2 + U_{2i}^2 + \cdots + U_{ii}^2 \Rightarrow S_{ii} = \sum_{k=1}^{i} U_{ki}^2 \text{ (diagonal principal)}$$

$$S_{ij} = U_{1i}.U_{1j} + U_{2i}.U_{2j} + \cdots + U_{ii}.U_{ij} \Rightarrow S_{ij} = \sum_{k=1}^{i} U_{ki}.U_{kj} \text{ para } i < j \text{ pois}$$

$$U_{ij} = 0 \text{ para } i > j$$

Ou seja,

$$U_{11} = \sqrt{S_{11}} \text{ e } U_{ii} = \sqrt{S_{ii} - \sum_{k=1}^{i-1} U_{ki}^2} \text{ para } 1 < i = j$$

$$U_{1j} = \frac{S_{1j}}{U_{11}} \text{ e } U_{ij} = \frac{S_{ij} - \sum_{k=1}^{i-1} U_{ki}.U_{kj}}{U_{ii}} \text{ para } 1 < i < j$$

$$U_{ij} = 0 \text{ se } i > j$$

Obtendo assim a matriz U decomposta [44].

Decomposição em Paralelo

A sequência do cálculo apresentada na seção anterior, quando implementada em paralelo, pode ser apresentada de diversas maneiras:

1. Dado U_{ii} calculado, calcula-se em seguida

$$U_{ij} = \frac{S_{ij} - \sum_{k=1}^{i-1} U_{ki}.U_{kj}}{U_{ii}} \text{ para } \forall\ j > i$$

   Nota-se que para um dado i todos os U_{ij} podem ser calculados em paralelo

2. Calcula-se a fatoração de cada coluna individualmente. Para uma coluna j

$$U_{ij} = \frac{S_{ij} - \sum_{k=1}^{i-1} U_{ki}.U_{kj}}{U_{ii}} \text{ para } i = 1, ..., j-1$$

$$U_{jj} = \sqrt{S_{jj} - \sum_{k=1}^{j-1} U_{kj}^2}$$

Este é o chamado *procedimento de coluna ativa*, mas é essencialmente sequencial, o que o torna limitado para paralelização. Em [56] e [20] encontram-se outras variações na paralelização do método.

Propõe-se a decomposição da matriz dos coeficientes da seguinte maneira:

Dado um sistema de equações decomposto até a equação i e que dispomos de p processadores.

1. Uma tarefa dedica-se a fatorar a coluna $i+1$, até obter $u_{i+1,k+1}$.
2. As tarefas 2 até p fatoram as colunas $k = i+2...n$ até o cálculo do valor U_{ik}.

Utiliza-se os dispositivos de threads disponíveis em sistemas multi-tarefas para implementar o procedimento acima descrito. As ferramentas paralelas são obtidas com a utilização das citadas bibliotecas.
As tarefas obedecem às seguintes regras:

1. uma tarefa dedica-se a fatorar a coluna $i+1$.
2. somente uma tarefa pode trabalhar em uma dada coluna.

Implementação do Método

Numa decomposição sequencial, uma coluna é fatorada desde a linha 0 até a diagonal principal e isto é feito coluna por coluna. Suponha uma coluna i sendo fatorada, a fatoração das colunas subsequentes só será iniciada após a coluna i ser totalmente decomposta. Numa decomposição paralela não é necessário esperar a total conclusão da coluna anterior para iniciar a fatoração das colunas subsequentes. Ou seja:
Uma coluna i sendo fatorada e no instante que esta fatoração se encontra na linha j.
Poderá ser iniciada a fatoração de colunas além da posição i quando houver disponíveis equações nas colunas posteriores localizadas acima da posição (linha) j. Cada tarefa opera com apenas uma coluna. Há na implementação um método que procura por uma coluna disponível para fatoração. Uma coluna estará disponível para fatoração quando:

- Esta não foi fatorada.
- Nenhuma tarefa estiver trabalhando com esta coluna. Uma tarefa fatorando uma coluna torna-se proprietária desta coluna. Apenas uma tarefa pode trabalhar com uma coluna.
- Esta coluna possuir coeficientes localizados em posição (linha) anterior à atual posição da fatoração na coluna $i-k$.

A Figura 3.1 (pag. 34) apresenta esquemáticamente a matriz SkyLine e as possíveis colunas sendo fatoradas em paralelo.

Desempenhos

O desempenho desta rotina foi testado comparando-se o tempo de decomposição de uma mesma matriz pela rotina paralela e pela serial.
O código paralelo mostrou-se mais rápido em todos os casos em que a memória RAM foi suficiente. Uma vez excedida a capacidade da RAM, o ler/escrever no disco rígido compromete o resultado, visto que este processo é serial.
Para matriz Skyline cheia, o ganho no desempenho está entre 40% e 100% (algumas vezes além de 100% quando aconteceu efeito de paginação). No entanto, para matriz Skyline com altura de skyline randômica o ganho não é tão expressivo, estando entre 20% e 40%.
O método paralelo sempre executou mais rápido quando comparado ao método serial.

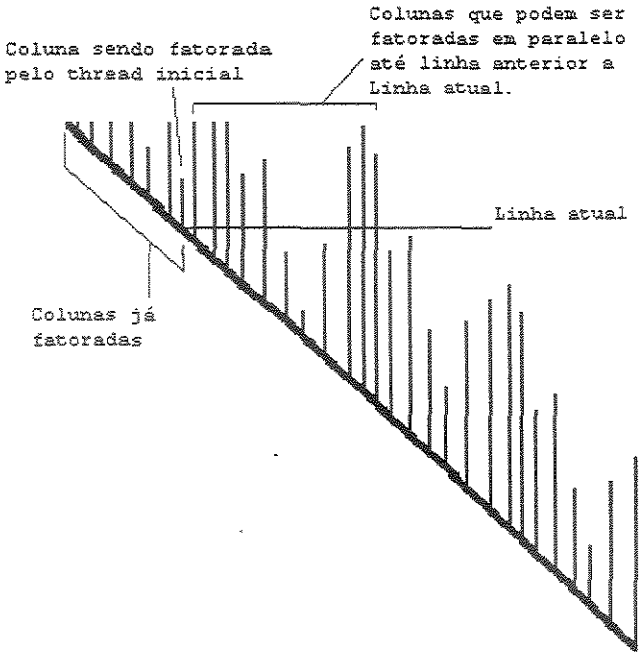


Figura 3.1: Matriz Skyline fatorada em paralelo

Avaliação dos Resultados

A velocidade dos processadores aumenta constantemente. É possível encontrar processadores com velocidade 1.5 GHz (1.500.000.000 de operações por segundo), porém a velocidade com que os dados trafegam entre processador e banco de memória não evoluíram proporcionalmente. O componente crítico esta sendo a taxa de transferência de dados.

Uma rotina será eficiente quando utilizar a memória disponível no cache , minimizando fluxo de dado entre processador e memória, que aparentemente é a fase mais lenta do processo.

Isso é possível se a máquina trabalhar com dados próximos uns dos outros na estrutura de dados. Desta forma, a maioria das operações será realizada com dados armazenados na memória *cache* do processador, cuja velocidade de transmissão é compatível com a velocidade do processamento.

Capítulo 4

Solver Frontal

4.1 Solução Direta de Sistemas de Equações

Diversos tipos de problemas são tratados pelo MEF, cada problema gera matrizes com morfologia particular e para cada tipo de matriz há um método mais adequado. Nos problemas de análise estrutural, a formulação utilizada gera matrizes positiva definidas, simétricas e esparsas.

Os métodos de resolução de sistemas de equações estão divididos em dois grandes grupos: diretos e iterativos (indiretos). O que difere nestes dois grupos é a maneira com que cada um busca a solução do sistema. O método direto necessita da fatoração da matriz dos coeficientes, sendo uma operação muito onerosa. O iterativo gera uma sequência de soluções aproximadas $\left\{x^{(k)}\right\}$, geralmente envolvendo a matriz dos coeficientes em multiplicações [49].

Os métodos iterativos podem ser mais rápidos que os diretos, porém a taxa de convergência depende do tipo de problema solucionado e do pré-condicionador disponível.

Foi implementado, neste projeto, um solver *Cholesky* para matrizes com armazenamento tipo *Skyline* com a fatoração realizada em paralelo. Esta implementação serviu como uma introdução às técnicas de programação paralela.

Depois do solver Skyline paralelo é desenvolvido o solver frontal serial (e.g. não paralelo) e o mesmo em paralelo (ver seção 4.2.2, pag. 38).

4.2 Solver Frontal

Para obtenção da solução do sistema de equações, o código é geralmente elaborado para armazenar a matriz dos coeficientes na memória RAM. Em seguida inicia-se o processo de decomposição desta matriz.

A matriz é formada pela contribuição de cada elemento do domínio aos graus de liberdade do problema. Quanto maior o problema, maior o tamanho desta matriz e consequentemente mais memória RAM será necessária para seu armazenamento. Portanto, um procedimento de montagem (assemblagem) anterior à decomposição pode ser inviabilizado devido ao tamanho da matriz.

Uma maneira mais apropriada seria a montagem parcial da matriz e em seguida a decomposição das equações envolvidas. Isto foi proposto por Irons em 1970 [26] e denominado método Frontal.

O solver frontal utiliza uma matriz de frente onde são realizadas as operações de fatoração das equações. O objetivo inicial do autor era tornar possível simulações que os equipamentos disponíveis na época não eram capazes de realizar, pois a memória RAM era excedida na assemblagem da matriz de rigidez.

Atualmente o método é amplamente reescrito [6], visando não só a montagem parcial, como também uma paralelização das operações em máquinas com arquiteturas paralelas. A matriz de rigidez, originalmente esparsa, é tratada agora (na matriz frontal) como densa, possibilitando a utilização de rotinas otimizadas/paralelas para operações com matrizes densas (*BLAS*).

4.2.1 Funcionamento do Método

O método frontal é muito utilizado por várias áreas do conhecimento e cada uma delas implementa as variações que melhor se adaptam aos seus problemas específicos.

Segundo Camarda [6], o esquema da eliminação frontal pode ser resumido como segue:

1. Montar uma linha na matriz frontal
2. Determinar se alguma coluna está totalmente somada na matriz frontal. Uma coluna estará totalmente somada, uma vez que todos seus elementos não zeros estiverem na matriz frontal.
3. Fatorar, no caso de colunas totalmente somadas, os elementos das linhas e colunas .
4. Realizar uma atualização com um produto externo entre a parte fatorada e o restante da matriz frontal.

Este procedimento se inicia com a montagem da linha 1 na inicialmente vazia matriz frontal, e segue linha por linha até que todas tenham sido fatoradas, completando então a fatoração por Cholesky.

Em termos matemáticos, considere a matriz em banda A:

$$A = \begin{bmatrix} D & A & 0 & 0 & 0 & 0 \\ A & D & A & A & 0 & 0 \\ 0 & A & D & A & 0 & 0 \\ 0 & A & A & D & A & 0 \\ 0 & 0 & 0 & A & D & A \\ 0 & 0 & 0 & 0 & A & D \end{bmatrix}$$

Pense numa matriz frontal F como uma "janela" que percorre a matriz A pela sua diagonal. Num primeiro passo a matriz F será:

$$A = \begin{bmatrix} F & F & 0 & 0 & 0 & 0 \\ F & F & A & A & 0 & 0 \\ 0 & A & D & A & 0 & 0 \\ 0 & A & A & D & A & 0 \\ 0 & 0 & 0 & A & D & A \\ 0 & 0 & 0 & 0 & A & D \end{bmatrix}$$

Alguns passos depois a matriz F será:

$$A = \begin{bmatrix} d & a & 0 & 0 & 0 & 0 \\ a & F & F & F & 0 & 0 \\ 0 & F & F & F & 0 & 0 \\ 0 & F & F & F & A & 0 \\ 0 & 0 & 0 & A & D & A \\ 0 & 0 & 0 & 0 & A & D \end{bmatrix}$$

A dimensão da matriz F é a igual a largura da frente (frontwidth ver item A, p. 68) da equação k na matriz A.
As sub-matrizes representadas com letras minúsculas são os elementos da matriz A já fatorados.
A decomposição é realizada sobre a matriz F que receberá os elementos da matriz global A.
Estando a matriz F montada, inicia-se a decomposição da mesma, que pode ser realizada utilizando qualquer método conhecido.

1. Fatoração da matriz F por LU
   Para a coluna c e linha r procede a fatoração de acordo com:

$$r_i = a_{ki}$$
$$r_k = 1$$
$$c_i = a_{ik}/a_{kk}$$
$$c_k = a_{kk}$$

   então:

$$\widetilde{A}_{ij} = A_{ij} - c_i r_j$$

2. Fatoração de F por Cholesky

$$r_i = a_{ki}/\sqrt{a_{kk}}$$
$$r_k = \sqrt{a_{kk}}$$
$$c_i = a_{ik}/\sqrt{a_{ik}}$$
$$c_k = \sqrt{a_{kk}}$$

   então:

$$\widetilde{A}_{ij} = A_{ij} - c_i r_j$$

3. Fatoração de F por LDL^T

$$r_i = a_{ki}$$

$$r_k = a_{kk}$$

$$c_i = a_{ik}$$

$$c_k = a_{kk}$$

então:

$$\tilde{A}_{ij} = A_{ij} - c_i r_j / d_k$$

A matriz F portanto, terá dimensão variável que dependerá da largura da frente. Enquanto a "janela" da matriz F percorre a diagonal, a largura da frente da equação k ditará a dimensão da matriz F.

Nota-se que a ordem em que as variáveis são eliminadas depende da ordenação das equações. Um tratamento na ordenação das equações é portanto, importante para a performance do método. Uma seção específica (Apêndice A pag. 68) tratará da questão de reordenação.

4.2.2 Funcionamento em Paralelo

O processo frontal em paralelo inicia-se com a criação de no mínimo 3 *threads* (tarefas) independentes. Um deles é responsável pela adição de matrizes elementares à matriz frontal, bem como a decomposição das equações totalmente adicionadas, o método *GlobalAssemble*(). Um segundo processo é responsável pela gravação em disco, das equações já decompostas, o método *WriteFile*(). Aos threads restantes é delegada a tarefa de montagem das matrizes elementares, o método *ElementAssemble*().

Todos os threads são disparados pelo mesmo método da classes estrutural TPZParFrontStructMatrix, o método *CreateAssemble*(). O método faz uma chamada de *pthread_create*() para cada um dos métodos responsáveis citados. Os threads são disparados e o *thread* corrente é forçado à esperar os outros pela chamada feita a *pthread_join*(). A implementação do método *CreateAssemble*() pode ser encontrada na documentação do solver frontal em anexo.

Dois grupos de dados são definidos na classe TPZParFrontMatrix. Um deles armazena informações sobre as matrizes elementares montadas, é uma estrutura formada por três objetos tipos TPZStack (que implementam pilhas), armazenam as matrizes elementares e informações de cronologia e sequência de contribuição na matriz frontal. O segundo grupo de dados diz respeito às equações decompostas, é formado por um objeto do tipo TPZStack e contém objetos do tipo TPZEqnArray decompostos.

As pilhas relacionadas com as matrizes elementares são compartilhadas pelos métodos *GlobalAssemble* e *ElementAssemble*, a pilha de TPZEqnArrays é por sua vez compartilhada pelos métodos *GlobalAssemble* e *WriteFile*, visto este compartilhamento de dados, faz-se necessária a existência de dispositivos de sincronização para mantenimento da integridade dos dados. Os dispositivos necessários são então definidos por:

- *pthread_mutex_t mutex_element_assemble.*
  No método *ElementAssemble*(), visto que mais de um processo pode estar executando esta função, é necessário um dispositivo do tipo *mutex* para garantir que apenas um processo por vez, decremente a quantidade de elementos a serem computados. Isso é feito realizando-se um lock sobre o objeto *mutex_element_assemble.*

- *pthread_mutex_t mutex_global_assemble.*
  Nos método *GlobalAssemble*() e *ElementAssemble*(), é necessário um dispositivo do tipo *mutex* para garantir que apenas um dos processos acesse, num mesmo instante, as pilhas referentes às matrizes elementares. Isso é feito com uma chamada de lock sobre o objeto *mutex_global_assemble.*
  O método *ElementAssemble*() realiza o lock e adiciona elementos às pilhas, enquanto que o método *GlobalAssemble*() realiza o lock para retirar elementos das pilhas.

- *pthread_cond_t stackfull.*
  Para evitar um acúmulo de matrizes elementares montadas na memória RAM, define-se uma variável especificando o tamanho máximo desta pilha. Os processos de montagem de matrizes, checam se este valor foi atingido, caso sim, realizam uma chamada à *cond_wait*() (suspensão), esperando pela sinalização da variável *stackfull.* O método que realizará esta sinalização será o *GlobalAssemble*() após computar uma matriz elementar na matriz frontal.

- pthread_cond_t *condassemble.*
  No processo frontal, define-se uma ordem de contribuições de matrizes elementares à matriz frontal. Quando existem matrizes elementares formadas mas não a da sequência requerida, uma chamada a *cond_wait*(), esperando a sinalização da variável *condassemble* é realizada. O método que realizará esta sinalização será o *ElementAssemble*() após computar a requerida matriz.

- *pthread_mutex_t fwritelock.*
  Os processos de decomposição das equações e escritura em disco dos coeficientes decompostos trabalham em *threads* independentes. Define-se então uma pilha de equações decompostas que é acessada pelos dois processos citados. Garante-se o acesso não concorrente com uma chamada de lock sobre o objeto *fwritelock.* O método *GlobalAssemble*(), após uma chamada à DecomposeEquations() realiza o lock e adiciona elementos à pilha de equações. Já o método *WriteFile*() realiza o lock e retira termos da pilha escrevendo-os em disco.

- *pthread_cond_t fwritecond.*
  No método *WriteFile*(), após a realização de um lock, checa-se à existência de elementos na pilha de equações. Quando da não existência destes, realiza-se uma chamada a *cond_wait*(), esperando pela sinalização da variável *fwritecond.* O método que realizará esta sinalização será o *GlobalAssemble*() após adicionar uma equação decomposta à pilha.

A Figura 4.1 (pag 40) representa com diagramas os três processos independentes e suas intercomunicações.

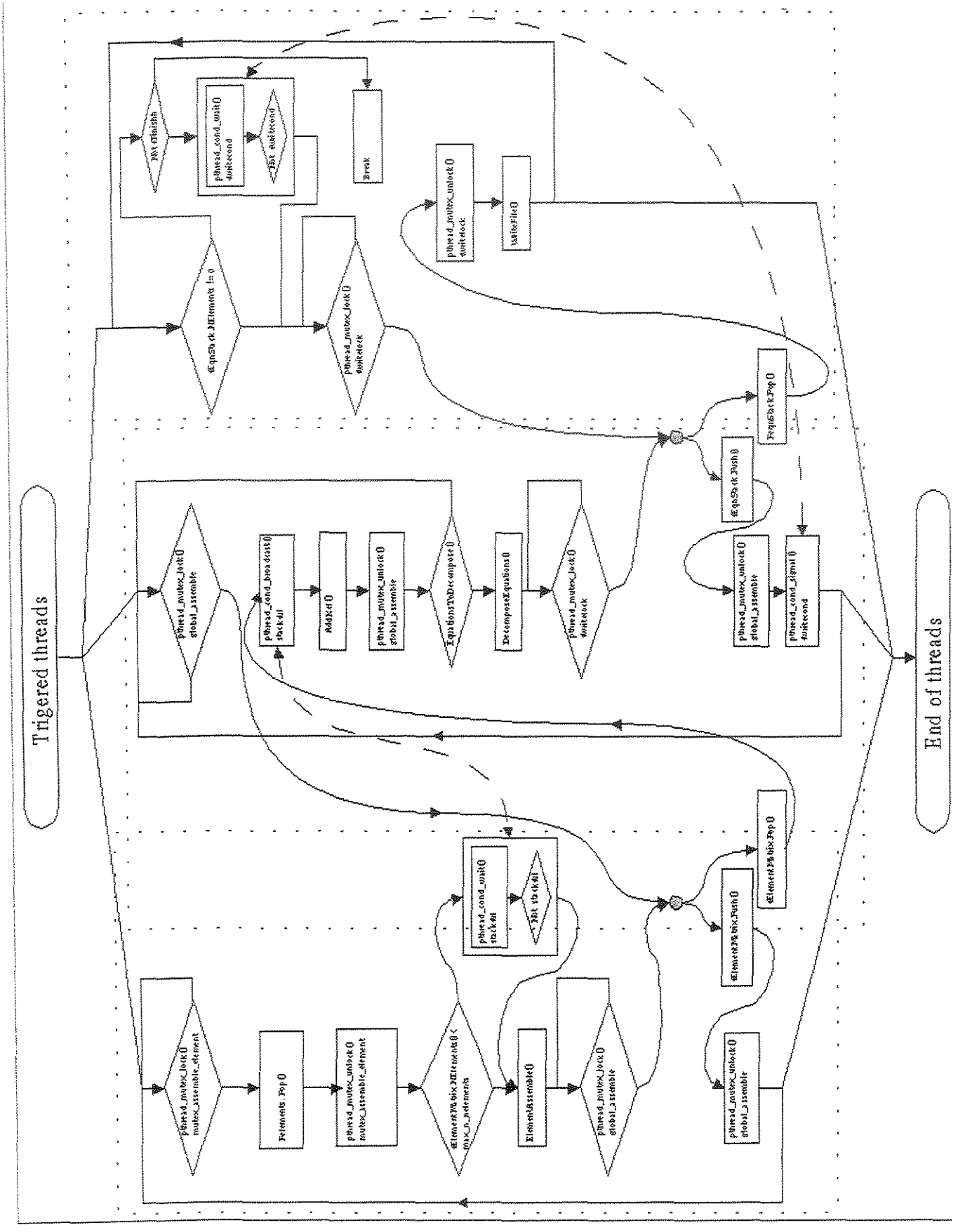


Figura 4.1: Processos independentes em paralelo

Implementação Paralela do Método

Sabe-se que escrita e leitura do disco consome tempo de interface (*buffering*), mas pouco tempo de CPU. A leitura/escrita será então executada numa tarefa (*thread*) exclusiva. Em paralelo ocorrem a montagem da matriz do elemento e a decomposição da matriz frontal. Cada um dos processos deve ser executado numa tarefa (*thread*) independente.

O processo de montagem das matrizes elementares pode ocorrer em um ou mais de um *thread*. Visto que os dados necessários para montagem de diferentes matrizes elementares não são dados compartilhados, executa-se a montagem de mais de uma matriz elementar ao mesmo tempo, cada qual em seu respectivo processo.

Paralelamente a formação das matrizes elementares ocorre a decomposição das equações adicionadas na matriz frontal. Conforme as equações são decompostas, estas são armazenadas nos vetores de equações decompostas (Concebidos na implementação pela classe TPZEqnArrays 5.1.1 pag 47).

Juntamente com os dois processos citados, as equações decompostas e armazenadas em EqnStorage são gravadas em disco. À medida que equações são gravadas em disco, espaço é liberado em EqnStorage para receber mais equações. A Figura 4.2 (pag 42) representa o fluxo dos dados pelo processo.

Montagem das Matrizes Elementares

Os dados necessários para montagem das matrizes elementares apresentam-se de forma bem granular. Isso possibilita a execução em paralelo de tantas quantas forem necessárias montagens de matrizes de rigidez elementares.

A integração dos métodos de resolução ao pacote PZ é realizada pelas classes TPZStructMatrix. Estas classes implementam a interface entre as classes de elementos finitos genéricas e as classes matriciais.

Na implementação em paralelo, dispara-se um processo destinado a montagem de matrizes elementares. Uma sequência de montagem deve ser obedecida de maneira a otimizar a dimensão da matriz frontal. Aplica-se sobre a malha computacional o método OrderElement e a sequência é definida.

Com a sequência definida, o processo dedicado a montagem de matrizes elementares é disparado. Este será concluído quando todas as matrizes de todos elementos estiverem montadas e armazenadas na memória RAM. Como uma das virtudes do método frontal é não consumir os recursos de memória RAM da máquina, a quantidade de matrizes elementares montadas deve ser controlada, protegendo assim a característica do método frontal.

Defini-se uma pilha de matrizes elementares cuja dimensão não deve exceder um determinado parâmetro. Quando este valor é alcançado o processo dedicado à montagem recebe uma chamada *cond_wait* e aguarda até que um outro processo dedicado à adição de matrizes elementares à matriz frontal esvazie a pilha de matrizes. A medida que matrizes são adicionadas a matriz frontal (e consequentemente retiradas da pilha de matrizes elementares), novas matrizes elementares podem ser computadas.

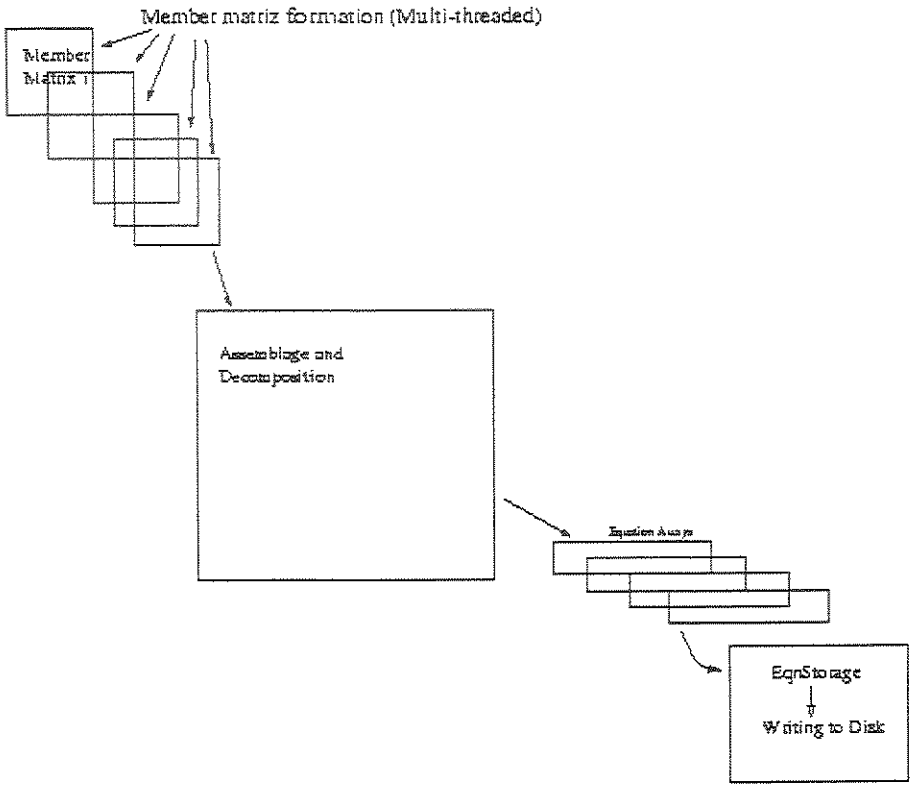


Figura 4.2: Processo paralelo

Adição de Matrizes Elementares

Pela concepção do método frontal, uma equação totalmente adicionada à matriz frontal pode ser decomposta. Uma equação será totalmente adicionada quando todos elementos vinculados à equação já tiverem contribuído com sua matriz de rigidez. A cada adição de matriz elementar realizada verifica-se a possibilidade de decomposição de alguma equação.

O processo de adição de matrizes funciona paralelamente ao processo de montagem de matrizes elementares. Utiliza-se facilidades de multi-threading para comunicação entre os dois processos pois ambos acessam a mesma estrutura de dados, a pilha de matrizes elementares montadas. O processo de adição à matriz frontal retira elementos (matrizes) da pilha somando-as na matriz frontal, enquanto que o processo de montagem computa matrizes elementares e as carrega na pilha de matrizes elementares.

Percebe-se portanto que o método vincula a adição de matrizes elementares ao processo de decomposição das equações. No final de processo de adição de matrizes elementares, quando todas as matrizes tiverem sido somadas à matriz frontal, o sistema já estará decomposto.

Armazenamento das Equações Decompostas

Para um bom rendimento do método frontal, deve-se minimizar ao máximo a troca de dados entre processador(es) e bancos de memória RAM. Optou-se na atual implementação por escrever em disco, em forma de arquivos binários, as equações decompostas.

Como para os processos acima, define-se um thread exclusivo para esta tarefa. Define-se também uma pilha de equações decompostas, estas equações são armazenadas em formato esparso (classe TPZEqnArray 5.1.1 pag 47), a pilha contém portanto objetos do tipo TPZEqnArray. Sobre este arquivo binário com os elementos das equações já em seu estado decomposto, serão aplicadas as substituições *Backward* e *Forward* para resolução do sistema de equações.

Dois processos atuam sobre esta pilha. O primeiro deles é o processo de montagem/decomposição da matriz frontal que adiciona à pilha equações decompostas. O segundo processo é o de armazenamento em disco das equações decompostas. Equações decompostas presentes na pilha são gravadas em disco em forma de arquivo binário. Dispositivos multi-threading gerenciam a comunicação entre este processo e o de montagem/decomposição no acesso aos dados da pilha de equações.

A classe responsável pelo armazenamento dos dados gerencia eficientemente os arquivos binários gerados. Não cabe ao usuário definir o nome do arquivo, a classe TPZFileEqnStorage utiliza a função *mktemp(InitialFileName)* do "C" que gera nomes únicos para arquivos. A função recebe uma string de no mínimo 6 caracteres contendo no final 6 letras "X" que serão alterados pela função e conterão um código único no sistema de arquivos. Nos caracteres iniciais qualquer combinação é aceita.

Isso possibilita futuras implementações de um solver multi-grid multi-frontal, onde várias decomposições serão executadas em paralelo. Ocorrendo a geração de vários arquivos binários simultaneamente.

Estrutura de Armazenamento da Matriz Frontal

Visto que estamos trabalhando com métodos de resolução diretos, a matriz frontal prevê uma estrutura de acesso a uma matriz cheia, porém o armazenamento é feito na forma de um vetor, armazenando coluna por coluna. Optou-se por esta estrutura visando facilitar a aplicação de rotinas BLAS na matriz frontal. A matriz pode ainda ser simétrica ou não.

Implementa-se então duas classes TPZFrontSym e TPZFrontNonSym responsáveis pelo armazenamento simétrico e não simétrico respectivamente. Nas duas implementações aplica-se rotinas BLAS na fatoração das equações. Na versão simétrica utiliza-se o método dspr_(...) que, conforme [13], executa a operação descrita abaixo denominada "Rank n Update" sobre uma matriz simétrica. A chamada completa da função encontra-se no algoritmo 1pag 44.

Algorithm 1 Chamada dsysrk da BLAS.

dsyrk(uplo, trans, n, k, α, a, lda, β, c, ldc)

$$c = \alpha \times a \times' + \beta \times c$$

ou

$$c = \alpha \times a' \times a + \beta \times c$$

onde:

α e β são escalares.

c é uma matriz n por n

a é uma matriz n por n no primeiro caso e k por n no segundo caso.

Os parâmetros de entrada uplo, trans, lda e ldc estão descritos na página 2-99 do manual [13] e dizem respeito a morfologia de armazenamento.

Na versão não simétrica utiliza-se o método dger_(...) que, conforme [13], executa a operação descrita abaixo denominada "Rank n Update" sobre uma matriz genérica.

Algorithm 2 Chamada dger da BLAS.

dger(m, n, α, x, incx, y, incy, a, lda)

$$a = \alpha \times x \times y' + \alpha$$

onde:

α é um escalar.

x é um vetor de m elementos

y é um vetor de n elementos

a é uma matriz n por n.

Os parâmetros de entrada m, n e x estão descritos na página 2-28 do manual [13] e dizem respeito a morfologia de armazenamento.

Número de Equações	t(s) Solver Skyline	t(s) Solver Frontal
4163	31	20
9316	125	180
16517	440	727
25766	1511	1949
37063	36726	4707

Tabela 4.1: Tempo (segundos) X Número de Equações dos Solvers Skyline e Frontal.

Métodos de Fatoração da Matriz Frontal

A implementação prevê a utilização do solver frontal em problemas não simétricos. Implementa-se então o método LU para fatoração de matrizes não simétricas. Para matrizes simétricas utiliza-se o método de Cholesky.

Desempenhos

Compara-se o desempenho do método frontal, obtendo-se a solução do mesmo sistemas de equações ora com o solver Skyline, ora com o solver frontal. Nos processos onde o solver Skyline suportou a carga de dados necessária, ou seja, quando a memória RAM foi suficiente para armazenar os dados, este obteve um melhor desempenho que o solver frontal.

Uma vez que a quantidade de dados excede o espaço disponível na memória RAM, a vantagem do solver Frontal fica clara. Nestas circunstâncias fica impossível de se comparar os desempenhos pois o tempo gasto pelo solver Skyline não é mais um dado significativo. A tabela 4.1, pag 45, apresenta os tempos gastos em segundos pelos solvers Skyline e Frontal para os listados número de equações.

Avaliação dos Resultados

Como previsto, um dos pontos diretamente ligados à eficiência de um processo de cálculo, diz respeito ao volume de tráfego de dados entre processador e banco de memória, a relação entre fluxo de dado e eficiência do código é uma relação inversamente proporcional, quanto menor este fluxo mais eficiente o código será.

Em solvers diretos nota-se uma quantidade elevada de tráfego de dados na operação de fatoração das equações. Uma matriz que inicialmente apresenta-se de forma esparsa, após a decomposição de uma equação com respeito às demais, preenche as anteriormente vazias posições na matriz sendo decomposta. Isso penaliza os métodos diretos em termos de eficiência de armazenamento de dados e performance, visto que os dados sendo utilizados durante este processo encontram-se geralmente na memória RAM.

As simples constatações acima sobre solvers diretos fazem do processo frontal um método diferenciado. No processo frontal a matriz manipulada durante a fatoração das equações possui tamanho reduzido, diminuindo assim o tráfego de dados entre processador e memória RAM. O dano causado pelo preenchimento de valores anteriormente nulos na matriz dos coeficientes é superado pela possibilidade de se utilizar rotinas BLAS na fatoração das equações, visto que a dimensão da matriz sendo manipulada pelo processo é de dimensão reduzida.

Pode-se contestar a eficiência de performance do método para um numero reduzido de equações, porém não é objetivo do solver Frontal tratar de problemas de pequena escala. Os testes demonstram que para um intervalo de número de equações entre 4000 e 33000 o solver Skyline é mais eficiente. O limite superior deste intervalo está diretamente ligado com a quantidade de memória RAM disponível no equipamento. Para estes casos foi utilizada uma máquina com 512 MB de memória RAM. Para valores em torno 30000 a 35000 a paginação necessária compromete os resultados do solver Skyline.

Estas particularidades das performances de cada um dos métodos, em diferentes intervalos de número de equações, era esperado. O método frontal exige uma série de rotinas para administração dos dados, matrizes, arquivos binários e etc, o que resulta num "overhead" administrativo, porém trabalha com matrizes esparsas. Já o solver Skyline, trabalha diretamente com operações vetoriais.

No primeiro intervalo de número de equações entre 4000 e 30000 o solver skyline mostra-se vantajoso conforme os resultados na tabela 4.1 pag 45.

Para sistemas com quantidade de graus de liberdade acima de 30000 não há a possibilidade de se utilizar um outro processo que não seja o frontal. A simples alocação de memória para armazenamento da matriz global comprometeria todos os recursos de RAM da máquina, forçando assim o sistema a executar *swaps* nos discos rígidos, acabando com qualquer tentativa de melhora de performance.

Com os resultados obtidos nos testes e com o a experiência adquirida durante o desenvolvimento do projeto, comprova-se a eficiência do método frontal. Não resta dúvida dos benefícios de uma estrutura que não comprometa os recursos da máquina. Quando da utilização de outros solvers, nota-se o comprometimento da máquina para concluir a operação de resolução do sistema de equações. Utilizando o solver frontal, o equipamento torna-se apenas um pouco mais lento, porém não comprometido.

Capítulo 5

Implementações Orientadas para Objeto

5.1 Classes Desenvolvidas para Solver Frontal

Na fase final do projeto trata-se o solver frontal sob uma abordagem orientada para objetos. Definem-se cinco classes bases para o completo tratamento do problema. Destas cinco classes uma delas é abstrata, sendo portanto redefinida em duas sub-classes. Sub-tópicos são dedicados a cada uma das classes. A Figura 5.1 (pag 48) representa a estrutura de classes para o solver frontal.

5.1.1 Classe TPZEqnArray

A classe TPZEqnArray é responsável pelo armazenamento dos coeficientes das equações do sistema sendo resolvido. Geralmente, a equação armazenada encontra-se em estado decomposto.

Utiliza-se o esquema de armazenamento esparso para a estrutura de dados. Visto que esta classe é responsável pelo armazenamento de dados decompostos, a responsabilidade algébrica da classe está nas operações com valores decompostos. Esta classe é responsável então pelos métodos forward e backward (Algoritmo 3 pag 48 e Algoritmo 4 pag 49).

EqnArray irá conter parte desta matriz representada na tabela 5.1 (pag 47).

Os métodos *Forward* (Algoritmo 3 pag 48) e *Backward* (Algoritmo 4 pag 49) para decomposição LU:

Armazenamento Esparso

Utiliza-se a técnica de elementos finitos, para obtenção de aproximações numéricas dos mais variados tipos: problemas de estática, dinâmica, propagação de choques e etc. Em casos particulares, as matrizes de rigidez apresentam uma morfologia característica, estas são simétricas e

Tabela 5.1: $\begin{cases} a_{i_0 k}, k \text{ para qualquer } k \text{ tal que } i_0 \leq k < n \text{ e } a_{i_0 k} \neq 0 \\ \vdots \\ a_{i_n k}, k \text{ para qualquer } k \text{ tal que } i_n \leq k < n \text{ e } a_{i_n k} \neq 0 \end{cases}$

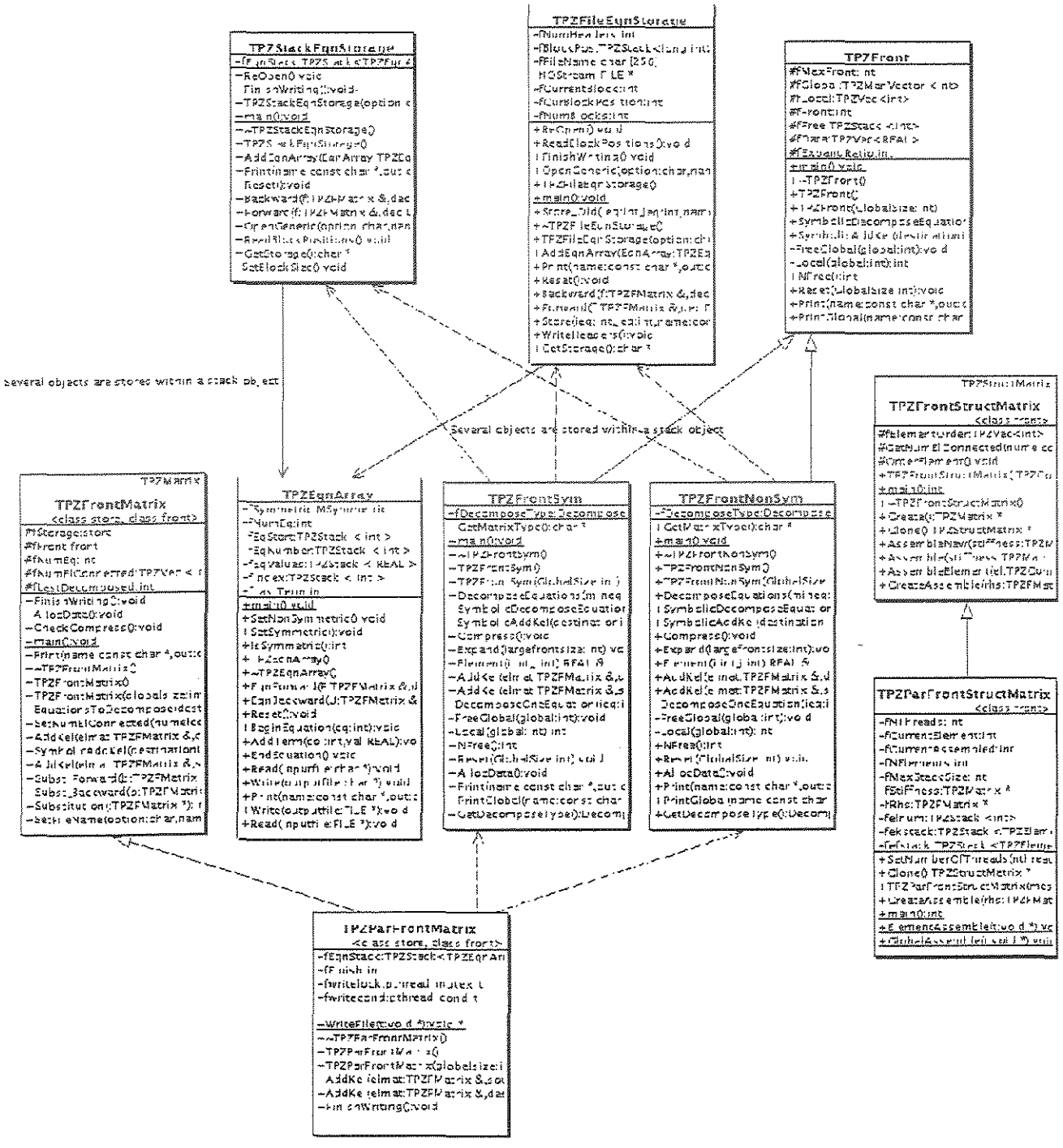


Figura 5.1: Classes para o solver frontal

Algorithm 3 Forward

for $i = 0,\ n$

  for $j = 0,\ n$

    for $k = j + 1,\ n$

      $F_{k,i} = F_{k,i} - a_{i,k} \cdot F_{k,i}$

Algorithm 4 Backward

for $i = 0,\ n$
  for $j = n - 1,\ 0$
    for $k = j + 1,\ n$
      $F_{j,i} = F_{j,i} - a_{j,k} \cdot F_{k,i}$
    $F_{j,i} = F_{j,i}/F_{j,j}$

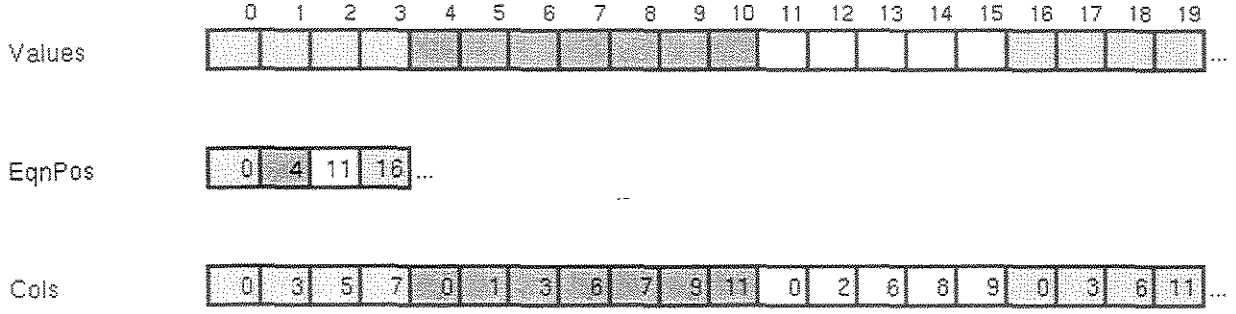


Figura 5.2: Estrutura esparsa de armazenamento

esparsas, ou seja, os elementos a_{ij} de uma matriz A são iguais aos elementos a_{ji} para $i \neq j$ e também a quantidade de elementos nulos nestas matrizes supera a quantidade de elementos não nulos.

Buscando otimizar o uso de recursos dos equipamentos no armazenamento destas matrizes, utiliza-se para um sistema com n equações não uma matriz padrão A_{nxn} mas sim uma estrutura formada por três vetores. Com a seguinte estrutura e relacionamento:

- Um vetor tipo *double* $Values$ armazena os termos não nulos da matriz de rigidez.
- Um vetor tipo *int* $EqnPos$ armazena a posição no vetor $Values$ para o início de cada equação.
- Um terceiro vetor tipo *int* $Cols$ armazena para cada posição i no vetor $Values$ a coluna de cada valor.

Os vetores $Values$ e $Cols$ possuem o mesmo tamanho q. Sendo esta a quantidade de valores não nulos na matriz de rigidez. A Figura 5.2 (pag. 49) representa esquematicamente a estrutura.

Estrutura de Dados

1. int fNumEq
   Número de equações no sistema.
2. int fLastTerm
   Último termo da equação inserida

3. TPZManVector<int> fEqStart
   Índice do início de cada equação

4. TPZStack<REAL> fEqValues
   Valores dos coeficientes da equação

5. TPZStack<int> fIndexes
   Número da linha coluna associada a cada valor

Métodos para TPZEqnArray

Os métodos de TPZEqnArray são :

1. void *Reset*()
   Reinicializa a estrutura de dados do objeto.

2. void *BeginEquation*(int eq)
   Inicia o processo de adição de termos para a equação eq

3. void *AddTerm*(int col, REAL val)
   Adiciona o valor val localizado na coluna col no vetor de equações

4. void *EndEquation*()
   Termina o processo de adição de termos para uma equação

5. void *EqnForward*(MDecomposeType dec, TPZFMatrix &F)
   Realiza uma substituição forward sobre o argumento F com a equação decomposta.

6. void *EqnBackward*(MDecomposeType dec, TPZFMatrix &U)
   Realiza uma substituição backward sobre o argumento U com a equação decomposta.

7. void *Print* (const char * name, ostream & out)
   Imprime os dados de TPZEqnArray. O parâmetro *name* contem o título a ser impresso no inicio do arquivo. O parâmetro *out* é o objeto tipo *ostream* de saída de dados.

8. void *Read* (FILE * inputfile)
   Realiza leitura de dados para TPZEqnArray obtendo os dados do arquivo *inputfile*.

Os procedimentos *EqnForward* e *EqnBackward* dependem do método utilizado para fatoração das equações. O tipo de decomposição é especificado pela variável *dec* passada como parâmetro.

Testes de Validação

TPZEqnArray contém em sua estrutura de dados, termos de uma equação em sua forma decomposta. Para a realização dos testes de validação, utilizou-se as classes matriciais já operacionais e validadas do ambiente PZ.

Visto que até o momento, implementou-se as rotinas *forward* e *backward* para decomposição por Cholesky, uma matriz cheia foi gerada automaticamente e em seguida decomposta.

Adicionou-se seus termos a um objeto tipo TPZEqnArray e sobre eles executou-se as substituições *forward* e *backward*. Sobre a matriz teste original executou-se também a duas substituições e os resultados obtidos com *EqnArray* são comprovados pelos obtidos com a matriz teste.

Com estes testes valida-se não só as rotinas citadas, mas também toda estrutura de dados e métodos envolvidos no seu gerenciamento.

5.1.2 Classe TPZFront

A classe TPZFront é a abstração da classe responsável pelo armazenamento e decomposição da matriz frontal. Desta classe foram herdadas duas classes filhas com o mesmo propósito porém com particularidades:

- A classe TPZFrontSym executa suas funções sob uma estrutura simétrica. Possui métodos de decomposição e armazenamento que contemplam a característica de simetria.
- A classe TPZFrontNonSym possui as mesmas responsabilidades e as executa sob uma estrutura não simétrica. Os métodos de decomposição e armazenamento contemplam a característica de não simetria.

TPZFront recebe a contribuição dos elementos no domínio tratado. Dentro do procedimento de decomposição frontal, as equações são decompostas a medida que são totalmente adicionadas à matriz frontal. Uma equação é totalmente adicionada quando todos os elementos associados ao grau de liberdade contribuírem à matriz de rigidez. A classe TPZFront é responsável por decompor estas equações uma a uma.

A informação da totalidade ou não da adição de uma equação, é obtido percorrendo-se inicialmente todos os elementos e montando com isso uma lista de incidências por grau de liberdade. Computa-se a quantidade de elementos que contribuêm para o $n$ésimo grau de liberdade. Esta lista é então usada para determinação de graus de liberdade totalmente adicionados. Sabe-se que o custo computacional desta tarefa é relativamente baixo, por tratar-se de operações com valores inteiros.

As equações são adicionadas e decompostas de forma dinâmica. Isto implica que a equação n não encontra-se necessariamente na posição n na matriz frontal. Necessita-se da indexação das equações entre as matrizes global e frontal. A estrutura de dados da classe permite a inversão de uma equação global em equação local de maneira eficiente.

Estrutura de Dados

1. int *fMaxFront*
   Tamanho máximo que a frente irá atingir
2. TPZVec<int> *fGlobal*
   Equação global associada a cada equação da frente.
3. TPZVec<int> *fLocal*
   Equação na matriz frontal para cada equação global.

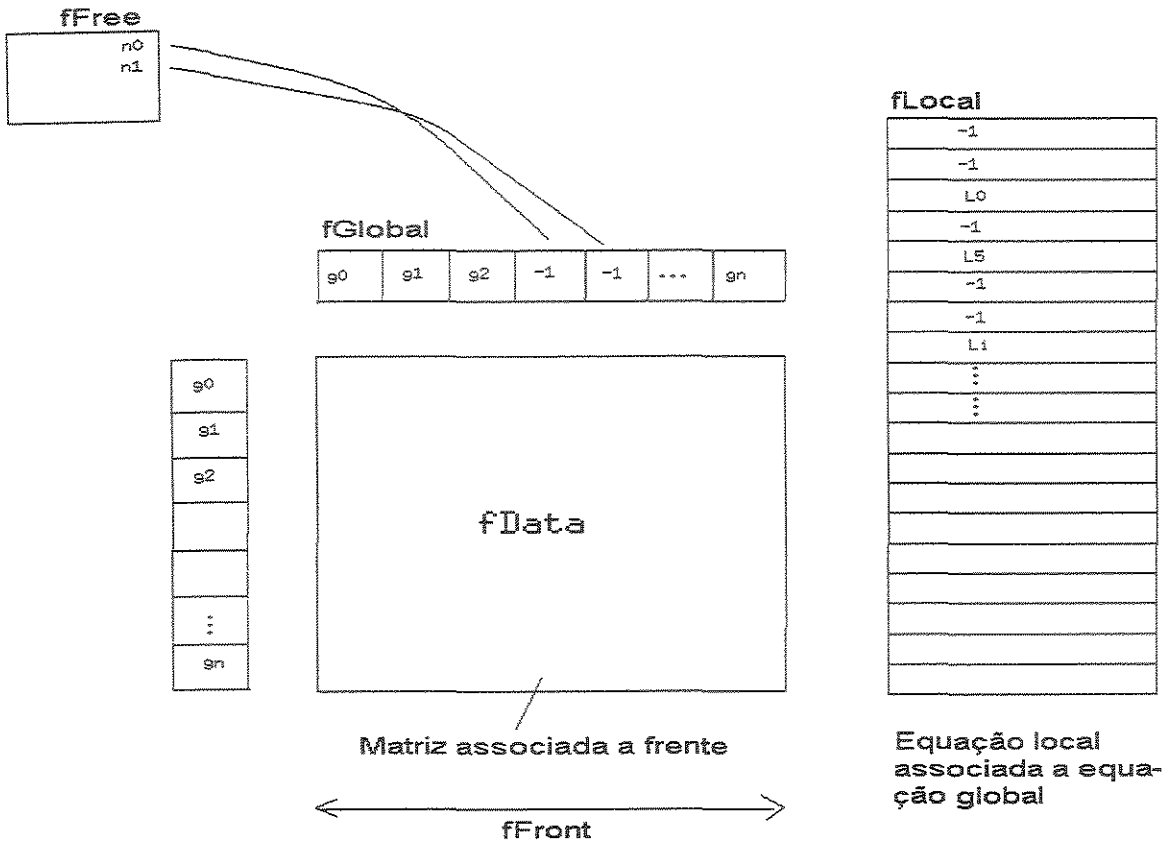


Figura 5.3: Esquema de armazenamento na matriz frontal.

4. int *fFront*
   Tamanho da frente utilizada.

5. TPZStack<int> *fFree*
   Linhas/colunas na frente sem utilização.

6. TPZVec<REAL> *fData*
   Dados da matriz frontal.

A Figura 5.3 (pag 52) ilustra o esquema de armazenamento e gerenciamento dos dados na matriz frontal.

Métodos

Os métodos associados a TPZFront são:

1. void *DecomposeEquations*(int *mineq*, int *maxeq*, TPZEqnArray & *result*)
   Decompõe o conjunto de equações enumeradas de mineq até maxeq e atribui o resultado a *result*.

2. void *Compress*()
   Comprime a estrutura de dados. (Reutiliza a linhas/colunas indicadas por *fFree*).

3. void *FreeGlobal*()
   Atribui a equação global como liberada. O espaço usado por esta equação é agora disponível para próximas equações.

4. void *SymbolicDecomposeEquations*(int mineq , int maxeq)
   Decompõe as equações de maneira simbólica e armazena os índices liberados em *fFree*

5. void *SymbolicAddKel*(*TPZVec* < *int* > & *destinationindex*)
   Adiciona simbolicamente a contribuição de um elemento à equação.

Testes para Validação

Os testes para validação desta classe envolvem as seguintes tarefas

- Adição de elementos a matriz
- Decomposição de uma equação quando de sua total adição a matriz global.

A correta adição dos membros das equações à matriz do tipo TPZFront é comprovada através de posteriores chamadas das rotinas de tradução da indexação local para global e vice e versa.

Comprova-se a decomposição de uma determinada equação, com resultados obtidos na decomposição de uma matriz teste, que é de algum tipo anteriormente comprovado no ambiente PZ.

5.1.3 Classe TPZFrontSym

Possui as mesmas atribuições que classe base TPZFront, conforme item 5.1.2 (pag. 51). Especializa-se TPZFront para contemplar matrizes simétricas.

Talvez a mais comum otimização em estruturas de dados seja a relacionada a simetria de matrizes, onde são armazenados a diagonal e a parte inferior ou superior a diagonal. Além das melhorias com respeito a recursos de armazenamento, métodos de decomposição específicos para matrizes simétricas são mais eficientes que os convencionais, pois realizam uma quantidade inferior de operações de ponto flutuantes.

Especializa-se então TPZFront em TPZFrontSym para utilizar as vantagens inerentes às matrizes simétricas. Tanto estrutura de dados quanto os métodos são os mesmos que em TPZFront, sendo agora estes redefinidos para uma estrutura simétrica. Para decomposição da matriz frontal utiliza-se fatoração por Cholesky.

Métodos para TPZFrontSym

Especializa-se alguns métodos de TPZFront com o objetivo de contemplar a morfologia da nova matriz concebida em TPZFrontSym.

1. void *PrintGlobal(const char *name, ostream& out)*
   Imprime a estrutura global da matriz frontal

2. void *AllocData()*
   Viabiliza espaços na estrutura de dados.

3. void *Reset(int GlobalSize)*
   Reinicializa a estrutura de dados.

4. void *DecomposeOneEquation(int ieq, TPZEqnArray &eqnarray)*
   Decompõe a equação *ieq*.

5. void *AddKel(TPZFMatrix &elmat, TPZVec<int> &sourceindex, TPZVec<int> &destinationindex)*
   Adiciona uma matriz de rigidez elementar à matriz frontal. Contém dois vetores contendo índices de origem e de destino.

6. void *AddKel(TPZFMatrix &elmat, TPZVec<int> &destinationindex)*
   Adiciona uma matriz de rigidez elementar à matriz frontal, passa-se como parâmetro apenas os índices de destino.

7. void *Expand(int larger)*
   Expande o tamanho da frente.

8. void *Compress()*
   Comprime o tamanho da frente.

9. void *SymbolicAddKel(TPZVec < int > & destinationindex)*
   Executa uma adição simbólica das matrizes elementares. Tem por objetivo predefinir o tamanho da frente.

10. void *SymbolicDecomposeEquations(int mineq, int maxeq)*
    Decompõe simbolicamente as equações.

11. void *DecomposeEquations(int mineq, int maxeq, TPZEqnArray & eqnarray)*
    Decompõe as equações de *mineq* até *maxeq* armazenando o resultado em *eqnarray*.

5.1.4 Classe TPZFrontNonSym

Possui as mesmas atribuições que a classe base TPZFront, conforme item 5.1.2 (pag. 51). Especializa-se TPZFront para contemplar matrizes simétricas.

Tanto estrutura de dados quanto os métodos são os mesmos que em TPZFront, sendo agora estes redefinidos para uma estrutura não simétrica. Para decomposição da matriz frontal utiliza-se fatoração por LU.

5.1.5 Classe TPZFrontMatrix

Uma das principais características do solver frontal, é a não montagem da matriz de rigidez global da malha de elementos finitos. Visto isso, há a necessidade de uma entidade que administre o fornecimento dos dados para a matriz frontal

Define-se então a classe TPZFrontMatrix que é responsável pela coordenação da montagem/decomposi da matriz . Ao contrário de se executar a montagem global da matriz de rigidez e depois a decomposição, como ocorreria num processo convencional, esta classe possui um método que identifica as equações que podem ser decompostas.

As equações são adicionadas através do método *AddKel*. Este método irá decrementar os valores de *fNumElConnected* 2 (pag 55) das *n*ésimas equações sendo adicionadas. Caso uma equação seja totalmente montada (uma equação que não possui mais elementos à contribuir) a mesma é passível de decomposição.

O método *DecomposeEquation* do objeto *fFront* irá decompor a equação (ou um grupo de equações), armazenar o resultado no argumento *result* e liberar o espaço associado a equação decomposta. O objeto *result* será guardado pelo objeto *fStorage* do tipo TPZStackEqnStorage ou TPZFileEqnStorage.

Este processo ocorre enquanto houver equações a serem decompostas no domínio. A Figura 2.1 (pag 19) ilustra o processo de decomposição.

Durante o processo de montagem da matriz frontal, é necessário saber o tamanho da frente e, com isso, o espaço na memória necessário para armazenamento da matriz frontal. Este espaço pode ser calculado utilizando uma montagem simbólica utilizando SymbolicAddKel. No entanto, aplica-se nesta implementação, uma estratégia diferente, onde o tamanho da matriz frontal é alterado dinamicamente de acordo com a necessidade.

A classe TPZFrontMatrix é uma classe template, na sua construção são passados dois parâmetros referentes a: tipo de armazenamento das equações decompostas; tipo de armazenamento e decomposição da matriz frontal. O primeiro parâmetro é passado para a variável *store* enquanto que o segundo para a variável *front*.

O parâmetro *store* pode assumir dois valores, sendo eles TPZStackEqnArray ou TPZFileEqnArray onde o primeiro armazena as equações decompostas em uma pilha de TPZEqnArrays, e o segundo armazena as TPZEqnArrays na forma de um arquivo binário.

Por sua vez o parâmetro *front* pode também assumir dois valores, sendo eles TPZFrontSym e TPZFrontNonSym. A classe TPZFrontSym executa o armazenamento na matriz frontal de forma simétrica e a decomposição por Cholesky. A classe TPZFrontNonSym armazena a matriz frontal de forma não simétrica e executa a decomposição por LU.

Estrutura de Dados

1. <template> *fFront*
   Matriz frontal podendo assumir os valores TPZFrontSym e TPZFrontNonSym.

2. TPZVec<int> *fNumElConnected*
   Contém o número de elementos que ainda precisam contribuir para uma dada equação.

3. <template> *fStorage*
   Objeto que receberá as equações decompostas. Pode assumir os valores TPZStackEqnStorage e TPZFileEqnStorage.

Métodos

Os métodos associados a TPZFrontMatrix são:

1. void *AssembleMatrix*(TPZVec<int> *&eqnumbers*, TPZFMatrix *&ek*, TPZFMatrix *&ef*)
   Adiciona a matriz (e vetor de carga) na matriz frontal.

2. int *GetMaxFrontSize*();
   Retorna o tamanho máximo da frente

3. int *EquationsToDecompose*();
   Determina o número de equações que podem ser decompostas.

4. void *SetNumElConnected*(TPZVec < int > *&numelconnected*)
   Inicializa o número de elementos conectados a cada equação.

5. void *AddKel*(TPZFMatrix & *elmat*, TPZVec < int > & *destinationindex*)
   Adiciona a contribuição de um elemento à equação.

6. void *SymbolicAddKel*(TPZVec < int > & *destinationindex*)
   Adiciona simbolicamente uma contribuição de uma matriz de rigidez usando os índices para computar o tamanho da frente.

5.1.6 Classes TPZStackEqnStorage e TPZFileEqnStorage

Após decomposição, a matriz global consiste de um conjunto de objetos TPZEqnArray. Cada objeto TPZEqnArray contém linhas de equações sob forma decomposta. Definem-se as classes TPZStackEqnStorage e TPZFileEqnStorage para administrar este conjunto de objetos do tipo TPZEqnArray. Estes podem ser salvos em disco (TPZFileEqnStorage) ou armazenados numa pilha (TPZStackEqnStorage).
A definição de qual estratégia de armazenamento será utilizada ocorre na declaração do objeto TPZFrontMatrix que necessita de dois parâmetros <templates> sendo um deles o store que assume uma das classes tratadas neste tópico.

1. TPZStackEqnStorage armazena os dados numa pilha de EqnArrays.

2. TPZFileEqnStorage armazena os dados em forma de arquivos binários.

A estrutura de dados e os métodos documentados para TPZStackEqnStorage e TPZFileEqnStorage são redefinidos nas duas classes, contendo cada um suas especializações.
Para realização das substituições *forward* e *backward* que devem ser executadas sobre todas as EqnArrays contidas na estrutura de armazenamento, há uma especialização dos métodos citados em cada uma das classes. As alterações basicamente conformam-se com os diferentes dispositivos de armazenamento.

Estrutura de Dados para TPZStackEqnStorage

TPZStackEqnStorage armazena os dados na forma de uma pilha de objetos tipo TPZEqnArrays. Define-se como estrutura principal um objeto tipo TPZStack. TPZStack é uma pilha e possui os métodos *Push* e *Pop* para inclusão e exclusão dos objetos na pilha. A estrutura principal de armazenamento consiste então de:

- TPZStack<TPZEqnArray> *fEqnStack*(TPZEqnArray *Eqn*)
  Armazenará todos objetos EqnArrays

Estrutura de Dados para TPZFileEqnStorage

Para esta classe define-se como entidade de armazenamento um objeto tipo arquivo binário fFileName. Sobre ele são executadas as rotinas de leitura e escrita em disco dos objetos TPZEqnArrays envolvidos na decomposição. A estrutura principal consiste então de:

- FILE *fIOStream

Pode ser aberto para *input* ou *output* binário. A opção de se utilizar um arquivo binário para armazenamento deve-se a performance obtida com arquivos deste tipo.

Esta alternativa gera a necessidade da implementação de métodos para gerenciar a leitura, escrita e procura sob formato binário, aumentando portanto a complexidade de alguns códigos. Mesmo com esse adicional na complexidade do código, o formato binário mostra-se vantajoso quando comparado aos outros formatos.

As funções de leitura, escrita e procura redefinidas nesta classe são as principais diferenças entre as classes de armazenamento.

Organização do Arquivo Binário

O formato do arquivo binário é organizado por blocos, cada bloco consiste de um cabeçalho contendo os endereços das equações contidas no bloco.

O cabeçalho dos blocos possui n endereços e $n-1$ equações. O cabeçalho n guarda o endereço do cabeçalho do próximo bloco.

A primeira linha do arquivo contem informações globais sobre o arquivo, contém a quantidade de blocos no arquivo e a quantidade de equações por bloco.

A Figura 5.4 (pag 58) ilustra a organização do arquivo binário.

Métodos para TPZStackEqnStorage e TPZFileEqnStorage

1. void *AddEqnArray*(TPZEqnArray *EqnArray*)
   Adiciona o objeto *EqnArray* à pilha de objetos do tipo TPZEqnArray.

2. void *Forward*(TPZFMatrix &f , DecomposeType dec)
   Realiza uma substituição *forward* sobre todas equações na pilha.

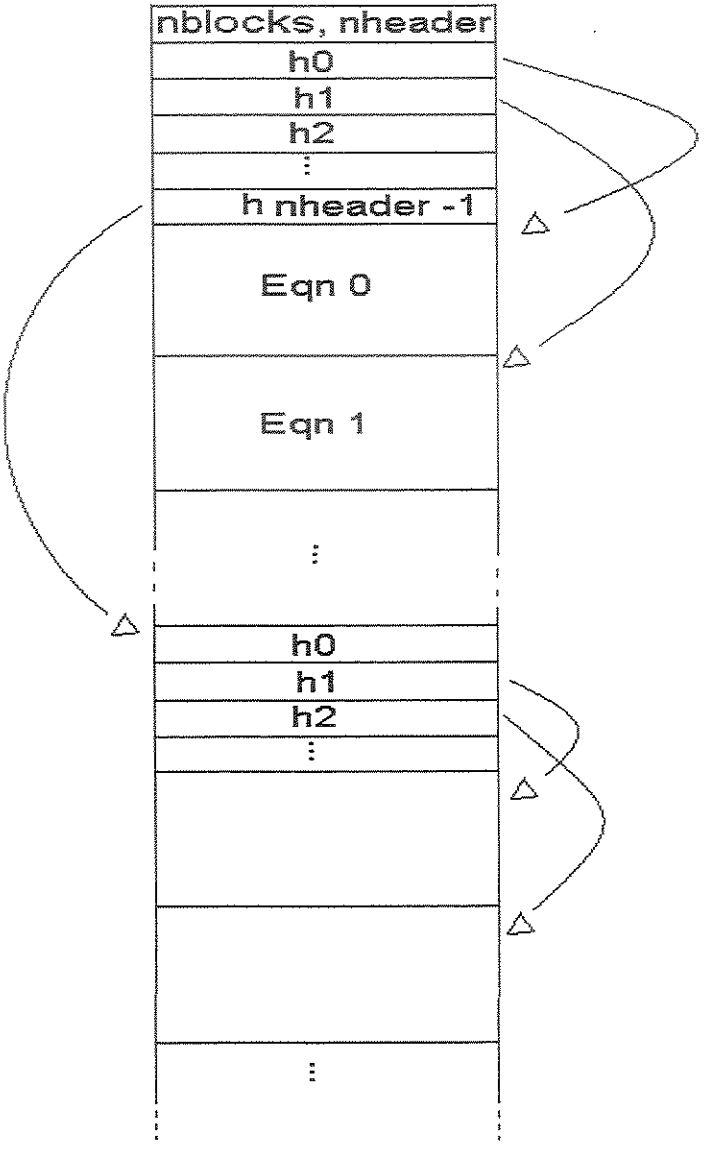


Figura 5.4: Organização do arquivo binário

3. void *Backward*(TPZFMatrix &f , DecomposeType dec)
   Realiza uma substituição *backward* sobre todas equações na pilha.

4. void *Reset*()
   Reinicializa a estrutura de dados de TPZStackEqnStorage e TPZFileEqnStorage.

5. void *Print*(const char *name , ostream& out)
   Imprime os dados armazenados em TPZStackEqnStorage e TPZFileEqnStorage.

Os métodos específicos para gerenciamento de leitura/escrita binários da classe TPZFileEqnStorage podem ser encontrados na documentação em anexo.

5.1.7 Classe TPZParFrontMatrix

Para paralelização do método frontal foi necessária a especialização da classe TPZFrontMatrix, esta especialização, resultou na classe TPZParFrontMatrix que implementa as funcionalidades de otimização por paralelismo, utilizando multi-threading. A estratégia utilizada para implementação do método redefine os seguintes métodos.

Método *AddKel*

A paralelização do processo afetou este método diretamente. Na classe base, a mesma tarefa de execução (*thread*) que decompõe as equações na matriz frontal, executa a escritura em disco das equações decompostas.
Na proposta paralela, as equações decompostas são adicionadas numa pilha de equações decompostas que fica disponível à tarefa de execução (thread) de escritura em disco. Este segundo processo (método *WriteFile*) descarrega a pilha, enquanto que AddKell carrega a pilha. Cada um dos processos ocorre em *threads* independentes.

Método *WriteFile*

Como citado no tópico anterior, o método *WriteFile* descarrega uma pilha de equações decompostas. Estas equações são então armazenadas num arquivo binário em disco. Este processo é também executado em *thread* independente.

Método *FinishWriting*

Na classe mãe, este método executa uma chamada *FinishWriting* sobre a variável fStorage. Na especialização do método para a classe paralela, um flag indicador do término da decomposição é confirmado.
No processo não paralelo, *WriteFile* escreveria em disco todas as equações. Isso ocorre percorrendo todas as equações e escrevendo-as em disco.
No processo paralelo, quando o método *WriteFile* esvazia a pilha de equações, não siginifica que todas as equações foram decompostas. Isso será indicado pelo flag confirmado por *FinishWriting*. Neste instante sabe-se que todas as equações foram decompostas e adicionadas à pilha de equações. Este método altera o valor de um flag utilizado para gerenciamento dos processos.

5.2 Classes Estruturais

Em janeiro de 2001 foram implantadas no pacote as classes de matrizes estruturais. Esta alteração atingiu diretamente o relacionamento entre as classes de elementos finitos e as classes matriciais. Anteriormente às classes estruturais, o relacionamento entre classes de elementos finitos e de matrizes ocorria de forma direta. As matrizes estruturais foram concebidas de forma a tornarem-se a interface entre as classes matriciais e as de elementos finitos.

A utilização de classes estruturais simplificam, para o usuário final, a utilização do pacote. Os serviços disponibilizados pelas classes matriciais às classes de elementos finitos, são agora encapsulados pelas matrizes estruturais. Para um programador, não existe mais a necessidade de conhecer as funcionalidades das classes matriciais, e sim o conhecimento das classes estruturais para implementação de uma análise completa.

Não fosse o pacote desenvolvido orientado para objetos, uma alteração desta magnitude nas estruturas estáticas (implantação de novas classes) e dinâmicas (relacionamento entre classes) seria uma tarefa com um alto índice de risco. A orientação para objetos fez com que estas mudanças ocorressem de forma bem suave, sem trazer maiores problemas ao grupo de desenvolvimento.

Para o funcionamento da nova estrutura, foi necessário o desenvolvimento de uma classe de matriz estrutural para cada classe matricial. Para a classe TPZFrontMatrix, foi implementada a classe TPZFrontStructMatrix por exemplo. As responsabilidade das classes estruturais não condizem à simplicidade de sua implementação, visto isso, não foi onerosa implementação de todas as classes estruturais necessárias.

5.2.1 Classe TPZFrontStructMatrix e TPZParFrontStructMatrix

A responsabilidade de classes matrizes estruturais é implementar uma interface entre as classes matriciais e as classes de elementos finitos. Ressalta-se que a maioria dos métodos serão mensagens enviadas as classes matriciais. A classe TPZParFrontStructMatrix é uma especialização da classe TPZFrontStructMatrix e opera sobre matrizes TPZParFrontMatrix, que implementam o método frontal em paralelo.

Métodos para classe TPZFrontStructMatrix

- void Assemble(TPZMatrix &, TPZFMatrix &)
  Envia uma mensagem de montagem de matriz de Rigidez à respectiva classe matricial.
- TPZStructMatrix * Clone()
  Este método retorna uma nova matriz estrutural clonada do atual objeto.
- TPZMatrix * Create()
  Retorna um ponteiro para uma matriz.
- TPZMatrix * CreateAssemble(TPZFMatrix &rhs);
  Retorna um ponteiro para uma matriz. Esta é uma função obrigatória nas matrizes estruturais. Nota-se o polimorfismo nesta função, uma vez um que ponteiro é retornado, qualquer classe matricial filha de TPZMatrix pode ser retornada.

- void GetNumElConnected(TPZVec <int> &numelconnected)
  Retorna no vetor *numelconnected* o número de elementos conectado ao iésimo grau de liberdade.

- void OrderElement()
  Reordena os elementos de acordo com enumeração dos nós de maneira a diminuir o tamanho da frente.

Capítulo 6

Conclusões

6.1 Solvers Frontais

O método frontal apresentou-se de maneira muito eficiente em termos de performance. Foram executados testes envolvendo o solver frontal, o solver skyline. A performance dos dois processos paralelizados, comparados aos seus respectivos não paralelizados, é superior em todos os casos significativos. Em casos com números reduzidos de equações (menos que 1000) os processos com paralelização são menos eficientes pois o *overhead* necessário para gerenciamento de processos e dados, prejudica o desempenho. A tabela 4.1pag. 45, apresenta resultados comparativos entre os solvers paralelizados frontal e skyline. Nota-se nesta tabela o número elevado de equações.

A eficiência do método frontal é também comprovada quando são analisados dados referentes a demanda de recursos. Em solvers com uma estrutura de dados não frontal, o consumo de memória RAM torna-se um fator dominante na determinação do rendimento. No solver frontal percebe-se que a demanda de memória, para um elevado número de equações, é inferior quando comparado com outros métodos diretos.

A estrutura do solver frontal, culmina inclusive, com um tráfego inferior de dados entre processadores e bancos de memórias, isso é efetivamente uma vantagem, pois não há até o presente momento, tecnologia de barramento de memória que promova uma velocidade de transmissão de dados, compatível com as atuais velocidades dos processadores (de 1 a 1.5 GHz). Este tráfego é então uma tarefa crítica que deve ser contemplada para aumentar a performance de um método. O tratamento dado pelo processo frontal a esta tarefa crítica, é a diminuição no volume de dados trocados entre processadores e memórias.

O armazenamento compacto da matriz frontal, promove inclusive, a possibilidade de utilização mais vantajosa de rotinas BLAS na fatoração das equações. Isso foi também um ponto crucial na determinação dos resultados de performance.

6.2 Paralelização do Método Frontal

A granularidade do processo obtido com o método frontal, permite várias perspectivas de paralelização. A solução da atual implementação é apenas uma das possíveis alternativas. Isso facilita muito a alteração da técnica de paralelização utilizada. A implementação atual utiliza

paralelismo em equipamentos com memória compartilhada (máquinas bi-processadas) utilizando *multi-threading*, uma nova implementação focada em equipamentos com memória distribuída, seria facilmente definida com um trabalho relativamente pequeno. O solução para paralelização do solver frontal é muito adaptável.

Utilizou-se para realização de testes uma análise envolvendo elementos bi-dimensionais com materiais lineares. Com este tipo de problema, nota-se que o thread destinado à montagem das matrizes elementares é finalizado num tempo bem inferior ao de decomposição das equações. Sabe-se ainda que problemas bi-dimensionais lineares apresentam matrizes elementares bem simples quando comparados a problemas bi e/ou tri-dimensionais não lineares. Portanto, espera-se que a eficiência seja ainda melhorada quando o problema envolvido for um problema não linear, pois o thread destinado à montagem das matrizes elementares receberá uma tarefa um pouco mais onerosa.

Conclui-se ainda, que a experiência adquirida com processamento paralelo, contribuiu efetivamente para ampliação do horizonte de aplicação do pacote PZ utilizado no projeto. O conhecimento adquirido facilita desenvolvimento de possíveis paralelizações em diferentes fases de uma análise numérica, bem como da aplicação de paralelismo sob diferentes técnicas.

6.3 *Multi-Threading*

Em principio, a maioria dos códigos que contemplam multi-threading, são desenvolvidos para serem utilizados em equipamentos com mais de um processador. Porém os mesmos códigos paralelos, quando executados em máquinas mono-processadas, apresentam praticamente o mesmo desempenho que códigos seriais.

Conclui-se ainda que equipamentos multi-processados fazem melhor proveito das bibliotecas BLAS, uma vez que estas são implementadas contemplando estas arquiteturas. No processo frontal, a utilização de multi-threading na decomposição da matriz frontal através de BLAS, contribui muito para a eficiência do código.

A idéia principal para utilização de multi-threading, visa a separação entre os processos de decomposição e gravação em disco. A utilização de multi-threading sob esta perspectiva, aplica-se não somente a equipamentos com mais de um processador, e também a equipamentos mono-processados. E esta aplicação mostrou-se bastante eficiente, não computando no tempo global de decomposição o tempo gasto com gravação em disco. E ainda, quando a utilização de multi-threading prejudica o desempenho de um código, este prejuízo é mínimo.

6.4 UML e Orientação para Objetos

Confirma-se mais uma vez, com a conclusão deste projeto, a importância e viabilidade da filosofia de orientação para objetos no desenvolvimento corporativo de software. Durante o desenvolvimento, algumas alterações não triviais foram executadas no pacote PZ. Estas alterações influenciaram diretamente o conteúdo deste projeto (matrizes estruturais 5.2, 60). As consequentes adaptações necessárias, foram feitas de maneira eficiente. O código estava completamente operacional em poucos dias.

Ressalta-se também a importância da aplicação das linguagens de modelagem no planejamento e desenvolvimento. No presente projeto a utilização de UML (*Unified Modeling Language*) somou ao projeto maior qualidade nas documentações geradas e também, fez com que problemas anteriormente encontrados no momento da implementação, fossem agora, isolados durante a modelagem do sistema. Nota-se na fase final do projeto uma configuração no processo de desenvolvimento, onde planejamento é efetivamente executado, e não se trata apenas de um formalismo necessário para o processo.

Alguns diagramas de UML são encontrados em 2.1 (pag 19), 2.2 (pag 20), 2.4 (pag 21) sendo estes diagramas de sequência, os diagramas 5.1 (pag 48), A.1 (pag 73) representam diagramas de classes.

6.5 Controles ActiveX

O objetivo inicial de disponibilizar funcionalidades particulares à linguagens como *C++* à outras linguagens, encapsulando estas funcionalidades em um controles *ActiveX*, apesar de alguns contratempos, foi atingido. O trabalho com a plataforma *Microsoft*, apresenta-se de maneira muito instável, a citada fabricante não possui uma filosofia de comprometimento com seus usuários. Durante o projeto de mestrado, a plataforma Windows passou de Windows NT em inglês para Windows 2000 em português, essa mudança fez com que o software desenvolvido em *Visual BASIC* que seria a base para o teste do controle *ActiveX*, parasse de funcionar. Isolou-se a causa do erro e todos estes dados foram enviados à *Microsoft*. Até o momento nenhuma resposta foi obtida por parte da *Microsoft*.

Mesmo com as turbulências encontradas no desenvolvimento sobre a plataforma Windows, a implementação do controle *ActiveX* disponibilizando o solver frontal foi concluída. O mesmo já se encontra funcional.

Capítulo 7

Bibliografia

7.1 Revisão Bibliográfica

Utilizou-se as seguintes bibliografias para os tópicos estudados:

7.1.1 Programação Orientada para Objetos e *UML*

As bibliografias para estes temas complementam-se, sendo estas [18], [1], [10] e [40] as mais utilizadas para descrição estática dos sistemas (diagramas de classes). Furlan [18] apresenta uma abordagem mais histórica mostrando-se bem interessante para um entendimento global do assunto.

No livro de Alhir [1] encontra-se uma detalhada explicação da metodologia *UML*,uma espécie de manual da *UML*.

Os autores Coad, Yourdon [10] e Rumbaugh [40] abordam nestes livros os temas por eles desenvolvidos anteriormente a *UML*. Assunto tratado no item 2.2 onde uma abordagem orientada a objetos é proposta com a utilização de métodos anteriores a *UML*.

Nos livros [41] e [7] adquiridos no segundo ano do projeto encontram-se os mais interessantes trabalhos (aos quais consultamos) sobre o assunto. Em 1 os autores e criadores da UML Rumbaugh, Jacobson e Booch detalham estensivamente todo o escopo, procedimentos e aplicações da UML, é com certeza o melhor manual de referência para a UML.

No livro [7] o autor apresenta de maneira bem didática a maioria dos conceitos da UML e durante os capítulos do livro desenvolve um projeto orientado para objetos de um simulador de vôo. No projeto do simulador de vôo a UML é utilizada para especificação e documentação do projeto. O autor aborda de maneira clara a perspectiva do desenvolvimento de software corporativo de larga escala, projetos com várias equipes de analistas e programadores e etc.

Mostra-se também como uma literatura muito interessante e que muito contribui para alguns esclarecimentos dos conceitos e aplicações da UML.

7.1.2 Linguagem C++

O livro [45] de Stroustrup continua sendo o mais indicado para referências da linguagem C++ . Apresenta explicações detalhadas de várias ferramentas da linguagem C++ exploradas no projeto.

Para trabalhar com o pacote Visual C++ da *Microsoft* na criação dos controles ActiveX, Horton [24] mostrou-se o mais adequado.

7.1.3 Programação Multi-Linguagem

Este tema envolve diferentes linguagens de programação, a referência [22] traz alguns capítulos dedicados ao assunto *MLP* (*Mixed Language Programming*). Apesar do título referenciar apenas *Visual BASIC*, encontra-se em seu conteúdo outras linguagens, inclusive o *C++*.

7.1.4 Criação de Controles ActiveX

Os livros utilizados para desenvolvimento deste assunto são geralmente direcionados ao desenvolvimento de aplicativos na linguagem Visual BASIC. Destacam-se os livros de Appleman [2], Cornell [12] e McKelvy [34]. O livro de Horton [24] sobre Visual C++ foi de grande valia para o tema. O mesmo traz exemplos do processo de criação e distribuição dos controles criados. Demonstra como deve ser feita a interação entre *C++* e *Visual BASIC*.

7.1.5 Solvers

Frontal

Trabalhos de Bruce Irons são os mais importantes para o tema. Os livros [26] e [?] de Irons trazem conceitos básicos sobre o assunto e são de grande ajuda para o entendimento do processo.

Além dos trabalhos de Irons, foram utilizadas as referências de Camarda [6], Lesoinne [32], Chandrupatla [9], Duff [16] e Zitney [57].

Cholesky para Matriz Skyline

O livro [49] de Golub e van Loan apresenta explicações mais detalhadas sobre o método de fatoração de Cholesky e vários outros métodos. Os trabalhos de George [20] e Zheng [56] aparentemente trazem variações sobre a utilização do método com paralelização, porém não tivemos acesso as referências.

O trabalho de Burden [17] mostrou-se de grande valia no entendimento detalhado dos métodos de fatoração, apresentando explicações extensas sobre as rotinas envolvidas.

7.1.6 Reordenação de Equações

O trabalho de Sloan [48] onde foi descrito o método por ele desenvolvido foi o mais utilizado neste projeto. Além deste utilizou-se também a documentação do pacote *Metis* [30].

7.1.7 Programação Paralela

O manual da SUN de programação paralela [47] foi o que mais ajudou para a entendimento do funcionamento dos dispositivos de programação multi-tarefa. A documentação da biblioteca *OMNI Thread* [39] também foi de grande valia. Estes baseiam-se nas publicações [38], [27], [37], [58], [25] e [8] as quais não tivemos acesso e também não houve necessidade.

Dongarra [15] também aborda o assunto através de uma série de estudos com diferentes métodos e diferentes arquiteturas.

Apêndice A

Reordenação de Equações

A simples implementação e utilização do método apropriado ao problema não garante o sucesso na performance, muitas vezes é necessário um tratamento inicial ao sistema a ser resolvido.

Para uma previsão da performance que o método frontal obterá na solução de um sistema, deve-se avaliar um fator denominado *profile*.

Dadas as definições:

- Largura da frente (*frontwidth*) como sendo a diferença entre os índices das colunas do primeiro elemento não nulo e a diagonal principal.
- Profile como sendo a somatória das larguras da frente (*frontwidth*) para todas as linhas.

Sabe-se que a eficiente utilização do solver frontal requer o menor *profile* possível para a matriz dos coeficientes. Nota-se ainda que o profile depende das adjacências entre elementos e nós. Sendo a lista das adjacências objeto dos métodos de reordenação. [48]

A.1 Métodos de Reordenação

O ambiente PZ possui duas implementações de métodos para reordenação de equações. Um método desenvolvido por George Karypis e Vipin Kumar em 1997 denominado *METIS* [30], sendo esta uma biblioteca colocada a disposição para uso acadêmico. A implementação deste método encontra-se na classe TPZMetis.

Outro procedimento foi desenvolvido por Sloan [48] em 1989. Este método foi incorporado no ambiente PZ neste projeto e portanto é descrito a seguir.

A.1.1 Método de Sloan

Na descrição do algoritmo de reenumeração são utilizados conceitos da teoria dos grafos, portanto, algumas definições de tal teoria são necessárias. Em [48] Sloan introduz as seguintes definições:

Grafo. É definido como sendo uma série de nós, juntamente com uma série de pares desordenados distintos, chamados arestas. Um grafo que satisfaça esta condição é denominado indireto se todos

os pares definindo as arestas são desordenados. Não são computadas arestas que conectem nós a si próprio, nem arestas múltiplas, onde um par de nós é conectado por mais de uma aresta.

O grau de um nó é o número de arestas incidentes a este nó.

Qualquer par de nós conectados por uma aresta são denominados adjacentes. A distância entre um par de nós é definida como o número de arestas existentes no caminho entre os dois nós. A maior distância encontrada entre uma par de nós é denominada diâmetro. Geralmente o diâmetro é definido por vários pares de nós. Estes nós que definem o diâmetro são chamados nós periféricos.

Baseado ainda na teoria dos grafos, define-se "pseudo-diâmetro" como sendo qualquer estimativa do diâmetro que é encontrado com algum algoritmo de aproximação. Nós que definem pseudo-diâmetro são denominados nós "pseudo-periféricos".

Um conceito comum à maioria dos algoritmos de reenumeração é o conceito de estruturas de níveis, isto é, uma partição dos nós onde cada nó é assinalado a um dos níveis, $n_1, n_2, ..., n_h(r)$ de acordo com sua distância até um nó específico r. Todos equidistantes de r pertencem ao mesmo nível. A profundidade de uma estrutura de níveis, $h(r)$, é simplesmente o número total de níveis e a sua largura é o máximo número de nós pertencentes a um único nível.

Pode-se tratar uma matriz simétrica como sendo um grafo indireto. Cada nó em um grafo corresponde a uma linha na matriz e o grau do nó corresponde ao número de termos não nulos localizados fora da diagonal principal. Sendo que um par de nós i e j estão conectados por uma aresta apenas se a_{ij} e a_{ji} são não nulos, segue que o total de termos não nulos na matriz é dado por $2E + N$ onde E é o número de arestas e N é o número de nós.

Nas aplicações de elementos finitos, o grafo correspondente a uma malha é construído de maneira que todos os nós que incidem no mesmo elemento estão conectados por uma aresta.

A.1.2 O Algoritmo de Sloan

Conforme Sloan em [48], a rotina pode ser dividida em duas partes distintas: na primeira é realizada uma escolha dos nós pseudo-periféricos, e então prossegue a reenumeração dos nós.

Seleção dos nós pseudo-periféricos. Antes de iniciar a reenumeração são determinados nós pseudo-periféricos que se encontram nos extremos de um pseudo-diâmetro. Estes nós serão os nós inicial e final para subsequente reenumeração. O processo ocorre da seguinte maneira.

1. (Primeira tentativa para nó inicial) Selecionar o nó s com menor grau em todo grafo.
2. (Geração da estrutura de níveis) Montagem da estrutura de níveis baseada no nó s.
3. (Ordenar o último nível) Ordenar os nós pertencentes ao último nível em ordem crescente.
4. (Diminuir último nível) Percorrer o último nível e formar uma lista q contendo apenas um nó de cada grau.
5. (Inicialização) Atribuir a $w(e)$ um valor infinito.
6. (Teste para terminar rotina) Para cada nó i pertencente à q, em ordem crescente de grau, gerar $L(i) = \{n_1, n_2, ..., n_h(i)\}$. Se $h(i) > h(s)$ e $w(i) < w(e)$ atribui à s o valor i e retorna ao passo 3. Ou então se $w(i) > w(e)$ atribui à e o valor i e $w(e) = w(i)$.

7. (Final) Término da rotina com nó inicial s e nó final e.

Depois de definidos um par de nós pseudo-periféricos com o algoritmo anterior, a reenumeração dos nós é realizada num único passo.

Para uma descrição suscinta do algoritmo, define-se qualquer nó que já tenha sido reenumerado como sendo pós-ativo. Nós adjacentes a nós pós-ativos e que não forem nós pós-ativos, são denominados nós ativos. Qualquer nó adjacente a um nó ativo, que não seja pós-ativo nem ativo, será denominado pré-ativo, e finalizando, qualquer nó que não pertença a nenhum dos grupos acima descritos são chamados nós inativos.

A medida que o processo de reenumeração ocorre, o crescimento na *wavefront* atual é medido por uma quantidade chamada "grau corrente". Todos os nós pós-ativos possuem grau corrente igual a zero, enquanto que o grau corrente de um nó ativo é igual ao número de nós pré-ativos vizinhos. O grau corrente de um nó pré-ativo é igual ao número de nós pré-ativos e inativos vizinhos mais um. Cada nó inativo possui um grau corrente que é igual ao seu grau mais um. Anterior à reenumeração, cada nó no grafo é inativo e portanto possui um grau corrente igual ao seu grau mais um. Depois da reenumeração completa, todos os nós são pós-ativos e possuem grau corrente igual a zero.

Para iniciar a reenumeração dos nós são necessários os nós inicial e final que definirá o pseudo-diâmetro. Em seguida renumera-se o nó inicial como sendo o nó 1 e monta-se a lista de possíveis nós a serem renumerados depois. Esta lista é formada por nós ativos e/ou pré-ativos e constitui uma fila prioritária. Cada nó na fila possui uma prioridade onde um nó com baixo grau corrente e uma longa distância do nó é traduzido como alta prioridade. O nó com a mais alta prioridade é então escolhido como sendo o próximo nó a ser reenumerado e excluído da fila.

A fila é portanto atualizada utilizando a informação de conectividade do grafo e todo processo é repetido até que todos os nós tenham sido renumerados.

O algoritmo total pode ser resumido em:

1. (Entrada de dados) Entrar com os nós finais de um pseudo-diâmetro, nós s e e.

2. (Compute as distâncias) Montar a estrutura de níveis no nó final, $L(e) = \{l_1, l_2, ..., l_h(e)\}$, e calcular a distância Δ_i de cada nó i até o nó final. Perceba que se o nó i pertence a l_j, então $\Delta_i = j - 1$.

3. (Definir estado inicial e prioridades) Defina cada nó no grafo como sendo inativo e compute sua prioridade, p_i, conforme: $p_i = W_1 * \Delta_i * W_2(d_i + 1)$ onde W_1 e W_2 são pesos inteiros e d_i é o grau do nó i.

4. (Inicializar contagem dos nós e fila de prioridades) Fazer $l = 0$, onde l é o total de nós já renumerados, e defina nó s como pré-ativo. Sendo q a fila de prioridades de comprimento n. Insira s na fila de prioridade fazendo $n = 1$ e $q_n = s$.

5. (Teste para finalização). Enquanto a fila de prioridade não está vazia, que significa $n > 0$, repita passos de 6 até 9.

6. (Selecionar nó para ser reenumerado) Procurar na fila de prioridades o nó i que possui a maior prioridade. Seja m o índice do nó i tal que $q_m = i$.

7. (Atualizar fila e prioridades) Excluir o nó i da fila fazendo $q_m = q_n$ e decrementando $n- > n-1$. Se o nó é não pré-ativo, ir para o passo 8. Caso contrário, examinar cada nó j adjacente ao nó i e incrementar a prioridade deste fazendo $p_j = p_j + W_2$ (isso corresponde a decrementar o grau do nó j de uma unidade). Se nó j é inativo, então inserí-lo na fila de prioridades como pré-ativo fazendo $n < -n+1$ e $q_n = j$.

8. (Renumerar próximo nó) Renumerar nó i com seu novo número incrementando o contador de nós de uma unidade, e fazendo $\eta_i = l$, onde η é a lista com os novos números dos nós. Assinalar ao nó i o estado pós-ativo.

9. (Atualizar prioridades e fila) Examinar cada nó j que é adjacente ao ao nó i. Se o nó j não é pré-ativo nada ocorre. Caso contrário, assinalar ao nó j o estado ativo, fazer $p_j = p_j + W_2$, e examinar cada nó k que é adjacente ao nó j. Se o nó k não é pós-ativo, incrementar sua prioridade conforme $p_k < -p_k + W_2$. Se nó k é inativo, inserí-lo à fila de prioridades com um estado pré-ativo fazendo $n < -n+1$ e $q_n = k$.

10. (Sair) Término da rotina com os novos números dos nós em η, de maneira que o η_i é o novo número do nó i.

A.1.3 Conclusões sobre o Método de Sloan

Testes realizados verificam a eficiência da rotina. Uma lista de adjacências má formada é sempre renumerada resultando em ordenações melhores. Na utilização do método frontal, um bom desempenho depende do tamanho do *profile* da matriz dos coeficientes. Infelizmente os resultados dos citados testes perderam-se ao longo do projeto, estes foram realizados num equipamento com Windows NT, programando-se em VB.

Aplicando a rotina a uma lista de adjacências, o *profile* obtido é sempre menor quando comparado ao inicial. Em alguns casos a largura da banda, após a reenumeração, sofre um aumento. Para armazenamento em banda isso seria desfavorável. Como o objetivo da reenumeração não é a diminuição da banda e sim do *profile*, não há dúvida da funcionalidade da rotina.

A.2 Classes Desenvolvidas para Solver Skyline

Na fase inicial do projeto foram desenvolvidas duas classes. A primeira implementa a funcionalidade da reordenação de equações pelo método de Sloan, foi denominada TPZSloan e adicionada ao ambiente PZ.

A segunda classe foi a TPZSkylParMatrix que redefine o método de decomposição por Cholesky para matrizes do tipo Skyline com técnicas de otimização por paralelismo. O desenvolvimento desta classe serviu como uma introdução às técnicas de gerenciamento de multi-threading.

A.2.1 Classe TPZSloan

A classe responsável pelo método Sloan de reenumeração de equações é a TPZSloan. Tanto esta quanto a TPZMetis redefinem o método Resequence(), utilizando cada uma sua rotina de reordenação das incidências. O tópico A, pag. 68, trata do assunto implementado.

São passados a este método a lista das adjacências e retornado a lista com a nova ordenação. Na posição i do vetor reordenado estará a nova posição para o nó i. Obtém-se também o *profile* das novas adjacências.

A.2.2 Classe TPZSkylParMatrix

Especializou-se a classe TPZSkylMatrix, responsável por armazenamento e operações de matrizes tipo Skyline, redefinindo o método de decomposição por Cholesky. Na nova classe, TPZSkylParMatrix, o método de *Cholesky* é implementado com recursos de paralelismo.

A nova classe herda toda estrutura e funcionalidade de sua classe mãe, que são traduzidos nos atributos e operações. A Figura A.1, pag. 73 contém o diagrama de classes do pacote PZMatrix, onde se encontra a classe em questão:

O diagrama mostrado na Figura A.1, p. 73 representa a estrutura de classes do módulo matriz. O detalhe da classe TPZSkylParMatrix implementada neste projeto é mostrado na Figura A.2, p. 74.

A decomposição em paralelo implementada neste método foi descrita na seção 3.2.1, pag. 32, Para o funcionamento da rotina é necessário a obtenção de colunas disponíveis para fatoração, trabalho realizado pelo método ColumnsToWork. Neste método estão envolvidas algumas funções de gerenciamento multi-tarefas. Segue desenho esquemático e funcionamento da rotina.

ColumnsToWork

A procura de uma coluna disponível, não pode ser executada por mais de uma tarefa, caso contrário dois ou mais threads encontrariam a mesma coluna para fatorar. Isso é garantido utilizando o dispositivo mutex.
Esta rotina envolve a utilização de três funções multi-tarefa, mutex_lock, cond_wait e cond_signal. São aplicadas da seguinte maneira:

1. Mutex_Lock
   Travando um mutex, garante-se que apenas o thread que obteve sucesso no travamento está trabalhando com os dados naquele instante.
   Se um thread encontra uma coluna à ser fatorada, este irá liberar o mutex e chamar Cond_Signal para reativar threads suspensos.

2. Cond_Wait
   Um thread trava um mutex e procura uma coluna para trabalhar. Caso não ache nenhuma este irá suspender-se chamando Cond_Wait.

3. Cond_Signal
   Quando um thread termina a procura de uma coluna, chama Cond_Signal que irá reativar os threads suspensos com Cond_Wait.

O método descrito é importante para o funcionamento paralelo da rotina, pois este garante a integridade dos dados gerenciando as tarefas. Além das funções citadas, a implementação utiliza

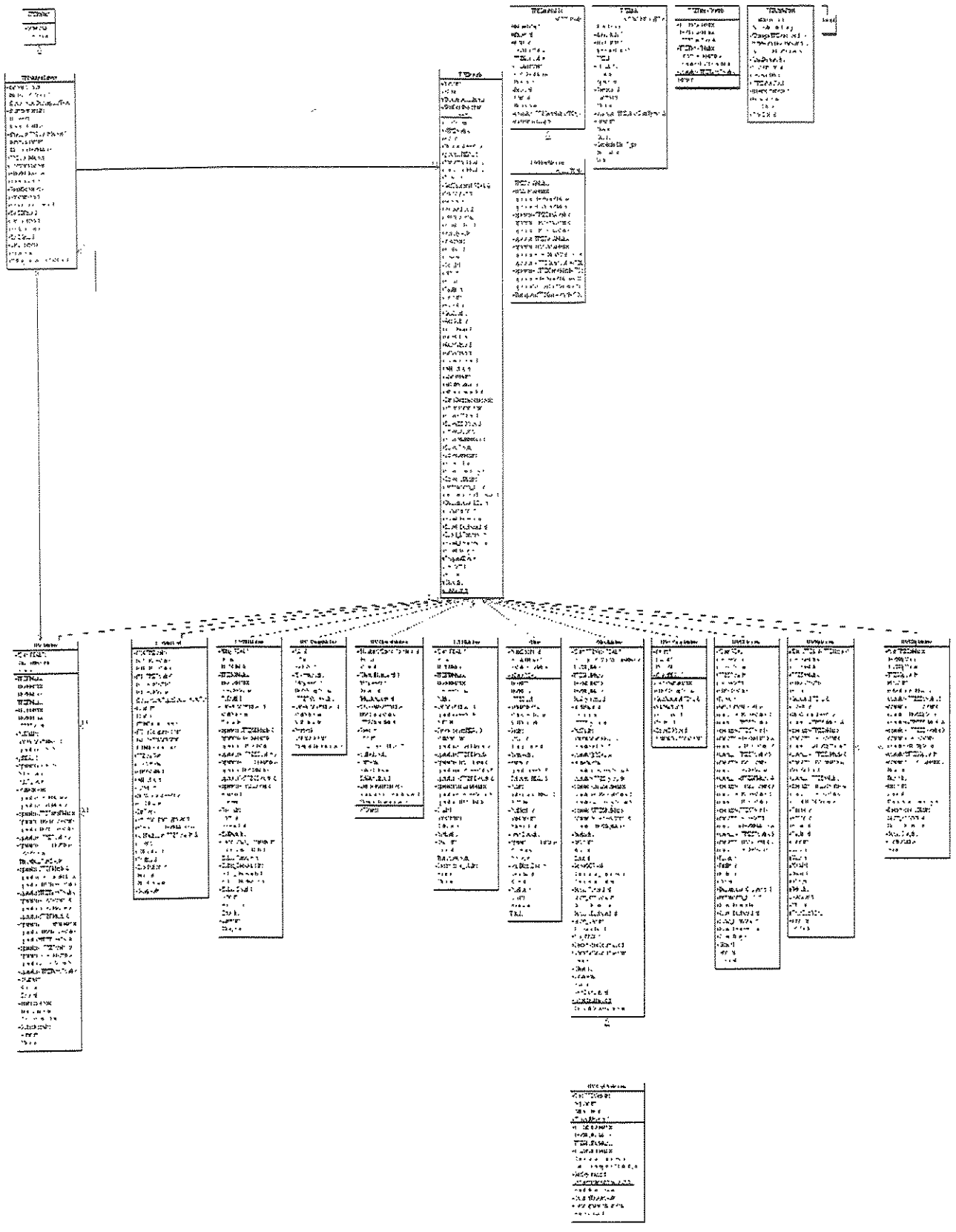


Figura A.1: Estrutura de classes matriciais

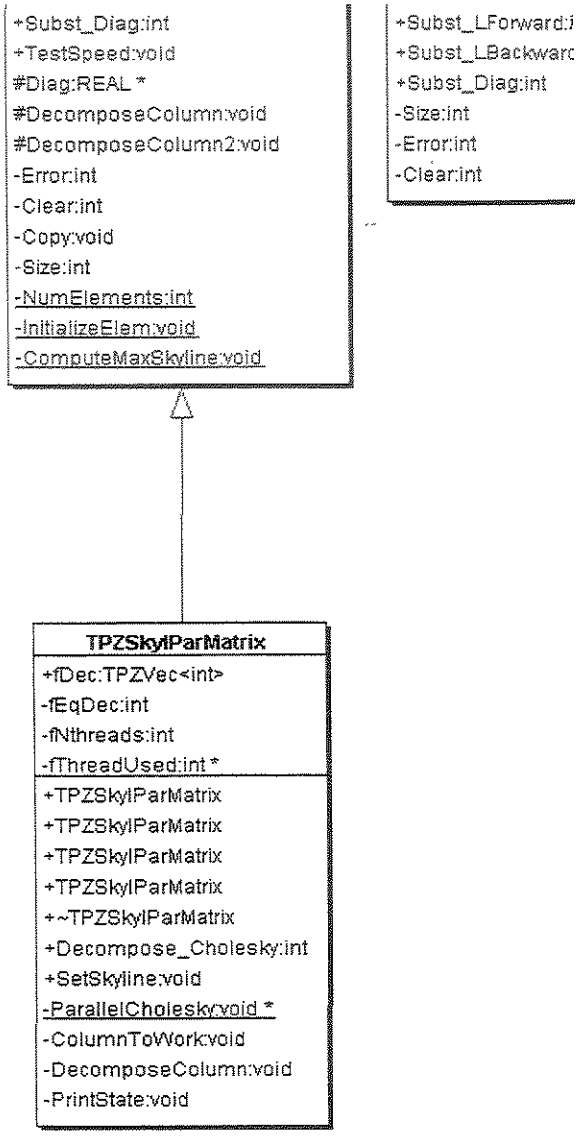


Figura A.2: Classe TPZSkylParMatrix

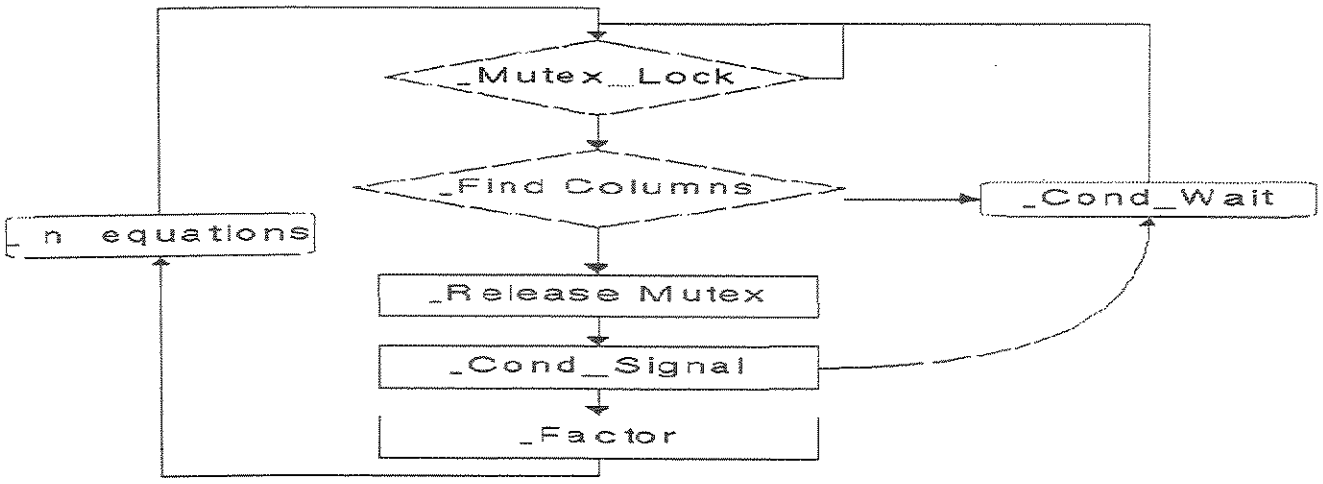


Figura A.3: Funcionamento do processo de decomposição em paralelo

pthread_create() e pthread_join() (estas para biblioteca pthread, para omni são utilizadas as similares). Uma descrição mais detalhada destas encontra-se disponível nos ítens 3.1.2, p. 27. A Figura A.3, p. 75 apresenta o esquema de fatoração em paralelo.

Apêndice B

Templates e *BLAS*

B.1 Estudo de Desempenho Numérico

B.1.1 Utilização de *Templates*

Um dos últimos artifícios de programação incorporados ao C++ foram os Templates. Estes foram experimentalmente inseridos ao pacote C++ como uma ferramenta de programação.

Segundo Mark Nelson [35] pode-se conceber Templates como uma maneira automatizada de se gerar classes e funções especializadas dinamicamente. A utilização de templates incorpora ao projeto vantagens de implementação e performance.

Implementação

Suponha uma parte específica de um código onde há a necessidade de se executar as mesmas funções Porém com dados de tipos diferentes. Por exemplo uma multiplicação entre um vetor e uma matriz para as seguintes situações; as duas variáveis com dupla precisão e as duas variáveis com precisão simples.

Ao invés de se escrever duas funções sendo que a diferença entre elas seja apenas o tipo dos dados, utiliza-se Templates onde uma espécie de "receita" será passada como parâmetro. Contendo o algoritmo e estrutura e tipo de dados utilizado. Com isso, no momento da execução será gerado automaticamente o código com os tipos de dados corretos.

Uma vez entendida a sintaxe específica para este tipo de codificação um programador não precisará mais reescrever seus códigos cada vez que for necessário mudar o tipo de dados. Dominando esta técnica, o tempo de implementação será reduzido e os códigos serão cada vez mais reutilizáveis.

Desempenho

Templates podem ser utilizado para expandir (unroll) loops em nível de compilação. Assim :

Sum<5> a; será expandido como :

a[0]+a[1]+a[2]+a[3]+a[4]

O código a seguir representa uma função que soma os valores de dois vetores, a função TemplateSum é também definida no código.

```
template<int N>
inline double TemplateSum(const REAL *p1, const REAL *p2){
return *p1 * *p2 + TemplateSum<1>(const double *p1, const double *p2);
}
template<> inline double TemplateSum<1>(const double *p1, const double *p2)
{
return *p1 * *p2;
}
#define templatedepth 10;
#ifdef USETEMPLATE
while(run1-ptrprev > templatedepth) {
run1-=templatedepth;
run2-=templatedepth;
sum += TemplateSum<templatedepth>(run1-,run2-);
}
#endif
```

pela própria função template. O compilador consegue otimizar a expressão expandida de forma muito eficiente, o que resulta em uma melhor performance do código final. Nos trabalhos de Veldhuizen [51] e [52], no projeto Blitz++ mostrou que código que utiliza o conceito de templates alcança performance maior que FORTRAN.

Aprendemos a implementar e a utilizar este tipo de estrutura da linguagem C++. As referências consultadas no assunto foram [24], [52], [51].

B.1.2 Intel BLAS MKL

Uma *BLAS* (*Basic Linear Algebra System*) é uma biblioteca otimizada de funções algébricas. Para cada arquitetura de processador encontra-se a *BLAS* específica, desenvolvida especialmente para aquela arquitetura.

Uma *BLAS* é dividida em três níveis, o nível 1 envolve operações entre vetores, o nível 2 operações entre vetores e matrizes e o terceiro realiza operações entre matrizes e matrizes. Possuem distinção para tipo de precisão dupla e simples.

A *Intel* disponibiliza a biblioteca *MKL* (*Math Kernel Library*) desenvolvida para as arquiteturas PENTIUM (II e III). A utilização desta *BLAS* em máquinas equipadas com os processadores citados, aumenta de maneira significativa a performance do código.

Foi utilizado neste projeto duas versões da biblioteca BLAS MKL da intel, na fase inicial que compreende o desenvolvimento do solver Skyline, utilizou-se a versão 3.2, os resultados são descritos no tópico B.1.2, pag. 78.

Para o solver Frontal foi utilizada a versão 5.0 da intel. Resultados são descritos na seção B.1.2, pag. 78. Diferente dos resultados na aplicação do solver Skyline, os resultados com o solver Frontal foram bem satisfatórios.

Performance com MKL 3.2 para Solver Skyline

Para a biblioteca MKL 3.2 a afirmação de que a utilização de BLAS melhora a performance do código não aplica-se a rotinas com estrutura paralela. Testes realizados com rotinas paralelas e não paralelas, ambas contendo *BLAS* (*MKL3.2*) mostram que nas implementações sem suporte a paralelismo (i.e. serial) a performance é aumentada. Porém, quando implementadas as rotinas *BLAS* em códigos paralelos o resultado obtido é exatamente o inverso, a *BLAS* faz com que a performance caia a níveis até incompatíveis.

Durante a execução do método de fatoração da matriz dos coeficientes, há uma chamada por parte dos dois (ou mais) threads concorrentes de uma função da biblioteca BLAS. Esta chamada concorrente deve ser a causa da queda na performance.

Acredita-se que rotinas pertencentes da biblioteca BLAS são executadas por apenas um thread de cada vez (i.e. elas não são reentrantes). Quando uma segunda instância da rotina é requerida esta terá que esperar a total conclusão da primeira instância. Apesar da segunda instância estar sendo direcionada para um outro processador, o primeiro bloqueará o funcionamento deste.

Esta função não permite ser chamada recursivamente mais de uma vez, garantindo assim todos os recursos da máquina para instância atual. Não há porém, documentação disponível para comprovar estas suposições [13].

Na Figura B.1 p. 79 encontra-se gráfico com os dados dos testes realizados com a biblioteca MKL 3.2 para o sistema NT.

Os testes realizados com a versão 3.2 da blas Intel MKL foram realizados na fase inicial do projeto. Na implementação de bibliotecas BLAS no solver frontal utilizou-se a mesma BLAS MKL da Intel, versão 5.0. Esta última versão foi também implementada no solver Skyline e o rendimento do código paralelo continuou sendo desastroso.

Performance com MKL 5.0 com o Solver Frontal

As técnicas de paralelização aplicadas nos solvers Skyline e Frontal são distintas. No solver skyline é paralelizada a rotina de fatoração de uma equação, onde um produto externo entre dois vetores é realizado e a matriz resultante deste produto é subtraida da matriz de rigidez. Conclui-se que o baixo rendimento do solver skyline com a utilização de BLAS é decorrente da existência de mais de um processo utilizando a mesma rotina da biblioteca BLAS.

No solver Frontal a tática de paralelização empregada e o próprio esquema de decomposição e montagem da matriz frontal, resulta na aplicação de rotinas BLAS em processos não paralelizados, havendo portanto um ganho de performance quando da utilização destas bibliotecas.

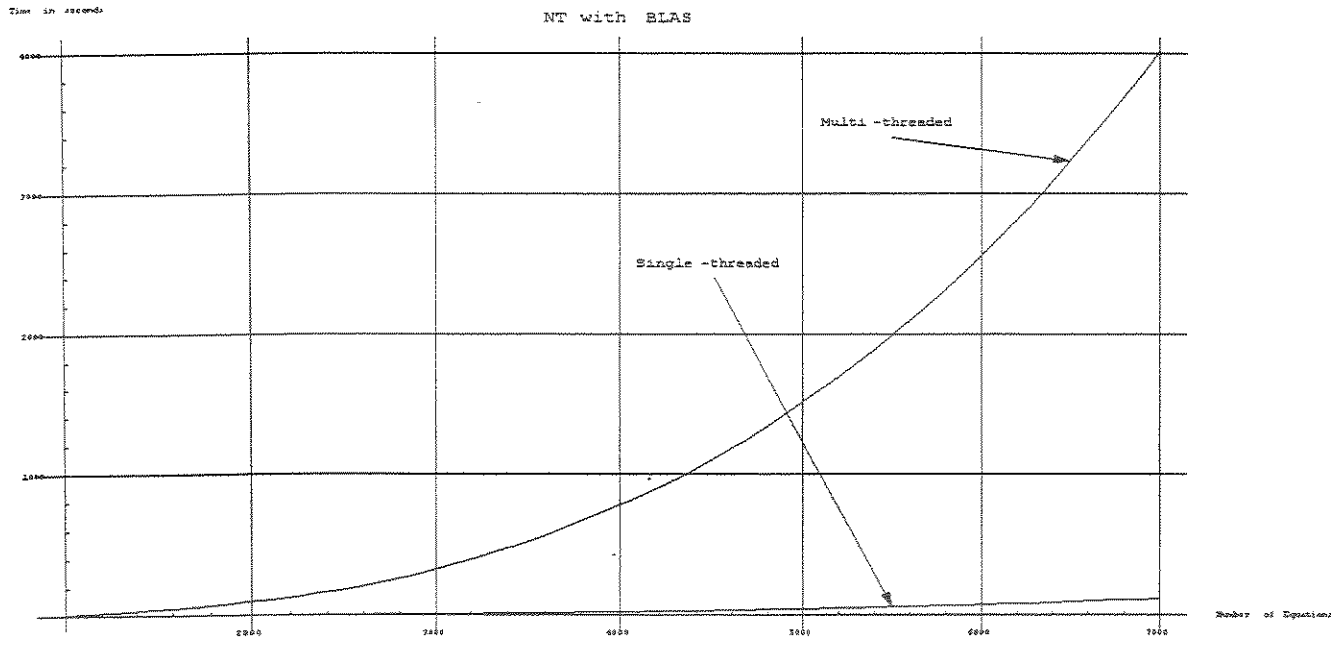


Figura B.1: Performance MKL 3.2 no sistema NT

A sofisticação dos códigos encontrados na BLAS da intel, permite ainda a utilização de multi-threading internamente à biblioteca. Ativa-se a variável de ambiente MKL_NPROCS atribuindo-a o valor de número de processadores (2 nos nossos equipamentos) encontrados no equipamento e a performance melhora ainda mais.

Apêndice C

Ambientes Computacionais

C.1 Ambientes Computacionais Utilizados

O projeto foi desenvolvido utilizando-se duas plataformas computacionais: plataforma WINDOWS NT de propriedade da *Microsoft*; plataforma GNU&Linux sob licença GPL (*Gnu Public Licence*).

Na fase inicial do projeto os pacotes disponibilizados para o ambiente WINDOWS NT superavam os da plataforma GNU&Linux. Isso ocorreu durante o primeiro ano do projeto. Em seguida foram lançados novas plataformas IDE para desenvolvimento em GNU&Linux.

C.1.1 Plataforma WINDOWS NT

Ambiente de Desenvolvimento

Para o desenvolvimento na plataforma NT, utiliza-se o pacote Visual Studio 6.0 da Microsoft, este contém o Visual C++ e o Visual BASIC. Possui uma interface de desenvolvimento amigável, com requintadas ferramentas de edição.

O IDE (*Integrated Development Environment*) que acompanha o Visual C++ versão 6.0 é a única ferramenta necessária para se criar, compilar, gerar executável e testar aplicativos desenvolvidos com o *Visual C++* [24].

Apesar das atrativas características do produto fornecido pela Microsoft, o desenvolvimento baseado nesta ferramenta foi quase impossível. Como a fabricante não distribui software sob licenças GPL, foi necessária a compra do pacote Visual Studio, dentre os editores que compõe o pacote, os principais são o *Visual C++* para *C++*, *Visual BASIC*, *FoxPro* e o *Visual J++* para linguagem *Java*. Havia por parte do orientador a realização de trabalhos com *Java*, quando procuramos pelo o aplicativo *Visual J++* na instalação fornecida pela Unicamp, sob licença da *Microsoft*, notamos que este não existia. Tentou-se solucionar tal problema através da *Microsoft* e mesmo pelo centro de computação da Unicamp. Até hoje o problema não foi solucionado.

A posição da fabricante é que a obtenção das mídias com os pacotes pode ser feita de qualquer maneira, desde que na instalação seja feito o registro do produto baseado na licença. Percebe-se que apesar de se pagar pelo produto, não há um comprometimento do fornecedor com o usuário.

Muito mais interessante é a licença GPL, onde o produto é gratuíto e a obtenção dos mesmos pode ser feita por simples *dowloads*.

Devido aos problemas relacionados acima envolvendo diretamente a Microsoft, que o desenvolvimento do projeto norteou-se mais para as tendências do movimento GPL, opção que contribuiu muito para o perfeito andamento do projeto.

Editor

O editor fornece um ambiente interativo para criação e edição de fontes *C++*. Bem como os dispositivos comuns de *copiar/colar*. Há também uma seleção de cores para diferenciação das palavras protegidas.

Um dispositivo de auxílio à edição muito interessante, é o *Intelisense* . Uma vez definida uma função, este dispositivo lista o tipo e em qual ordem as variáveis devem ser passadas. A digitação é monitorada.

Compilador

A função do Compilador é interpretar o código fonte e transformá-lo em linguagem de máquina, também detectar e informar a presença de erros no processo de compilação. O compilador é capaz de encontrar erros de várias naturezas, desde codificação inválida até erros estruturais. Os resultados de uma compilação são conhecidos como os códigos dos objetos e são armazenados em arquivos tipo objeto, que normalmente possuem extensão *obj*.

O *Linker*

O linker combina os vários módulos gerados pelo compilador a partir dos códigos fontes, adiciona os módulos necessários das bibliotecas diposnibilizadas como parte do *C++*, e conecta todos estes módulos gerando um executável. O *Linker* pode também detectar erros na *linkagem*, por exemplo quando alguma biblioteca requerida não está disponível ou parte do programa está perdida.

C.1.2 Plataforma GNU&Linux

No primeiro ano do projeto, as interfaces gráficas disponibilizadas para a plataforma GNU&Linux encontravam-se ainda em desenvolvimento. Depois do primeiro ano de projeto foram lançados alguns pacotes para desenvolvimento com ambiente IDE, conforme os tópicos seguintes.

Ambiente de Desenvolvimento

Para o desenvolvimento na plataforma GNU&Linux, inicialmente não existia um ambiente integrado onde todas as ações referentes a desenvolvimento poderiam ser executadas. A partir do segundo ano de projeto foi lançado o aplicativo KDEStudio, que apresenta algumas facilidades para desenvolvimento como: arvore de classes; compilador e linker integrados e etc. Outro aplicativo também com ambiente integrado de desenvolvimento, foi lançado, o KDeveloper, mas este não foi utilizado no projeto.

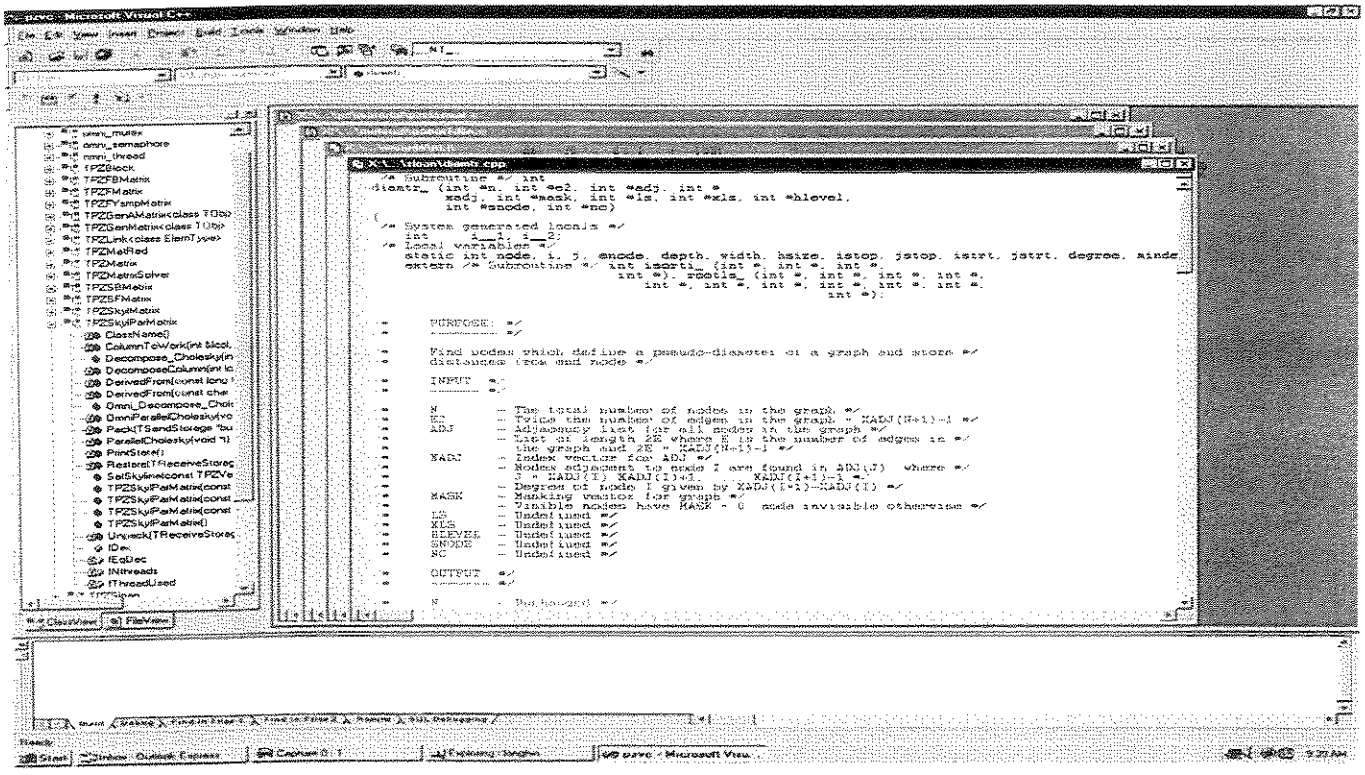


Figura C.1: Desenvolvimento com Visual C++

Desenvolvimento com KDEStudio

O aplicativo KDEStudio foi desenvolvido pela empresa *The Kompany*, é disponibilizado gratuitamente sob licença GPL, são distribuídos tanto os arquivos binários para instalação quanto os códigos fontes para compilação e linkagem. O ambiente integrado de desenvolvimento apresenta, de maneira geral, as mesmas facilidades que o software da microsoft (Visual Studio) com a vantagem de ser mais estável e gratuito. A Figura C.2 (pag. 83) apresenta uma captura de tela do software KDEStudio.

Editor para Fase Inicial

Inicialmente utilizou-se o editor *xemacs* para o ambiente X (ambiente gráfico do GNU&Linux). Este editor não é específico para desenvolvimento apenas em C++, é configurável permitindo edições em diferentes linguagens, sendo C++ uma delas.

Disponibiliza recursos comuns de edição como copiar/colar e também diferencia palavras protegidas na linguagem escolhida para o desenvolvimento.

Compilador e *Linker*

No Linux dispõe-se do compilador *g++*. Com uma sintaxe específica são passados como parâmetros os arquivos envolvidos, diretivas de compilação e caminhos das bibliotecas. Isto é realizado diretamente no terminal.

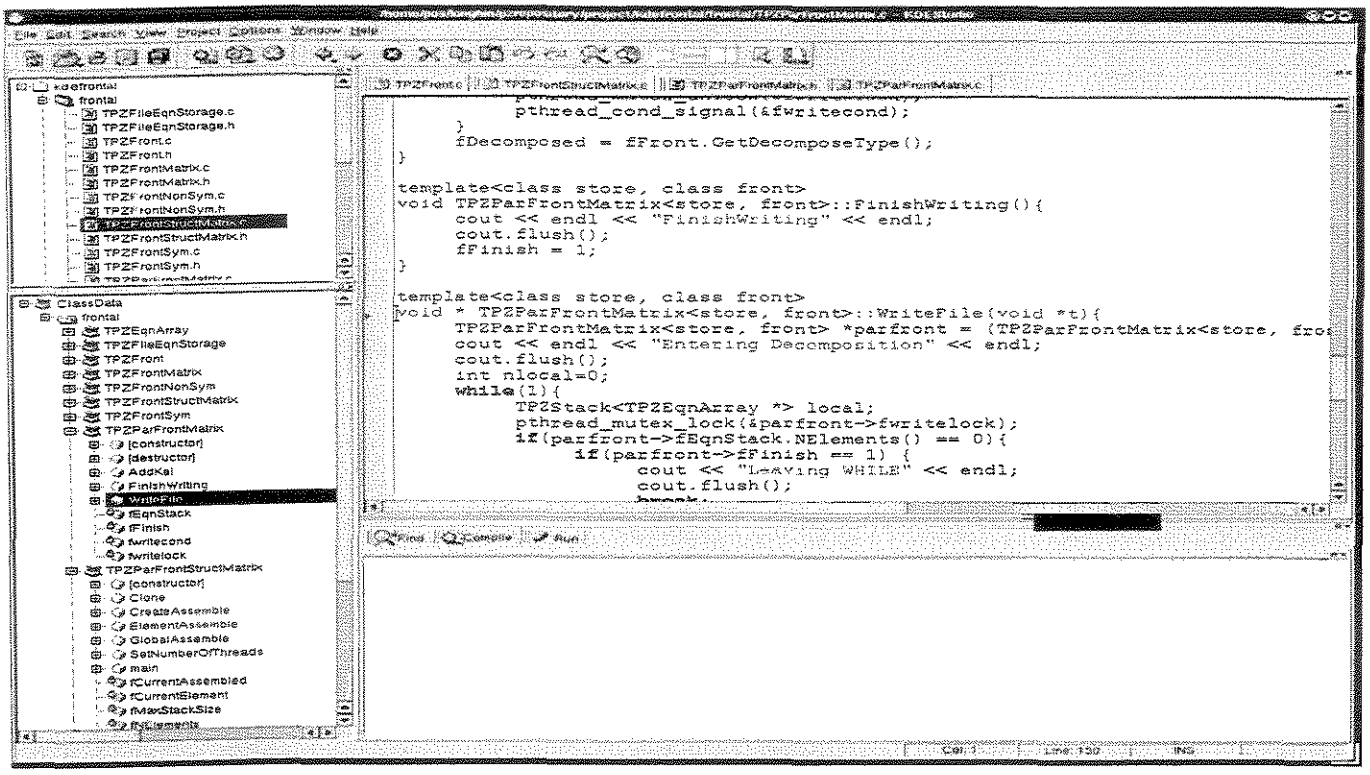


Figura C.2: KDEStudio

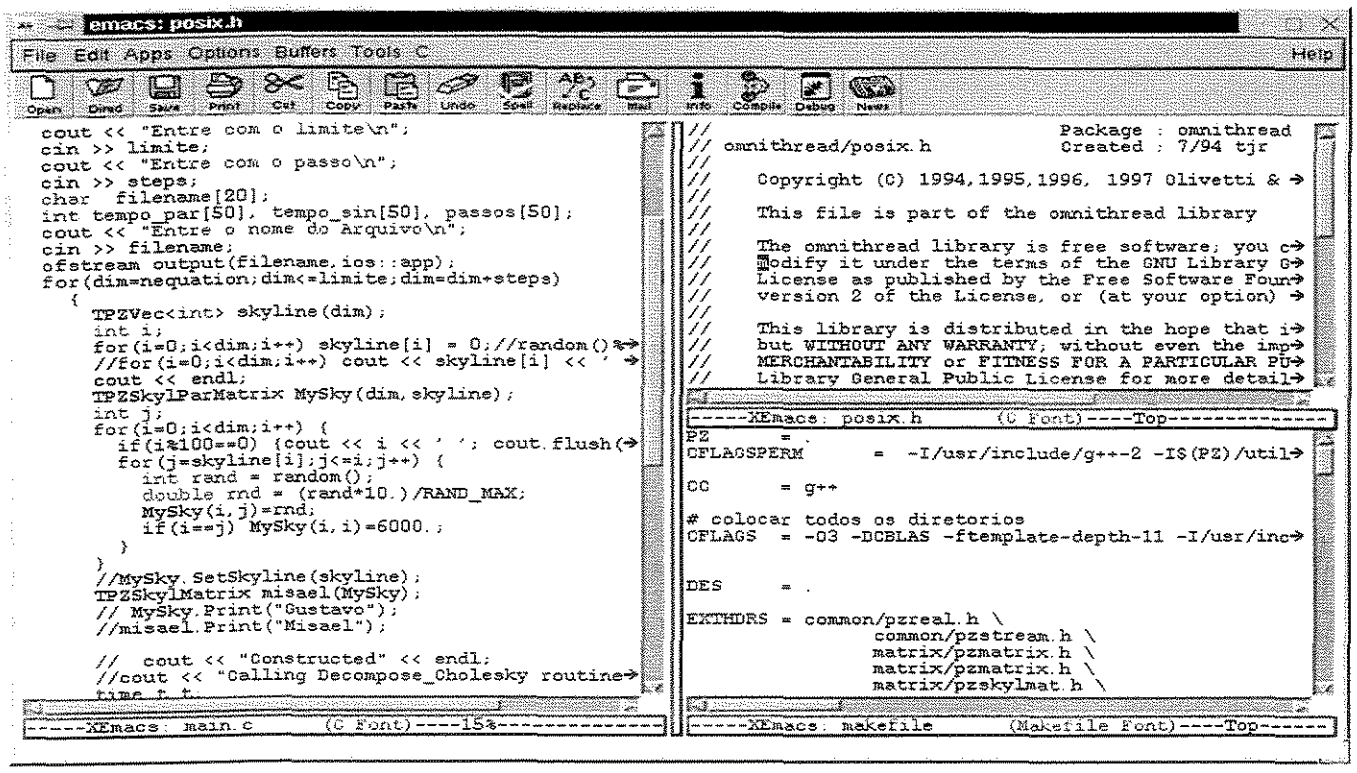


Figura C.3: Desenvolvimento com xemacs

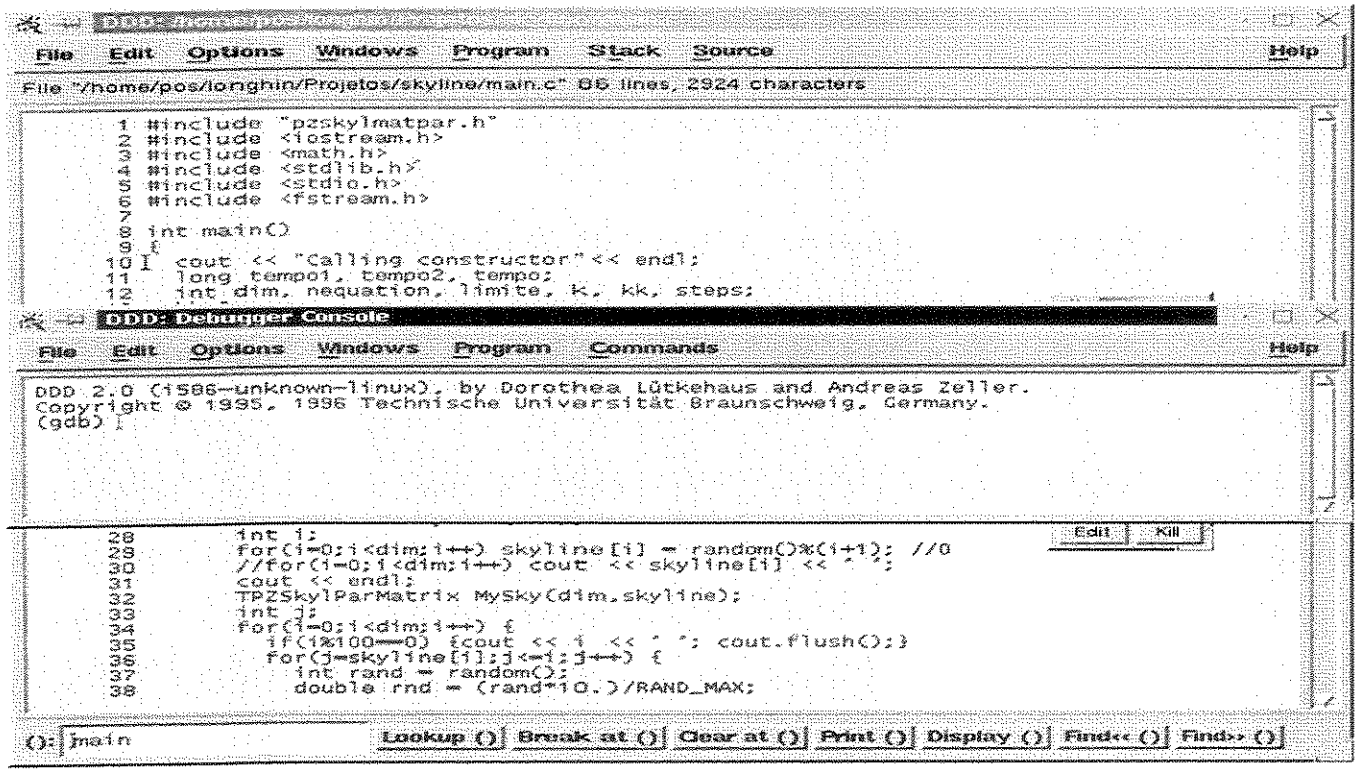


Figura C.4: Depurador DDD

Há aplicativos criados para facilitar estas chamadas do g++. É distribuído juntamente com o Linux o *make*, que utiliza um arquivo chamado *makefile* contendo as informações necessárias para a execução do g++. A ferramenta make é fundamental no desenvolvimento de grandes projetos.
O g++ é responsável não só pela compilação do código mas também pela linkagem do executável.

Depurador

No ambiente Linux a tarefa de depuração é executada pelo software DDD. Ver Figura C.4, p 84.

C.1.3 Aplicativos Multi-plataformas

Foram utilizados alguns aplicativos que são disponibilizados nas duas plataformas utilizadas. Um deles é o software Together para desenvolvimento com UML (*Unified Modeling Language*), o tópico 2.2 (pag. 14) trata de UML. Um outro aplicativo também utilizado foi o DoxyGen, para geração de documentação.

Together

Há programas que geram automaticamente toda implementação básica de um projeto desenvolvido com a *UML* numa linguagem específica. O Together é um software que executa tal tarefa escrevendo o código em *C++* ou *JAVA*.

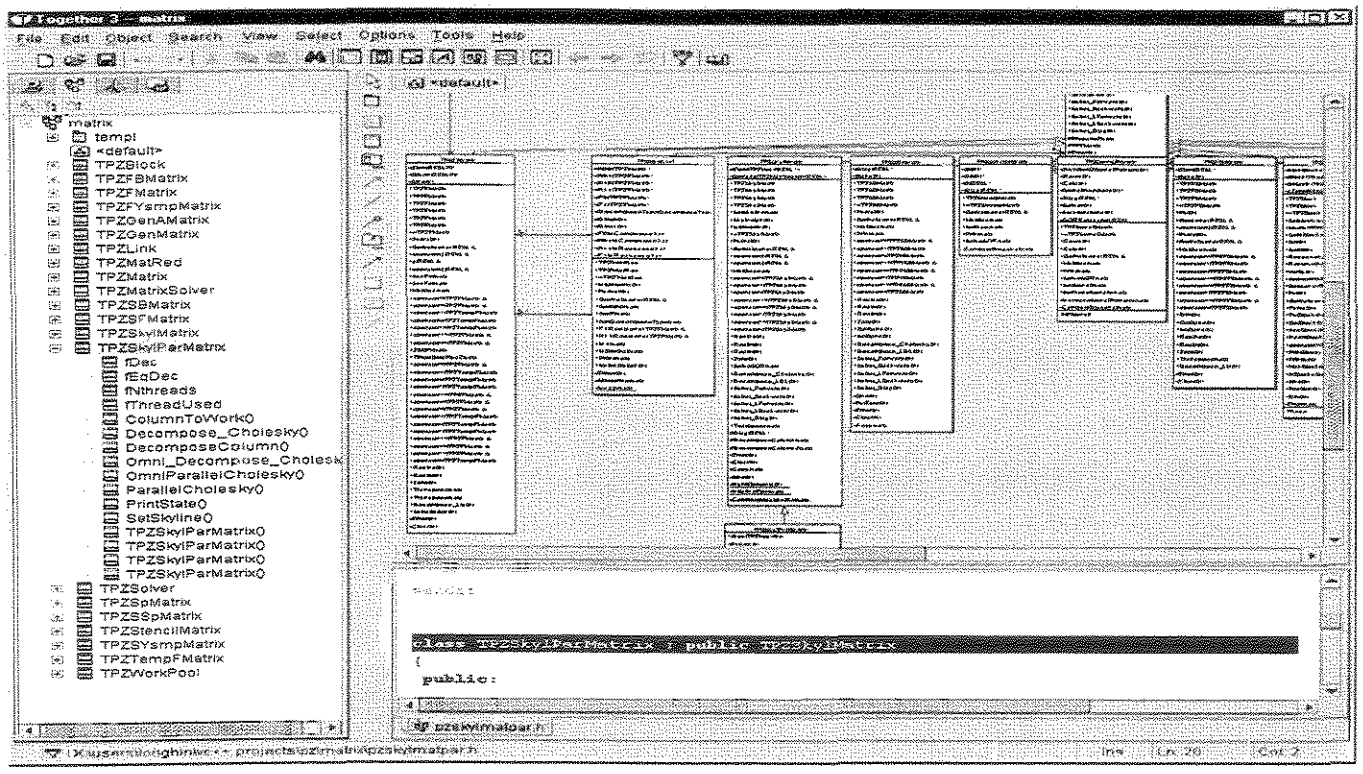


Figura C.5: Desenvolvimento com Together

Uma vez que um projeto é desenvolvido com *UML*, basta ao programador implementar a parte funcional do algoritmo. O programador não precisa ater-se a detalhes de declarações e etc.. A maioria destes pontos será supervisionada pelo aplicativo.

Esta filosofia tem sido importante durante o desenvolvimento deste projeto, visto que o ambiente *PZ* é base de trabalho de um grupo de programadores. As documentações geradas pelo *Together* são fundamentais no entendimento dos vários módulos envolvidos.

O Together foi desenvolvido por Peter Coad, junto a *OI* (*Object International*) atualmente sob nome *TogetherSoft Corporation* [11] [1] [18], e pode ser encontrada em *www.togethersoft.com.* O aplicativo possui facilidades de edição que transformam o trabalho de implementação de declarações, heranças e etc., numa tarefa simples e de fácil entendimento. Ver Figura C.5, p. 85. A *TogetherSoft* disponibiliza sua versão completa para desenvolvimento acadêmico, sendo esta a versão utilizada neste projeto.

Traduz-se com UML, descrições de estruturas e responsabilidades para uma linguagem gráfica. Esta linguagem baseia-se em Figuras e diagramas. O mais utilizado neste projeto é o diagrama de classes, onde define-se responsabilidades e relacionamentos das classes.

Aplicativo Doxygen

O sucesso de projetos corporativos pode ser dependente da qualidade da documentação dos códigos envolvidos em cada pacote. A utilização de técnicas como UML agrega muita qualidade aos códigos produzidos, esta qualidade deve porém, ser transmitida ao usuário de maneira eficaz. Mais interessante que um código documentado, é uma documentação, por exemplo, em formato

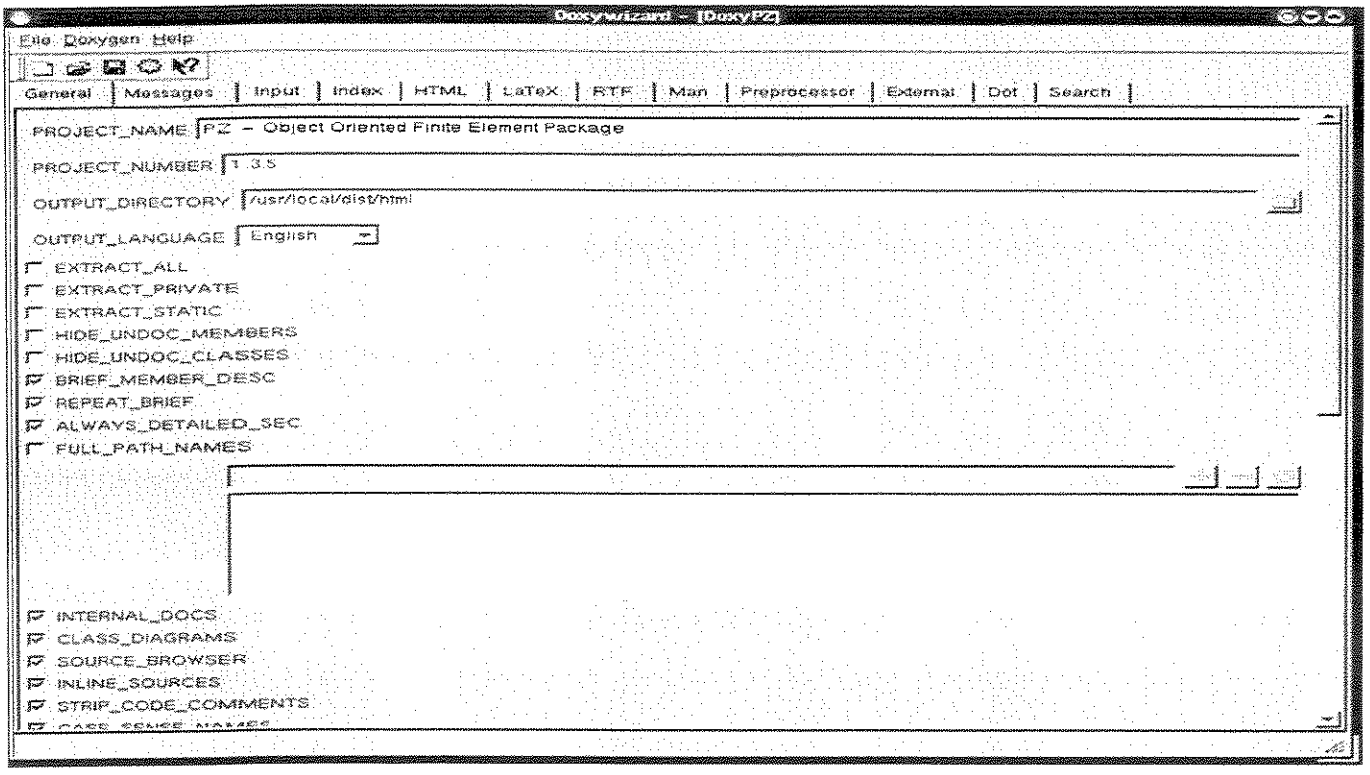


Figura C.6: DoxyWizard, uma GUI para o doxygen.

HTML ou mesmo um documento com a qualidade de um LATEX, contendo toda a interpretação da documentação existente no código fonte.

Existe uma série do softwares que realizam a tarefa de interpretar o que está documentado no código e traduzi-los para formatos Latex, HTML, man, rtf e etc. Neste projeto utilizamos o Doxygen para geração de toda documentação do pacote PZ. O site http://bohemia.fec.unicamp.br/pz contém a documentação gerada pelo aplicativo Doxygen.

Toda equipe envolvida no desenvolvimento do projeto PZ vem utilizando o citado site como guia de consulta. O resultado tem sido muito interessante, uma vez que vários usuários trocam informações sobre o mesmo pacote, isso melhora ainda mais a qualidade da documentação.

A Figura C.6 (pag 86) apresenta uma interface gráfica para o Doxygen, o DoxyWizard. Ambos os aplicativos estão distribuídos sob licença GPL.

Referências Bibliográficas

[1] Sinan Si Alhir. *UML in a nutshell, a desktop quick reference.* O'Reilly & Associates Inc., first edition edition, 1998.

[2] Dan Appleman. *VISUAL BASIC 5.0 Programmer's Guide to the WIN32 API.* Ziff-Davis Press, 200 Tamal Plaza, Suite 101, Corte Madera, CA 94925, 1997.

[3] Graham Birtwistle. *SIMULA begin.* Auerbach, 1973.

[4] Grady Booch. *Object oriented design : with applications.* Benjamin & Cummings, Redwood, Calif., 1991.

[5] Jorge L. D. Calle. *Introdução a Linguagem C++, Programação Orientada para Objetos.* CENAPAD - UNICAMP, Novembro 1999.

[6] K.V. Camarda and M.A. Stadtherr. Frontal solvers for process engineering: Local row ordering strategies. In *Computers & Chemical Engineering*, pages 333–341. Computers & Chemical Engineering, Computers & Chemical Engineering, 1998.

[7] Murray R. Cantor. *Object-Oriented Project Management with UML.* Wiley Computer Publishing. John Wiley & Sons, Inc., New York, first edition, 1998.

[8] DEC Systems Research Center. An introduction to programming with threads. Technical report, DEC Systems Research Center, Palo Alto, CA, 1989. Research Report 35.

[9] T.R. Chandrupatla and K.J. Berry. Advances in engineering software. In *Frontal Program for a PC-Based Solution of Unsymmetric Matrices Using a Buffered Pivot Search*, pages 191–199. ADVANCES IN ENGINEERING SOFTWARE, ADVANCES IN ENGINEERING SOFTWARE, December 1996.

[10] Peter Coad and Edward Yourdon. *Object-oriented analysis.* Prentice-Hall, Englewood Cliffs, 1991.

[11] Peter Coad and Edward Yourdon. *Object-Oriented Design.* 1991.

[12] Gary Cornell and Dave Jezak. *ActiveX: Visual Basic 5 Control Creation Edition.* Upper Saddle River, NJ 07458, 1997.

[13] Intel Corporation. *MKL 3.2 User's Manual and Documentation.* developer.intel.com/VTune/perflibst/MKL, 2000.

[14] OJ Dahl and K Nygaard. Simula - an algol-based simulation language. In *Communications of the ACM*, 1515 Broadway, New York, NY 10036, 1966. Assoc. Computing Machinery.

[15] Jack J. Dongarra, Ian S. Duff, Danny C. Sorensen, and Henk A. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Society for Industrial and Applied Mathematics., 3600 University City Science Center, Philadelphia, Pennsylvania 19104-2688, second edition, 1993.

[16] I.S. Duff and J.A. Scott. The design of a new frontal code for solving sparse, unsymetric systems. In *ACM Transactions on Mathematical Software*, pages 30–45. ACM Transactions on Mathematical Software, ACM Transactions on Mathematical Software, March 1996.

[17] Richard L. Burden & Douglas J. Faires. *Numerical Analysis.* Brooks & Cole, 6 ed. edition, 1997.

[18] José Davi Furlan. *Modelagem de Objetos Através da UML - The Unified Modeling Language.* Makron Books do Brasil Ltda., 1998.

[19] P. Geng, J.T. Oden, and R.A. VandeGeijn. A parallel multifrontal algorithm and its implementation. In *Computer Methods in Applied Mechanics and Engineering*, pages 289–301. Computer Methods in Applied Mechanics and Engineering, Computer Methods in Applied Mechanics and Engineering, 1997.

[20] A George and MT Heath. Parallel cholesky factorization on a shared-memory multiprocessor. In *Linear Algebra and its Application*, pages 165–187, University Waterloo, Dept. Comp. Science, Waterloo N2L 3G1, Ontario, Canada, May 1986. Elsevier Science Inc.

[21] Adele Goldberg. *Smalltalk-80 : The Interactive Programming Environmen.* Addison-Wesley Publishing, 1983.

[22] Stephen Guty, editor. *Advanced Microsoft Visual BASIC 6.0.* The Mandelbrot Set. Microsoft Press, second edition edition, 1998. Foreword by Sean Alexander.

[23] Sammule P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall software series, Englewoods Cliffs, New Jersey, 07632, second edition.

[24] Ivor Horton. *Beginning Visual C++ 6.* Wrox Press Ltd., 1998.

[25] IEEE. *Portable Operating System Interface (POSIX) Threads Extension*, 1994.

[26] Bruce Irons and N.G. Shrive. *Numerical Methods in Engineering and Applied Science.* Chichester, 1987.

[27] Isaak, Lewis, Thompson, and Straub. *Open Systems Handbook.* IEEE Standards Press, 1994.

[28] Ivar Jacobson. *Object-oriented software engineering.* Addison-Wesley, Reading, Mass, 1993.

[29] Ted Kaehler and Dave Patterson. *A Taste of Smalltalk.* WW Norton & Co, May 1986.

[30] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minessota, Department of Computer Science and Army High Performance Computing Research Center, Minneapolis, MN 55455, November 5 1997.

[31] Glen Krasner. *Smalltalk-80 : Bits of History, Words of Advice.* Addison-Wesley Publishing, 1983.

[32] M. Lesoinne, C. Farhat, and M. Geradin. Parallel vectors improvements of frontal method. In *International Journal for Numerical Methods in Engineering*, pages 1267–1281. International Journal for Numerical Methods in Engineering, International Journal for Numerical Methods in Engineering, October 1991.

[33] Stanley B. Lippman. *C++ Primer.* Addison-Wesley Publishing Company, second edition.

[34] Mike McKelvy, Ronald Martisen, and Jeff Webb. *Using Visual Basic.* Joseph B. Wikert, 201 W. 103rd Street, Indianapolis, IN, 46290, 1997.

[35] Mark Nelson. *C++ Programmer's Guide to Standard Template Library.* IDG Books Worldwide, Inc, first edition, 1995.

[36] Meilir PageJones. *Fundamentos do Projeto Orientado a Objeto com UML.* MAKRON Books ltda., 2001. From the Original Fundamentals of Object-Oriented Design in UML.

[37] Wilhelm Quarterman. *UNIX, POSIX, and Open Systems.* Addison Wesley, 1993.

[38] Wendy Rauch. *Distributed Open Systems Engineering.* Wiley Computer Publishing, 1996.

[39] Tristan Richardson. *The OMNI Thread Abstraction.* Olivetti Research Laboratory Cambridge, March 1997.

[40] James Rumbaugh. *Object-oriented modeling and design.* Prentice-Hall, Englewood Cliffs, 1991.

[41] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology. Addison-Wesley, Reading, Massachusetts, first edition, 1999.

[42] Stephen J. Mellor Sally Shlaer. *Object-oriented systems analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[43] Robert Sebesta. *Concepts of Programming Languages.* Addison-Wesley Publishing, CA., 1996.

[44] J. L. F. Serra. Solução de sistema de equações por cholesky. Universidade Estadual de Campinas, Cid. Universitária Zeferino Vaz, 1996. Departamento de Estruturas.

[45] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, 2nd edition, 1991.

[46] Bjarne Stroustrup. *Design and Evolution of C++.* Addison-Wesley, 1994.

[47] SunSoft. *Guide to Multi-thread Programming.* SUN Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043 USA, 1993.

[48] Sloan S.W. A fortran program for profile and wavefront reduction. In *International Journal for Profile and Wavefront Reduction,* volume 28, pages 2651–2679, 1989.

[49] Golub H.G. & van Loan C.F. *Matrix Computations.* The Johns Hopkins University Press, West 40th Street, Baltimore, MAryland 21211, fifth edition, 1987.

[50] D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured finite element meshes. In *International Journal for Numerical Methods in Engineering*, pages 433–450. International Journal for Numerical Methods in Engineering, International Journal for Numerical Methods in Engineering, February 1995.

[51] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[52] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[53] Todd Veldhuizen. Rapid linear algebra in C++. *Dr. Dobb's Journal*, August 1996.

[54] P. De Vincenzo, I. Klapka, and M. Geradin. Performance of static and dynamic solvers on a sub-domains object oriented finite element architecture. In B. H. V. Topping, editor, *Advances in Computational Mechanics with High Performance Computing*, pages 169–179, 10A Saxe-Coburg Place, Edinburgh, EH3 5BR, UK, 1998. Civil Comp Press.

[55] Z.Q. You, Z.W. Jiang, Y.S. Sun, and L. Udpa. Application of the substructured-frontal method for repeated solution of large sparse-matrix equations to field problems. In *IEEE Transactions on Magnetics*, pages 326–329. IEEE Transactions on Magnetics, IEEE Transactions on Magnetics, January 1988.

[56] D Zheng and TYP Chang. Parallel cholesky method on mimd with shared-memory. In *Computers & Structures*, pages 25–38, The Boulevard, Langoford Lane, Kidlington, Oxford, England, July 1995. Pergamon-Elsevier Science LTD.

[57] S.E. Zitney, J. Mallya, T.A. Davis, and M.A. Stadtherr. Multifrontal vs frontal techniques for chemical process simulation on supercomputers. In *Computers & Chemical Engineering*, pages 641–646. COMPUTERS & CHEMICAL ENGINEERING, COMPUTERS & CHEMICAL ENGINEERING, June-July 1996.

[58] Fred Zlotnick. *The POSIX Standard, a Programmer's Guide.* Benjamin Cummings, 1991.

Parallel Frontal Solver Reference Manual

1.0.1

Generated by Doxygen 1.2.14

Fri Jun 14 12:53:55 2002

Contents

Chapter 1

Parallel Frontal Solver Hierarchical Index

1.1 Parallel Frontal Solver Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Parallel Frontal Solver Compound Index

2.1 Parallel Frontal Solver Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Parallel Frontal Solver Class Documentation

3.1 TPZEqnArray Class Reference

`#include <tpzeqnarray.h>`

Public Methods

- void **SetNonSymmetric** ()
- void **SetSymmetric** ()
- int **IsSymmetric** ()
- **TPZEqnArray** ()
- **~TPZEqnArray** ()
- void **EqnForward** (TPZFMatrix &F, DecomposeType dec)
- void **EqnBackward** (TPZFMatrix &U, DecomposeType dec)
- void **Reset** ()
- void **BeginEquation** (int eq)
- void **AddTerm** (int col, REAL val)
- void **EndEquation** ()
- void **Read** (char *inputfile)
- void **Write** (char *outputfile)
- void **Print** (const char *name, ostream &out)
- void **Write** (FILE *outputfile)
- void **Read** (FILE *inputfile)

Static Public Methods

- void **main** ()

3.1.1 Detailed Description

It is an equation array, generally in its decomposed form.

Would be saved and read from disk.

It contains a forward and a backward method.

It is sparce, symmetric or not.

Definition at line 43 of file tpzeqnarray.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 TPZEqnArray::TPZEqnArray ()

Simple constructor

Definition at line 26 of file tpzeqnarray.c.

```
26                          : fIndex(0), fEqValues(0), fEqStart(0), fEqNumber(0) {
27         fEqStart.Push(0);
28         fNumEq=0;
29         fLastTerm=0;
30       fSymmetric=EIsUndefined;
31
32 }
```

3.1.2.2 TPZEqnArray::~TPZEqnArray ()

Simple desctructor

Definition at line 23 of file tpzeqnarray.c.

```
23                          {
24
25 }
```

3.1.3 Member Function Documentation

3.1.3.1 void TPZEqnArray::AddTerm (int *col*, REAL *val*) [inline]

Add a term to the current equation

Parameters:

col The collumn position of val

val The value beeing added itself

Definition at line 91 of file tpzeqnarray.h.

Referenced by main.

```
95      {
96              fIndex.Push(col);
97              fEqValues.Push(val);
98              fLastTerm++;
99      }
```

3.1.3.2 void TPZEqnArray::BeginEquation (int *eq*)

It starts an equation storage

Parameters:

eq Indicates what equation is beeing added to the stack

Definition at line 67 of file tpzeqnarray.c.

Referenced by main.

```
67                                      {
68          fEqNumber.Push(eq);
69          fNumEq++;
70 }
```

3.1.3.3 void TPZEqnArray::EndEquation ()

Ends the current equation

Definition at line 58 of file tpzeqnarray.c.

Referenced by main.

```
58                                {
59          fEqStart.Push(fLastTerm);
60 }
```

3.1.3.4 void TPZEqnArray::EqnBackward (TPZFMatrix & *U*, DecomposeType *dec*)

Backward substitution on equations stored in EqnArray

Parameters:

U Matrix to execute a Forward substitutio

dec Type of decomposition, depends on what method was used in its decomposition

Definition at line 81 of file tpzeqnarray.c.

References IsSymmetric.

Referenced by TPZFrontSym::main, TPZFrontNonSym::main, and TPZFront::main.

```
81                                                                    {
82      int n;
83      if(IsSymmetric()==EIsSymmetric){
84         for(n=fNumEq-1; n>=0; n--) {
85                 int index = fEqStart[n];
86
87     //          if(fEqStart.NElements()==n+1){
88     //                  fEqStart.Expand(n+2);
89     //          }
90                 int last_term = fEqStart[n + 1];
91                 REAL acum = 0.;
92                 int i;
93                 for(i = index + 1; i < last_term; i++) acum -= U(fIndex[i],0) * fEqValues[i];
94
96              U(fIndex[index],0) += acum;
97
99              if(dec == ECholesky){
100                  U(fIndex[index],0) /= fEqValues[index];
101                  }
102         }
103         }else if(IsSymmetric()==EIsNonSymmetric){
104         for(n=fNumEq-2; n>=0; n-=2) {
105             int index = fEqStart[n];
106
107        //               if(fEqStart.NElements()==n+1){
108        //                       fEqStart.Expand(n+2);
109        //               }
110             int last_term = fEqStart[n + 1];
111             REAL acum = 0.;
112             int i;
113             for(i = index + 1; i < last_term; i++) acum -= U(fIndex[i],0) * fEqValues[i];
114
116             U(fIndex[index],0) += acum;
117
119             if(dec == ECholesky || dec == ELU){
120                 U(fIndex[index],0) /= fEqValues[index];
```

```
121             }
122         }
123     }
124 }
```

3.1.3.5 void TPZEqnArray::EqnForward (TPZFMatrix & *F*, DecomposeType *dec*)

Forward substitution on equations stored in EqnArray

Parameters:

F Matrix to execute a Forward substitution on

dec Type of decomposition, depends on what method was used in its decomposition

Definition at line 126 of file tpzeqnarray.c.

References IsSymmetric.

Referenced by TPZFrontSym::main, TPZFrontNonSym::main, TPZFront::main, and main.

```
126                                                                 {
127   int j;
128   if(IsSymmetric()==EIsSymmetric){
129
130     for(j=0; j<fNumEq; j++) {
131       int index = fEqStart[j];
132       if(!fEqValues[index]){
133           cout << "Diagonal Value = 0 >> Aborting on Index = " << index << " Equation = " << j << endl;
134           exit(-1);
135       }
136       int last_term;
137       if(j==fNumEq-1){
138         last_term=fEqValues.NElements();
139       }else last_term = fEqStart[j+1];
140                 //if(fEqStart.NElements()
141
142       int num_terms = last_term - index;
144       if(dec==ECholesky || dec==ELU) {
145           F(fIndex[index],0) /= fEqValues[index];
146       }
148       REAL udiag = F(fIndex[index],0);
149
150
151       int i;
152           //+1 ou +2
153       for(i = index + 1; i < last_term; i++) F(fIndex[i],0) -= udiag * fEqValues[i];
155       if(dec == ELDLt) F(fIndex[index],0) /= fEqValues[index];
156     }
```

```
157   }else if(IsSymmetric()==EIsNonSymmetric){
158     for(j=1; j<fNumEq; j+=2) {
159       int index = fEqStart[j];
160
161       if(!fEqValues[index]){
162           cout << "Diagonal Value = 0 >> Aborting on Index = " << index << " Equation = " <<
163           exit(-1);
164       }
165
166       int last_term;
167       if(j==fNumEq-1){
168         last_term=fEqValues.NElements();
169       }else last_term = fEqStart[j+1];
170          //if(fEqStart.NElements()
171
172       int num_terms = last_term - index;
174       if(dec==ECholesky || dec==ELU) {
175                           F(fIndex[index],0) /= fEqValues[index];
176       }
178       REAL udiag = F(fIndex[index],0);
179
180
181       int i;
182       for(i = index + 1; i < last_term; i++) F(fIndex[i],0) -= udiag * fEqValues[i];
184       if(dec == ELDLt) F(fIndex[index],0) /= fEqValues[index];
185     }
186   }
187 }
```

3.1.3.6 int TPZEqnArray::IsSymmetric ()

Gets the symetry situation of EqnArray

Definition at line 17 of file tpzeqnarray.c.

Referenced by EqnBackward, and EqnForward.

```
17                              {
18     return fSymmetric;
19 }
```

3.1.3.7 void TPZEqnArray::main () [static]

Static main function for testing

Definition at line 249 of file tpzeqnarray.c.

References AddTerm, BeginEquation, EndEquation, EqnForward, Print, and Reset.

```
250 {
251         char  filename[20];
252 
253         cout << "Entre o nome do Arquivo\n";
254         cin >> filename;
255         ofstream output(filename,ios::app);
256 
257         TPZFMatrix MatrixA(10,10);
258         int i, j;
259         for(i=0;i<10;i++) {
260                 for(j=i;j<10;j++) {
261                         int random = rand();
262                         double rnd = (random*10.)/RAND_MAX;
263                         MatrixA(i,j)=rnd;
264                         MatrixA(j,i)=MatrixA(i,j);
265                         if(i==j) MatrixA(i,j)=6000.;
266                 }
267         }
268 
269 
270         MatrixA.Print("Teste 1");
271         MatrixA.Decompose_Cholesky();
272         MatrixA.Print("Decomposta");
273 
274         TPZEqnArray Test;
275         Test.Reset();
276         for(i=0;i<10;i++) {
277         Test.BeginEquation(i);
278         for(j=i;j<10;j++) {
279           Test.AddTerm(j, MatrixA(i,j));
280         }
281         Test.EndEquation();
282         }
283 
284         TPZFMatrix rhs(10,1), rhs2(10,1);
285         //Inicializar rhs:
286         rhs.Zero();
287         rhs(0,0) = 1.;
288         rhs2=rhs;
289 
290         Test.Print("Result", cout);
291 
292         MatrixA.Subst_Forward(&rhs);
293         DecomposeType decType = ECholesky;
294         Test.EqnForward(rhs2, decType);
295 
296         rhs.Print("FMatrix Decomposition");
297         rhs2.Print("FrontalMatrix Decomposition");
298 
299 }
```

3.1.3.8 void TPZEqnArray::Print (const char * *name*, ostream & *out*)

It prints all terms stored in TPZEqnArray

Definition at line 33 of file tpzeqnarray.c.

Referenced by TPZFrontSym::main, TPZFrontNonSym::main, TPZFront::main, and main.

```
34 {
35   if(name) out << name << endl;
36   int i,j;
37   for(i=0;i<fNumEq;i++){
38     int aux_limit;
39
40     if(i==fNumEq-1){
41       aux_limit=fEqValues.NElements();
42     }else aux_limit=fEqStart[i+1];
43     for(j=fEqStart[i]; j<aux_limit ; j++) {
44       out << "col = " << fIndex[j] << "   ";
45     }
46     out << endl;
47     for(j=fEqStart[i]; j<aux_limit ; j++) {
48       out << fEqValues[j] << "  ";
49     }
50     out << endl;
51     out << endl;
52   }
53 }
```

3.1.3.9 void TPZEqnArray::Read (FILE * *inputfile*)

Reads from binary file generated by 'WriteBinary' Also receives position and FILE for 'C' fread() function execution

Definition at line 216 of file tpzeqnarray.c.

```
216                                    {
218         fread(&fNumEq,sizeof(int),1,inputfile);
219         //cout << ftell(inputfile) << endl;
221         fread(&fLastTerm,sizeof(int),1,inputfile);
222         //cout << ftell(inputfile) << endl;
223         int aux=0;
225         fread(&aux,sizeof(int),1,inputfile);
226         fEqStart.Resize(aux);
227         fread(&fEqStart[0],sizeof(int), aux ,inputfile);
228
230         fread(&aux,sizeof(int),1,inputfile);
231         fEqNumber.Resize(aux);
232         fread(&fEqNumber[0],sizeof(int), aux ,inputfile);
```

```
233
235         fread(&aux,sizeof(int),1,inputfile);
236         fIndex.Resize(aux);
237         fread(&fIndex[0],sizeof(int), aux ,inputfile);
238
240         fread(&aux,sizeof(int),1,inputfile);
241         fEqValues.Resize(aux);
242         fread(&fEqValues[0],sizeof(REAL), aux ,inputfile);
243
244         this->fSymmetric = EIsSymmetric;
245         if(fNumEq && fNumEq%2==0 && fEqNumber[0]==fEqNumber[1]) fSymmetric = EIsNonSymmetric;
246 }
```

3.1.3.10 void TPZEqnArray::Read (char * *inputfile*)

Reads from disk

Definition at line 57 of file tpzeqnarray.c.

```
57 {}
```

3.1.3.11 void TPZEqnArray::Reset ()

Resets data structure

Definition at line 71 of file tpzeqnarray.c.

Referenced by TPZFrontSym::DecomposeEquations, TPZFrontNonSym::DecomposeEquations, and main.

```
71                          {
72         fEqStart.Resize(0);
73         fEqStart.Push(0);
74         fEqValues.Resize(0);
75         fIndex.Resize(0);
76         fNumEq=0;
77         fLastTerm=0;
78      fSymmetric=EIsUndefined;
79
80 }
```

3.1.3.12 void TPZEqnArray::SetNonSymmetric ()

Sets EqnArray to a non symmetric form

Definition at line 14 of file tpzeqnarray.c.

Referenced by TPZFrontNonSym::DecomposeEquations.

```
14                                  {
15     fSymmetric=EIsNonSymmetric;
16 }
```

3.1.3.13 void TPZEqnArray::SetSymmetric ()

Sets fSymmetric to the symmetric value

Definition at line 20 of file tpzeqnarray.c.

Referenced by TPZFrontSym::DecomposeEquations.

```
20                                  {
21     fSymmetric = EIsSymmetric;
22 }
```

3.1.3.14 void TPZEqnArray::Write (FILE * *outputfile*)

Writes to a file in binary mode. Used by FileEqnStorage Receives FILE and position to execute 'C' fwrite function

Definition at line 189 of file tpzeqnarray.c.

```
189                                             {
191         fwrite(&fNumEq,sizeof(int),1,outputfile);
192         //cout << ftell(outputfile) << endl;
194         fwrite(&fLastTerm,sizeof(int),1,outputfile);
195         //cout << ftell(outputfile) << endl;
197         int aux = fEqStart.NElements();
198         fwrite(&aux,sizeof(int),1,outputfile);
199         fwrite(&fEqStart[0],sizeof(int), aux ,outputfile);
200
202         aux = fEqNumber.NElements();
203         fwrite(&aux,sizeof(int),1,outputfile);
204         fwrite(&fEqNumber[0],sizeof(int), aux ,outputfile);
205
207         aux = fIndex.NElements();
208         fwrite(&aux,sizeof(int),1,outputfile);
209         fwrite(&fIndex[0],sizeof(int), aux ,outputfile);
210
212         aux = fEqValues.NElements();
213         fwrite(&aux,sizeof(int),1,outputfile);
214         fwrite(&fEqValues[0],sizeof(REAL), aux ,outputfile);
215 }
```

3.1.3.15 void TPZEqnArray::Write (char * *outputfile*)

Writes on disk

Definition at line 56 of file tpzeqnarray.c.

```
56 {}
```

The documentation for this class was generated from the following files:

- tpzeqnarray.h
- tpzeqnarray.c

3.2 TPZFileEqnStorage Class Reference

`#include <TPZFileEqnStorage.h>`

Public Methods

- void **ReOpen** ()
- void **ReadBlockPositions** ()
- void **FinishWriting** ()
- void **OpenGeneric** (char option, const char *name)
- **TPZFileEqnStorage** ()
- void **Store_Old** (int ieq, int jeq, const char *name)
- **~TPZFileEqnStorage** ()
- **TPZFileEqnStorage** (char option, const char *name)
- void **AddEqnArray** (**TPZEqnArray** *EqnArray)
- void **Print** (const char *name, ostream &out)
- void **Reset** ()
- void **Backward** (TPZFMatrix &f, DecomposeType dec) const
- void **Forward** (TPZFMatrix &f, DecomposeType dec) const
- void **Store** (int ieq, int jeq, const char *name)
- void **WriteHeaders** ()
- char * **GetStorage** ()

Static Public Methods

- void **main** ()

3.2.1 Detailed Description

Has the same purpouse of EqnStack but stores the EqnArrays

in a different form (binary files).

It has methods for operating over a set of equations.

The arrays of equations are in the form of a binary files of EqnArrays.

Definition at line 18 of file TPZFileEqnStorage.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 TPZFileEqnStorage::TPZFileEqnStorage ()

Simple constructor

Definition at line 338 of file TPZFileEqnStorage.c.

```
339 {
340      strcpy(filenamestorage, "/tmp/binary_frontalXXXXXX");
341      char * tempname;
342      //tempname = mktemp(filenamestorage);
343      mkstemp(filenamestorage);
344
345      cout << "Temporary file name " << fFileName << endl;
346      cout.flush();
347
348      fCurBlockPosition = -1;
349         fNumBlocks=0;
350         fCurrentBlock=0;
351         fNumHeaders=20;
352 //      SetBlockSize(10);
353         //fBlockPos.Resize(fNumHeaders);
354
355
356         strcpy(fFileName,filenamestorage);
357
358         fIOStream = fopen(fFileName,"wb"); //open for writing
363         int zero = 0;
364         fwrite(&fNumHeaders,sizeof(int),1,fIOStream);
365         //cout << ftell(fIOStream) << endl;
366         fwrite(&zero,sizeof(int),1,fIOStream);
367         //cout << ftell(fIOStream) << endl;
368         //fCurBlockPosition = ftell(fIOStream);
369
370
371 }
```

3.2.2.2 TPZFileEqnStorage::~TPZFileEqnStorage ()

Simple destructor

Definition at line 267 of file TPZFileEqnStorage.c.

```
268 {
269      remove(fFileName);
270 }
```

3.2.2.3 TPZFileEqnStorage::TPZFileEqnStorage (char *option*, const char * *name*)

Constructor option can assume "w" or "r" for writeing and reading respectively

Parameters:

option 'w' means writing and 'r' reading

name the file name to print to

Definition at line 237 of file TPZFileEqnStorage.c.

```
238 {
239         fCurBlockPosition = -1;
240         fNumBlocks=0;
241         fCurrentBlock=0;
242         fNumHeaders=20;
243         strcpy(fFileName,name);
244         if(option=='r'){
245                 fIOStream = fopen(fFileName,"rb"); //open for reading
250                 fread(&fNumHeaders,sizeof(int),1,fIOStream);
251                 fread(&fNumBlocks,sizeof(int),1,fIOStream);
252                 ReadBlockPositions();
253         }else if(option=='w'){
254                 fIOStream = fopen(fFileName,"wb"); //open for writing
259                 int zero = 0;
260                 fwrite(&fNumHeaders,sizeof(int),1,fIOStream);
261                 fwrite(&zero,sizeof(int),1,fIOStream);
262                 //fCurBlockPosition = ftell(fIOStream);
263         }
264
265 }
```

3.2.3 Member Function Documentation

3.2.3.1 void TPZFileEqnStorage::AddEqnArray (TPZEqnArray * *EqnArray*)

Adds an EqnArray

Parameters:

EqnArray EqnArray added to the binary file

Definition at line 199 of file TPZFileEqnStorage.c.

```
200 {
201
202
203         if(fCurrentBlock%(fNumHeaders-1)==0) {
204                 WriteHeaders();
205 //              fBlockPos.Push(fBlockPos[fCurrentBlock-1]+sizeof(long int));
206
207         }else if(fCurrentBlock!=0){
208                 fBlockPos.Push(fBlockPos[fCurrentBlock-1]+sizeof(long int));
209         }
210
211         EqnArray->Write(fIOStream);
212
```

```
214         long int nextaddress=ftell(fIOStream);
215
216
221         //fseek(fIOStream,fCurBlockPosition+sizeof(long int),SEEK_SET);
222         fseek(fIOStream,fBlockPos[fCurrentBlock]+sizeof(long int),SEEK_SET);
223         fCurBlockPosition=ftell(fIOStream);
224         fwrite(&nextaddress,sizeof(long int),1,fIOStream);
225
226
227         fseek(fIOStream,nextaddress,SEEK_SET);
228 //      fBlockPos.Push(fCurBlockPosition);
229
230         fCurrentBlock++;
231 //      fSubBlockIndex++;
232
233 //      cout << "NumBlocks " << fNumBlocks << endl;
234 //      cout << "NumHeaders " << fNumHeaders << endl;
235 }
```

3.2.3.2 void TPZFileEqnStorage::Backward (TPZFMatrix & *f*, DecomposeType *dec*) const

Executes a Backward substitution

Parameters:

f Full matrix already decomposed.

dec Decomposition type of f

Definition at line 155 of file TPZFileEqnStorage.c.

```
156 {
157     #ifdef _DEBUG
158         cout << "Inside TPZFileEqnStorage::Backward" << endl;
159         cout << "fBlockPos.NElements() = " << fBlockPos.NElements() << endl;
160     #endif
161
162        TPZEqnArray REqnArray;
163        int i;
164        for(i=fBlockPos.NElements()-1;i>=0;i--){
165                if (fBlockPos[i]) {
166                     if(!(i%10)) cout << "*";
167                     if(!(i%100)) cout << i << endl;
168                        fseek(fIOStream,fBlockPos[i],SEEK_SET);
169                        long int position;
170                        fread(&position,sizeof(long int),1,fIOStream);
171                        fseek(fIOStream,position,SEEK_SET);
172                        REqnArray.Read(fIOStream);
173                        REqnArray.EqnBackward(f,dec);
174                }
175        }
176
177 }
```

3.2.3.3 void TPZFileEqnStorage::FinishWriting ()

Method used for binary input/output

Definition at line 456 of file TPZFileEqnStorage.c.

```
457 {
458         fseek(fIOStream,sizeof(int),SEEK_SET);
459         //cout << "Second fseek " << ftell(fIOStream) << endl;
460         //fwrite(&fNumBlocks,sizeof(int),1,fIOStream);
461         fwrite(&fNumBlocks,sizeof(int),1,fIOStream);
462         fclose(fIOStream);
463         fIOStream = 0;
464 }
```

3.2.3.4 void TPZFileEqnStorage::Forward (TPZFMatrix & *f*, DecomposeType *dec*) const

Executes a Forward substitution

Parameters:

f Full matrix already decomposed.

dec Decomposition type of f

Definition at line 119 of file TPZFileEqnStorage.c.

```
120 {
121   #ifdef _DEBUG
122      cout << "Inside TPZFileEqnStorage::Forward" << endl;
123      cout << "fBlockPos.NElements() = " << fBlockPos.NElements() << endl;
124   #endif
125
126   //if(!fIOStream) SetFileName(fFileName);
127   TPZEqnArray REqnArray;
128   int i;
129   for(i=0;i<fBlockPos.NElements();i++) {
130     if (fBlockPos[i]) {
131      if(!(i%10)) cout << "*";
132      if(!(i%100)) cout << i << endl;
133
134       if(fseek(fIOStream,fBlockPos[i],SEEK_SET)){
135           cout << "fseek fail on Element " << i << " Position " << fBlockPos[i] << endl;
136           cout.flush();
137       }
138       long int position;
139       if(!fread(&position,sizeof(long int),1,fIOStream)){
140           cout << "fread fail on Element " << i << " Position " << position << endl;
141           cout << "EOF " << feof(fIOStream) << endl;
142           cout << "Error Number " << ferror(fIOStream) << endl;
143           cout.flush();
```

```
144       }
145       if(fseek(fIOStream,position,SEEK_SET)){
146           cout << "fseek fail on Element " << i << " Position " << position << endl;
147           cout.flush();
148       }
149
150       REqnArray.Read(fIOStream);
151       REqnArray.EqnForward(f,dec);
152     }
153   }
154 }
```

3.2.3.5 char * TPZFileEqnStorage::GetStorage ()

Type of Storage

Definition at line 482 of file TPZFileEqnStorage.c.

```
482 {return "File Storage";}
```

3.2.3.6 void TPZFileEqnStorage::main () [static]

Static main for testing

Definition at line 282 of file TPZFileEqnStorage.c.

```
283 {
284         int Loop_Limit=24;
285         //cout << "Loop_Limit <";
286         //cin >> Loop_Limit;
287         char * filename;
288         filename = "testbinary.txt\0";
289         TPZFileEqnStorage FileStoreW('w',filename);
290 //      FileStoreW.SetBlockSize(10);
291         TPZEqnArray EqnArray;
292
293         ifstream input("MatrizInversa.txt");
294         double aux;
295         int i, j;
296         TPZFMatrix DecMat(Loop_Limit,Loop_Limit);
297         for(i=0;i<Loop_Limit;i++){
298                 for(j=0;j<Loop_Limit;j++){
299                         input >> aux;
300                         DecMat(i,j)=aux;
301                 }
302         }
303
304         //DecMat.Print("MatrizInv");
305
306         for(i=0;i<Loop_Limit;i++){
```

```
307                 EqnArray.BeginEquation(i);
308                 for(j=i;j<Loop_Limit;j++){
309                         aux = DecMat(i,j);
310                         EqnArray.AddTerm(j,DecMat(i,j));
311                 }
312                 EqnArray.EndEquation();
313         }
314
315         FileStoreW.AddEqnArray(&EqnArray);
316
317         FileStoreW.FinishWriting();
318
319         FileStoreW.ReOpen();
320
321         ofstream output("testeFileBin.txt");
322
323         FileStoreW.Print("Teste",output);
324
325         TPZFMatrix f(Loop_Limit,1);
326
327         for(i=0;i<Loop_Limit;i++) {
328                 f(i,0) = (i+1)*2.1/23;
329         }
330
331         f.Print("Teste");
332
333         FileStoreW.Forward(f,ECholesky);
334
335 }
```

3.2.3.7 void TPZFileEqnStorage::OpenGeneric (char *option*, const char * *name*)

Sets file name and if it is for input or output, the second term can be either 'r' for input and 'w' for output.

Parameters:

option 'w' means writing and 'r' reading

name The file name to print to

Definition at line 400 of file TPZFileEqnStorage.c.

```
401 {
402         fCurBlockPosition = -1;
403         fNumBlocks=0;
404         fCurrentBlock=0;
405         fNumHeaders=11;
406 //      SetBlockSize(10);
407         //fBlockPos.Resize(fNumHeaders);
408
```

```
409
410         strcpy(fFileName,name);
411
412         if(option=='r'){
413                 fIOStream = fopen(fFileName,"rb"); //open for reading
418                 fread(&fNumHeaders,sizeof(int),1,fIOStream);
419                 fread(&fNumBlocks,sizeof(int),1,fIOStream);
420                 ReadBlockPositions();
421         }else if(option=='w'){
422                 fIOStream = fopen(fFileName,"wb"); //open for writing
427                 int zero = 0;
428                 fwrite(&fNumHeaders,sizeof(int),1,fIOStream);
429                 //cout << ftell(fIOStream) << endl;
430                 fwrite(&zero,sizeof(int),1,fIOStream);
431                 //cout << ftell(fIOStream) << endl;
432                 //fCurBlockPosition = ftell(fIOStream);
433         }
434 }
```

3.2.3.8 void TPZFileEqnStorage::Print (const char * *name*, ostream & *out*)

It prints TPZEqnStorage data.

Parameters:

name File name to print to

out ofstream object name

Definition at line 184 of file TPZFileEqnStorage.c.

```
184                                                             {
185     int i;
186         TPZEqnArray REqnArray;
187         out <<  "Number of entries on File  "<< fBlockPos.NElements() << endl;
188         for(i=0;i<fBlockPos.NElements();i++) {
189                 if (fBlockPos[i]) {
190                         fseek(fIOStream,fBlockPos[i],SEEK_SET);
191                         REqnArray.Read(fIOStream);
192                         REqnArray.Print(name, out);
193                 }
194         }
195 }
```

3.2.3.9 void TPZFileEqnStorage::ReadBlockPositions ()

Method used for binary input/output

Definition at line 466 of file TPZFileEqnStorage.c.

```
467 {
468         int aux = fNumBlocks * (fNumHeaders-1);
469         fBlockPos.Resize(aux);
470         int i, ibl = 0;
471         for(i=0;i<fNumBlocks;i++) {
472                 fread(&fBlockPos[ibl],sizeof(long int),fNumHeaders-1,fIOStream);
473                 ibl+=fNumHeaders-1;
474                 long int nextpos;
475                 fread(&nextpos,sizeof(long int),fNumHeaders-1,fIOStream);
476                 fseek(fIOStream,nextpos,SEEK_SET);
477         }
478
479
480 }
```

3.2.3.10 void TPZFileEqnStorage::ReOpen ()

Reopens an binary file with its current fFileName

Definition at line 387 of file TPZFileEqnStorage.c.

```
388 {
389         fIOStream = fopen(fFileName,"rb"); //open for reading
394         fread(&fNumHeaders,sizeof(int),1,fIOStream);
395         fread(&fNumBlocks,sizeof(int),1,fIOStream);
396 //      ReadBlockPositions();
397
398 }
```

3.2.3.11 void TPZFileEqnStorage::Reset ()

Resets data

Definition at line 179 of file TPZFileEqnStorage.c.

```
180 {
181
182 }
```

3.2.3.12 void TPZFileEqnStorage::Store (int *ieq*, int *jeq*, const char * *name*)

Stores from ieq to jeq equations on a binary file

Parameters:

ieq Initial equation to be added to EqnArray

jeq Final equation to be added to EqnArray

name Binary file name

Definition at line 61 of file TPZFileEqnStorage.c.

```
61                                                                    {
62          //Initial tests with C input output files !
63          int loop_limit=100;// = jeq-ieq;
64          int i;
65          int items_writen = 0;
66          int block_size;
67          FILE *out_file = fopen(name,"wb");
68          //cout << "Loop Limit ";
69          //cin >> loop_limit ;
70          //cout << "Block Size";
71          //cin >> block_size;
72
73
74
75 //       struct _iobuf *temp_i;
76          //From MSDN
77 /*
78
79          char list[30];
80          int  i, numread, numwritten;
81
82          /* Open file in text mode:
83          if( (stream = fopen( "fread.out", "w+t" )) != NULL )
84          {
85            for ( i = 0; i < 25; i++ )
86                  list[i] = (char)('z' - i);
87             Write 25 characters to stream
88            numwritten = fwrite( list, sizeof( char ), 25, stream );
89            fprintf( "Wrote %d items\n", numwritten );
90            fclose( stream );
91 */
92          double number=2.1;
93          double val = 0;
94          long int fPos[5] = {0};
95          int p;
96          long int firstpos = ftell(out_file);
97          fwrite(fPos,sizeof(long  int),5,out_file);
98          double readvec[4][100];
99          int iblock =0;
100          for(iblock=0; iblock<4; iblock++) {
101                  long int currentpos = ftell(out_file);
102                  fseek(out_file,firstpos+iblock*sizeof(long int),SEEK_SET);
103                  fwrite(&currentpos,sizeof(long int),1,out_file);
104                  fseek(out_file,currentpos,SEEK_SET);
105                  for(i=0;i<loop_limit;i++){
106                          val=number*i*(iblock+1);
107                          fwrite(&val, sizeof(double), 1, out_file);
108                  }
109          }
```

```
110         fclose(out_file);
111         out_file = fopen(name,"rb");
112         fread(fPos,sizeof(long int),5,out_file);
113         for(iblock = 0; iblock<4; iblock++) {
114                 fseek(out_file,fPos[iblock],SEEK_SET);
115                 fread(readvec[iblock],sizeof(double),loop_limit,out_file);
116         }
117 }
```

3.2.3.13 void TPZFileEqnStorage::Store_Old (int *ieq*, int *jeq*, const char * *name*)

Stores "n" equations in binary files. It uses a special indexation for improvements on searching and writing

Definition at line 271 of file TPZFileEqnStorage.c.

```
272 {
273         int loop_limit = jeq-ieq;
274         int i;
275         int items_writen = 0;
276         FILE *out_file = fopen(name,"wb");
277 /*      for(i=0;i<loop_limit;i++){
278                 fwrite((const char *)fEqnStack, 8, i, name);
279         }
280         */
281 }
```

3.2.3.14 void TPZFileEqnStorage::WriteHeaders ()

Writes the header of the binary file

Definition at line 7 of file TPZFileEqnStorage.c.

```
7                                       {
12        fNumBlocks++;
13        int i;
14
15        long int * Position = new long int[fNumHeaders];
16        for(i=0;i<fNumHeaders;i++) Position[i] = 0;
21        if(fCurrentBlock==0){
22                long int basepos = ftell(fIOStream);
23                fBlockPos.Push(basepos);
24        }else{
25                long int tempaddress = ftell(fIOStream);
26                fBlockPos.Push(tempaddress);
27        }
28
29        long int firstpos = ftell(fIOStream);
```

```
30          //if (fCurrentBlock) firstpos =
31
35          fwrite(Position,sizeof(long  int),fNumHeaders,fIOStream);
36          delete [] Position;
40          long int firstaddress = ftell(fIOStream);
41
45          fseek(fIOStream,firstpos,SEEK_SET);
46          fwrite(&firstaddress,sizeof(long int),1,fIOStream);
50          fCurBlockPosition = firstaddress;
51
55          fseek(fIOStream,firstaddress,SEEK_SET);
56
57 }
```

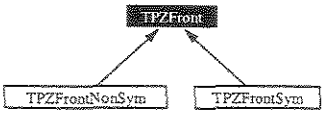
The documentation for this class was generated from the following files:

- **TPZFileEqnStorage.h**
- **TPZFileEqnStorage.c**

3.3 TPZFront Class Reference

`#include <TPZFront.h>`

Inheritance diagram for TPZFront:



Public Methods

- ~**TPZFront** ()
- **TPZFront** ()
- **TPZFront** (int GlobalSize)
- void **SymbolicDecomposeEquations** (int mineq, int maxeq)
- void **SymbolicAddKel** (TPZVec< int > &destinationindex)
- int **NFree** ()
- void **Reset** (int GlobalSize=0)
- void **Print** (const char *name, ostream &out)
- void **PrintGlobal** (const char *name, ostream &out=cout)

Static Public Methods

- void **main** ()

Protected Attributes

- int **fMaxFront**
- TPZManVector< int > **fGlobal**
- TPZVec< int > **fLocal**
- int **fFront**
- TPZStack< int > **fFree**
- TPZVec< REAL > **fData**
- int **fExpandRatio**

3.3.1 Detailed Description

The Front matrix itself.

It is controled by **TPZFrontMatrix** (p. 43).

Definition at line 17 of file TPZFront.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 TPZFront::~TPZFront ()

Simple destructor

Definition at line 159 of file TPZFront.c.

```
159                     {
160 }
```

3.3.2.2 TPZFront::TPZFront ()

Simple constructor

Definition at line 137 of file TPZFront.c.

References fExpandRatio, fFront, and fMaxFront.

```
137                     {
138         fFront=0;
139         fMaxFront=0;
140       fExpandRatio = 200;
141         fFront = 0;
142         fMaxFront=0;
143 }
```

3.3.2.3 TPZFront::TPZFront (int *GlobalSize*)

Constructor with a initial size parameter

Parameters:

GlobalSize Initial size of the Frontal Matrix

Definition at line 147 of file TPZFront.c.

References fExpandRatio, fFront, fLocal, and fMaxFront.

```
148 {
149         fFront=0;
150         fMaxFront=0;
151      fExpandRatio = 200;
152         fFront = 0;
153         fMaxFront=0;
154         fLocal.Resize(GlobalSize);
155         int i;
156         for(i=0;i<GlobalSize;i++) fLocal[i]=-1;
157 }
```

3.3.3 Member Function Documentation

3.3.3.1 void TPZFront::main () [static]

Static main used for testing

Reimplemented in **TPZFrontNonSym** (p. 61).

Definition at line 165 of file TPZFront.c.

References TPZEqnArray::EqnBackward, TPZEqnArray::EqnForward, TPZEqnArray::Print, SymbolicAddKel, and SymbolicDecomposeEquations.

```
166 {
167         int i, j;
171         int matsize=6;
172         TPZFMatrix TestMatrix(matsize,matsize);
173         for(i=0;i<matsize;i++) {
174                 for(j=i;j<matsize;j++) {
175                         int random = rand();
176                         double rnd = (random*matsize)/RAND_MAX;
177                         TestMatrix(i,j)=rnd;
178                         TestMatrix(j,i)=TestMatrix(i,j);
179                         if(i==j) TestMatrix(i,j)=6000.;
180                 }
181         }
182
183         TPZFMatrix Prova;
184         Prova=TestMatrix;
185
186 //      Prova.Decompose_Cholesky();
187         Prova.Print("TPZFMatrix Cholesky");
188
189         TPZFront TestFront(matsize);
190
191
192         TPZVec<int> DestIndex(matsize);
193         for(i=0;i<matsize;i++) DestIndex[i]=i;
194
195         TestFront.SymbolicAddKel(DestIndex);
196         TestFront.SymbolicDecomposeEquations(0,matsize-1);
197
```

```
198         char * OutFile;
199         OutFile = "TPZFrontTest.txt";
200
201         ofstream output(OutFile,ios::app);
202
203 //      TestFront.Compress();
204
205 //      TestFront.AllocData();
206
207 //      TestFront.AddKel(TestMatrix, DestIndex);
208         TPZEqnArray Result;
209
210         /*TestFront.DecomposeEquations(0,0,Result);
211
212         TestFront.Print(OutFile, output);
213
214         ofstream outeqn("TestEQNArray.txt",ios::app);
215         Result.Print("TestEQNArray.txt",outeqn);
216
217         TestFront.Compress();
218
219         TestFront.Print(OutFile, output);
220 */
221 //      TestFront.DecomposeEquations(0,matsize-1,Result);
222         ofstream outeqn("TestEQNArray.txt",ios::app);
223
224         Result.Print("TestEQNArray.txt",outeqn);
225
226
227         TPZFMatrix Load(matsize);
228
229         for(i=0;i<matsize;i++) {
230                 int random = rand();
231                 double rnd = (random*matsize)/RAND_MAX;
232                 Load(i,0)=rnd;
233         }
234
235         TPZFMatrix Load_2(matsize);
236         Load_2=Load;
237
238 //      Prova.Subst_Forward(&Load);
239 //      Prova.Subst_Backward(&Load);
240
241
242         DecomposeType decType = ECholesky;
243         Prova.SolveDirect(Load, decType);
244
245         Load.Print();
246         //TestFront.Print(OutFile, output);
247
248         Result.EqnForward(Load_2, decType);
249         Result.EqnBackward(Load_2, decType);
250
251         Load_2.Print("Eqn");
252
```

```
253
254
255 }
```

3.3.3.2 int TPZFront::NFree ()

Returns the number of free equations

Reimplemented in **TPZFrontNonSym** (p. 63).

Definition at line 269 of file TPZFront.c.

References fLocal.

```
270 {
271         int i;
272         int free_eq=0;
273         for(i=0;i<fLocal.NElements();i++)
274         {
275                 if(fLocal[i]==-1){
276                         free_eq=free_eq+1;
277                 }
278         }
279         return free_eq;
280 }
```

3.3.3.3 void TPZFront::Print (const char * *name*, ostream & *out*)

It prints TPZFront data

Reimplemented in **TPZFrontNonSym** (p. 64).

Definition at line 41 of file TPZFront.c.

```
42 {
43   if(name) out << name << endl;
44   /*int i,j,loop_limit;
45
46
47   out <<  "Frontal Matrix Size          "<< fFront << endl;
48   out <<  "Maximum Frontal Matrix Size  "<< fMaxFront << endl;
49
50   out << endl;
51   out <<  "Local Indexation "<< endl;
52   out <<  "Position  "<<" Local index"<< endl;
53   for(i=0;i<fLocal.NElements();i++){
54     out <<   i <<  "         "  << fLocal[i] << endl;
55   }
56
57   out << endl;
```

```
58   out <<  "Global Indexation "<< endl;
59   out <<  "Position  "<<" Global index"<< endl;
60
61   for(i=0;i<fGlobal.NElements();i++){
62     out <<   i <<  "            "<< fGlobal[i] << endl;
63   }
64
65   out << endl;
66   out <<  "Local Freed Equatoins  "<< fFree.NElements() << endl;
67   out <<  "position "<<   "Local Equation "<<endl;
68   loop_limit=fFree.NElements();
69   for(i=0;i<loop_limit;i++){
70     out <<  i <<  "             " << fFree[i] << endl;
71   }
72   out <<  "Frontal Matrix "<< endl;
73   if(fData.NElements() > 0) {
74           for(i=0;i<fFront;i++){
75                   for(j=0;j<fFront;j++) out << ((i<j) ? Element(i,j) : Element(j,i)) <<  " ";
76                 out << endl;
77           }
78   }*/
79
80         out << "Not implemented in the abstract Class !" << endl;
81         out << "Try one of the subclasses" << endl;
82 }
```

3.3.3.4 void TPZFront::Reset (int *GlobalSize* = 0)

Resets data structure

Reimplemented in **TPZFrontNonSym** (p. 65).

Definition at line 257 of file TPZFront.c.

References fData, fFree, fFront, fGlobal, fLocal, and fMaxFront.

```
258 {
259         fData.Resize(0);
260         fFree.Resize(0);
261         fFront=0;
262         fGlobal.Resize(0);
263         fLocal.Resize(GlobalSize);
264         fLocal.Fill(-1);
265         fMaxFront=0;
266 }
```

3.3.3.5 void TPZFront::SymbolicAddKel (TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix using the indexes to compute the frontwidth

Parameters:

destinationindex Destination index of each element added

Reimplemented in **TPZFrontNonSym** (p. 65).

Definition at line 120 of file TPZFront.c.

References fFront, fGlobal, and fMaxFront.

Referenced by main.

```
121 {
122         int i, loop_limit, aux;
123         loop_limit=destinationindex.NElements();
124         for(i=0;i<loop_limit;i++){
125                 aux=destinationindex[i];
126                 Local(aux);
127                 fFront = fGlobal.NElements();
128         }
129         fMaxFront=(fFront<fMaxFront)?fMaxFront:fFront;
130
131 }
```

3.3.3.6 void TPZFront::SymbolicDecomposeEquations (int *mineq*, int *maxeq*)

Decompose these equations in a symbolic way and store freed indexes in fFree

Parameters:

mineq Initial equation

maxeq Final equation

Reimplemented in **TPZFrontNonSym** (p. 66).

Definition at line 132 of file TPZFront.c.

Referenced by main.

```
133 {
134         int i;
135         for(i=mineq;i<=maxeq;i++) FreeGlobal(i);
136 }
```

3.3.4 Member Data Documentation

3.3.4.1 TPZVec<REAL> TPZFront::fData [protected]

Frontal matrix data

Definition at line 118 of file TPZFront.h.

Referenced by TPZFrontSym::AllocData, TPZFrontNonSym::AllocData, TPZFrontSym::Compress, TPZFrontNonSym::Compress, TPZFrontSym::Element, TPZFrontNonSym::Element, TPZFrontSym::Expand, TPZFrontNonSym::Expand, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, and Reset.

3.3.4.2 int TPZFront::fExpandRatio [protected]

Expantion Ratio of frontal matrix

Definition at line 123 of file TPZFront.h.

Referenced by TPZFront.

3.3.4.3 TPZStack<int> TPZFront::fFree [protected]

Colection of already decomposed equations still on the front

Definition at line 113 of file TPZFront.h.

Referenced by TPZFrontSym::Compress, TPZFrontNonSym::Compress, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, and Reset.

3.3.4.4 int TPZFront::fFront [protected]

Actual front size

Definition at line 108 of file TPZFront.h.

Referenced by TPZFrontSym::AllocData, TPZFrontNonSym::AllocData, TPZFrontSym::Compress, TPZFrontNonSym::Compress, TPZFrontNonSym::Expand, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, Reset, TPZFrontSym::SymbolicAddKel, TPZFrontNonSym::SymbolicAddKel, SymbolicAddKel, and TPZFront.

3.3.4.5 TPZManVector<int> TPZFront::fGlobal [protected]

Global equation associated to each front equation

If we need a position in globalmatrix of a equation "i" in the frontmatrix

then we can use fGlobal[i]. If the global equation "i" is not used

then fGlobal[i]==-1.

Definition at line 95 of file TPZFront.h.

Referenced by TPZFrontSym::AllocData, TPZFrontNonSym::AllocData, TPZFrontSym::Compress, TPZFrontNonSym::Compress, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, Reset, TPZFrontSym::SymbolicAddKel, TPZFrontNonSym::SymbolicAddKel, and SymbolicAddKel.

3.3.4.6 TPZVec<int> TPZFront::fLocal [protected]

Front equation to each global equation

If we need a position in frontmatrix of a global equation "i"

then we can use fLocal[i]. If the global equation is not represented in the front

then fLocal[i]==-1.

Definition at line 103 of file TPZFront.h.

Referenced by TPZFrontSym::Compress, TPZFrontNonSym::Compress, TPZFrontSym::NFree, TPZFrontNonSym::NFree, NFree, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, Reset, and TPZFront.

3.3.4.7 int TPZFront::fMaxFront [protected]

Maximum size of the front

Definition at line 86 of file TPZFront.h.

Referenced by TPZFrontSym::AllocData, TPZFrontNonSym::AllocData, TPZFrontNonSym::Element, TPZFrontSym::Expand, TPZFrontNonSym::Expand, TPZFrontSym::Print, TPZFrontNonSym::Print, TPZFrontSym::Reset, TPZFrontNonSym::Reset, Reset, TPZFrontSym::SymbolicAddKel, TPZFrontNonSym::SymbolicAddKel, SymbolicAddKel, and TPZFront.

The documentation for this class was generated from the following files:

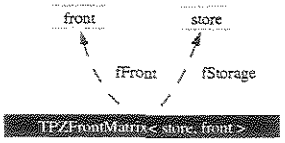
- **TPZFront.h**
- **TPZFront.c**

3.4 TPZFrontMatrix Class Template Reference

`#include <TPZFrontMatrix.h>`

Inheritance diagram for TPZFrontMatrix:

Collaboration diagram for TPZFrontMatrix:



Public Methods

- void **FinishWriting** ()
- void **AllocData** ()
- void **CheckCompress** ()
- void **Print** (const char *name, ostream &out)
- **~TPZFrontMatrix** ()
- **TPZFrontMatrix** ()
- **TPZFrontMatrix** (int globalsize)
- void **EquationsToDecompose** (TPZVec< int > &destinationindex, int &lower_eq, int &upper_eq)
- void **SetNumElConnected** (TPZVec< int > &numelconnected)
- virtual void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &destinationindex)
- void **SymbolicAddKel** (TPZVec< int > &destinationindex)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &sourceindex, TPZVec< int > &destinationindex)
- int **Subst_Forward** (TPZFMatrix *b) const
- int **Subst_Backward** (TPZFMatrix *b) const
- int **Substitution** (TPZFMatrix *) const
- void **SetFileName** (char option, const char *name)

Static Public Methods

- void **main** ()

Protected Attributes

- store **fStorage**
- front **fFront**
- int **fNumEq**
- int **fLastDecomposed**
- TPZVec< int > **fNumElConnected**

3.4.1 Detailed Description

template<class store, class front> class TPZFrontMatrix< store, front >

Class responsible for the frontal method as a whole Manages the remaining classes connecting them

Definition at line 26 of file TPZFrontMatrix.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 template<class store, class front> TPZFrontMatrix< store, front >::~TPZFrontMatrix ()

Simple Destructor

Definition at line 130 of file TPZFrontMatrix.c.

```
130                                                 {
131 }
```

3.4.2.2 template<class store, class front> TPZFrontMatrix< store, front >::TPZFrontMatrix ()

Simple Constructor

Definition at line 135 of file TPZFrontMatrix.c.

References fFront, fLastDecomposed, fNumElConnected, fNumEq, and fStorage.

```
136 {
137         fFront.Reset();
138         fStorage.Reset();
139         fNumElConnected.Resize(0);
140         fLastDecomposed = -1;
141         fNumEq=0;
142 }
```

3.4.2.3 template<class store, class front> TPZFrontMatrix< store, front >::TPZFrontMatrix (int *globalsize*)

Constructor with a globalsize parameter

Parameters:

globalsize Indicates initial global size

Definition at line 145 of file TPZFrontMatrix.c.

References fFront, fLastDecomposed, fNumElConnected, fNumEq, and fStorage.

```
145                                                     : TPZMatrix(globalsize,globalsize)
146 {
147         fFront.Reset(globalsize);
148         fStorage.Reset();
149         fNumElConnected.Resize(0);
150         fLastDecomposed = -1;
151         fNumEq=globalsize;
152 }
```

3.4.3 Member Function Documentation

3.4.3.1 template<class store, class front> void TPZFrontMatrix< store, front >::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *sourceindex*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Parameters:

elmat Member stiffness matrix beeing added

sourceindex Sorce position of values on member stiffness matrix

destinationindex Positioning of such members on global stiffness matrix

Reimplemented in **TPZParFrontMatrix** (p. 94).

Definition at line 85 of file TPZFrontMatrix.c.

References CheckCompress, EquationsToDecompose, fFront, and fStorage.

```
86 {
87         fFront.AddKel(elmat, sourceindex, destinationindex);
88 //      EquationsToDecompose(destinationindex);
89  //         cout << "AddKel::destination index 2" << endl;
90      int i;
91 //          for(i=0;i<destinationindex.NElements();i++) cout << destinationindex[i] << " ";
92  //         cout << endl;
93  //         cout.flush();
94 //          elmat.Print("AddKel: Element Matrix 2");
95         int mineq, maxeq;
96         EquationsToDecompose(destinationindex, mineq, maxeq);
97         TPZEqnArray AuxEqn;
98         if(maxeq >= mineq) {
99                 fFront.DecomposeEquations(mineq,maxeq,AuxEqn);
100                 CheckCompress();
101                 fStorage.AddEqnArray(&AuxEqn);
102                 if(maxeq == Rows()-1){
103                      fFront.Reset(0);
104                         fStorage.FinishWriting();
105                         fStorage.ReOpen();
106                 }
107
108         }
109         fDecomposed = fFront.GetDecomposeType();
110 }
```

3.4.3.2 template<class store, class front> void TPZFrontMatrix< store, front >::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *destinationindex*) [virtual]

Add a contribution of a stiffness matrix putting it on destination indexes position

Parameters:

elmat Member stiffness matrix beeing added

destinationindex Positioning of such members on global stiffness matrix

Reimplemented in **TPZParFrontMatrix** (p. 93).

Definition at line 55 of file TPZFrontMatrix.c.

References CheckCompress, EquationsToDecompose, fFront, and fStorage.

Referenced by main.

```
55
56
57          // message #1.3 to fFront:TPZFront
58          fFront.AddKel(elmat, destinationindex);
59     /*     cout << "destination index" << endl;
60          int i;
```

```
61          for(i=0;i<destinationindex.NElements();i++) cout << destinationindex[i] << " ";
62          cout << endl;
63          cout.flush();
64          elmat.Print("Element Matrix");
65          */
66                int mineq, maxeq;
67
68                EquationsToDecompose(destinationindex, mineq, maxeq);
69                TPZEqnArray AuxEqn;
70                if(maxeq >= mineq) {
71
72                        fFront.DecomposeEquations(mineq,maxeq,AuxEqn);
73                        CheckCompress();
74                        fStorage.AddEqnArray(&AuxEqn);
75                        if(maxeq == Rows()-1){
76                                fStorage.FinishWriting();
77                                fStorage.ReOpen();
78                        }
79                }
80                fDecomposed = fFront.GetDecomposeType();
81 }
```

3.4.3.3 template<class store, class front> void TPZFrontMatrix< store, front >::AllocData ()

Allocates data for the FrontMatrix

Definition at line 249 of file TPZFrontMatrix.c.

References fFront, and fLastDecomposed.

Referenced by main.

```
250 {
251         fFront.AllocData();
252         fLastDecomposed=-1;
253 }
```

3.4.3.4 template<class store, class front> void TPZFrontMatrix< store, front >::CheckCompress ()

Chacks if FrontMatrix needs a compression, if so calls Compress method

Definition at line 243 of file TPZFrontMatrix.c.

References fFront.

Referenced by TPZParFrontMatrix::AddKel, AddKel, and SymbolicAddKel.

```
244 {
```

```
245         if(fFront.NFree()>0) fFront.Compress();
246 }
```

3.4.3.5 template<class store, class front> void TPZFrontMatrix< store, front >::EquationsToDecompose (TPZVec< int > & *destinationindex*, int & *lower_eq*, int & *upper_eq*)

Sends a message to decompose equations from lower_eq to upper_eq, according to destination index

Parameters:

destinationindex Contains destination indexes of equations

lower_eq Starting index

upper_eq Finishing index

Definition at line 19 of file TPZFrontMatrix.c.

References fLastDecomposed, fNumElConnected, and fNumEq.

Referenced by TPZParFrontMatrix::AddKel, AddKel, and SymbolicAddKel.

```
20 {
21      int i;
22         int loop_limit, global;
23         loop_limit = destinationindex.NElements();
24         for(i=0;i<loop_limit;i++){
25                 global = destinationindex[i];
26                 fNumElConnected[global]--;
27         }
28         upper_eq=fLastDecomposed;
29         lower_eq=fLastDecomposed+1;
30         while(upper_eq < fNumEq-1 && fNumElConnected[upper_eq+1]==0) upper_eq++;
31         fLastDecomposed=upper_eq;
32 }
```

3.4.3.6 template<class store, class front> void TPZFrontMatrix< store, front >::FinishWriting ()

Finishes writing of a binary file and closes it

Reimplemented in **TPZParFrontMatrix** (p. 95).

Definition at line 293 of file TPZFrontMatrix.c.

References fStorage.

```
293                                                 {
294         fStorage.FinishWriting();
295 }
```

3.4.3.7 template<class store, class front> void TPZFrontMatrix< store, front >::main () [static]

Static main for testing

Definition at line 170 of file TPZFrontMatrix.c.

References AddKel, AllocData, SetNumElConnected, and SymbolicAddKel.

```
171 {
172         TPZFMatrix KEl1(2,2);
173         KEl1(0,0)=4.;
174         KEl1(0,1)=6.;
175         KEl1(1,0)=6.;
176         KEl1(1,1)=12.;
177         TPZVec<int> DestInd1(2);
178         DestInd1[0]=0;
179         DestInd1[1]=1;
180
181
182         TPZFMatrix KEl2(4,4);
183         KEl2(0,0)=4.;
184         KEl2(0,1)=-6.;
185         KEl2(0,2)=2.;
186         KEl2(0,3)=6.;
187
188         KEl2(1,1)=12;
189         KEl2(1,2)=-6.;
190         KEl2(1,3)=-12.;
191
192         KEl2(2,2)=4.;
193         KEl2(2,3)=6.;
194
195         KEl2(3,3)=12.;
196
197         int i, j;
198         for(i=0;i<4;i++) {
199                 for(j=i;j<4;j++) KEl2(j,i)=KEl2(i,j);
200         }
201
202         TPZVec<int> DestInd2(4);
203         DestInd2[0]=0;
204         DestInd2[1]=1;
205         DestInd2[2]=2;
206         DestInd2[3]=3;
207
208
209
210         TPZFMatrix KEl3(2,2);
211         KEl3(0,0)=3.;
212         KEl3(0,1)=3.;
213         KEl1(1,0)=3.;
214         KEl3(1,1)=6.;
215
216         TPZVec<int> DestInd3(2);
```

```
217         DestInd3[0]=2;
218         DestInd3[1]=3;
219
220
221
222         TPZFrontMatrix TestFront(4);
223
224         TPZVec<int> NumConnected(4);
225         for(i=0;i<4;i++) NumConnected[i]=2;
226
227
228         TestFront.SetNumElConnected(NumConnected);
229
230         TestFront.SymbolicAddKel(DestInd1);
231         TestFront.SymbolicAddKel(DestInd2);
232         TestFront.SymbolicAddKel(DestInd3);
233
234         TestFront.AllocData();
235         TestFront.SetNumElConnected(NumConnected);
236
237         TestFront.AddKel(KEl1,DestInd1);
238         TestFront.AddKel(KEl2,DestInd2);
239         TestFront.AddKel(KEl3,DestInd3);
240 }
```

3.4.3.8 template<class store, class front> void TPZFrontMatrix< store, front >::Print (const char * *name*, ostream & *out*)

Prints a FrontMatrix object

Definition at line 155 of file TPZFrontMatrix.c.

References fFront, fNumElConnected, fNumEq, and fStorage.

```
156 {
157         int i;
158         out << "Frontal Matrix associated"<< endl;
159         fFront.Print(name, out);
160         out << "Stored Equations" << endl;
161         fStorage.Print(name, out);
162         out << "Number of Equations " << fNumEq << endl;
163         out << "Number of Elements connected to DF" << endl;
164         for(i=0;i<fNumElConnected.NElements();i++){
165                 out << i << " " << fNumElConnected[i] << endl;
166         }
167 }
```

3.4.3.9 template<class store, class front> void TPZFrontMatrix< store, front >::SetFileName (char *option*, const char * *name*)

Sets a file name for generic input or output

Parameters:

option It can be either 'w' or 'r' for writeing or reading respectively

name Name of the file.

Definition at line 287 of file TPZFrontMatrix.c.

References fStorage.

```
287                                                                    {
288         fStorage.OpenGeneric(option, name);
289 }
```

3.4.3.10 template<class store, class front> void TPZFrontMatrix< store, front >::SetNumElConnected (TPZVec< int > & *numelconnected*)

Initializes the number of elements connected to each equation

Parameters:

numelconnected Indicates number of elements connected to that equation

Definition at line 37 of file TPZFrontMatrix.c.

References fFront, fNumElConnected, and fStorage.

Referenced by TPZFrontStructMatrix::Create, TPZParFrontStructMatrix::CreateAssemble, TPZFrontStructMatrix::CreateAssemble, and main.

```
37                                                                     {
38      fNumElConnected.Resize(numelconnected.NElements());
39         fNumElConnected=numelconnected;
40         cout << "Storage Schema -> " << fStorage.GetStorage() << endl;
41         cout << "Front Matrix Type -> " << fFront.GetMatrixType() << endl;
42         #ifdef BLAS
43         cout << "Using BLAS" << endl;
44      #endif
45      #ifdef ATLAS
46           cout << "Using ATLAS" << endl;
47      #endif
48         #ifdef NOBLAS
49         cout << "Not Using BLAS" << endl;
50         #endif
51 }
```

3.4.3.11 template<class store, class front> int TPZFrontMatrix< store, front >::Subst_Backward (TPZFMatrix * b) const

Backward substitution and result is on b

Definition at line 271 of file TPZFrontMatrix.c.

References fFront, and fStorage.

```
271                                                                    {
272         DecomposeType dec = fFront.GetDecomposeType();
273         if(dec != ECholesky) cout << "TPZFrontMatrix::Subst_Forward non matching decompositio
274         fStorage.Backward(*b, ECholesky);
275         return 1;
276 }
```

3.4.3.12 template<class store, class front> int TPZFrontMatrix< store, front >::Subst_Forward (TPZFMatrix * b) const

Forward substitution and result is on b

Parameters:

b Result of the substitution

Definition at line 263 of file TPZFrontMatrix.c.

References fFront, and fStorage.

```
263                                                                    {
264         DecomposeType dec = fFront.GetDecomposeType();
265         if(dec != ECholesky) cout << "TPZFrontMatrix::Subst_Forward non matching decompositio
266         fStorage.Forward(*b, ECholesky);
267         return 1;
268 }
```

3.4.3.13 template<class store, class front> int TPZFrontMatrix< store, front >::Substitution (TPZFMatrix *) const

Executes a substitution on a TPZFmatrix object applies both forward and backward substitution automaticaly

Definition at line 256 of file TPZFrontMatrix.c.

References fStorage.

```
256                                                                    {
257         fStorage.Forward(*b, ELU);
258         fStorage.Backward(*b, ELU);
259         return 1;
260 }
```

3.4.3.14 template<class store, class front> void TPZFrontMatrix< store, front >::SymbolicAddKel (TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix using the indexes to compute the frontwidth It does it symbolicaly

Parameters:
destinationindex Array containing destination indexes

Definition at line 115 of file TPZFrontMatrix.c.

References CheckCompress, EquationsToDecompose, and fFront.

Referenced by main.

```
116 {
117         fFront.SymbolicAddKel(destinationindex);
118         int mineq, maxeq;
119         EquationsToDecompose(destinationindex, mineq, maxeq);
120
121         if(maxeq >= mineq) {
122                 fFront.SymbolicDecomposeEquations(mineq,maxeq);
123                 CheckCompress();
124         }
125
126 }
```

3.4.4 Member Data Documentation

3.4.4.1 template<class store, class front> front TPZFrontMatrix::fFront [protected]

Indicates Front matrix type. Assumes values **TPZFrontSym** (p. 79) for symmetric front and **TPZFrontNonSym** (p. 55) for non symmetric front matrix

Definition at line 37 of file TPZFrontMatrix.h.

Referenced by TPZParFrontMatrix::AddKel, AddKel, AllocData, CheckCompress, Print, SetNumElConnected, Subst_Backward, Subst_Forward, SymbolicAddKel, and TPZFrontMatrix.

3.4.4.2 template<class store, class front> int TPZFrontMatrix::fLastDecomposed [protected]

Indicates last decomposed equation

Definition at line 128 of file TPZFrontMatrix.h.

Referenced by AllocData, EquationsToDecompose, and TPZFrontMatrix.

3.4.4.3 template<class store, class front> TPZVec<int> TPZFrontMatrix::fNumElConnected [protected]

Contains the number of elements which still need to contribute to a given equation

Definition at line 137 of file TPZFrontMatrix.h.

Referenced by EquationsToDecompose, Print, SetNumElConnected, and TPZFrontMatrix.

3.4.4.4 template<class store, class front> int TPZFrontMatrix::fNumEq [protected]

Indicates number of equations

Definition at line 126 of file TPZFrontMatrix.h.

Referenced by EquationsToDecompose, Print, and TPZFrontMatrix.

3.4.4.5 template<class store, class front> store TPZFrontMatrix::fStorage [protected]

Indicates storage schema. Assumes values **TPZFileEqnStorage** (p. 22) for binary file and **TPZStackEqnStorage** (p. 120) for a TPZStack storage

Definition at line 32 of file TPZFrontMatrix.h.

Referenced by AddKel, FinishWriting, Print, SetFileName, SetNumElConnected, Subst_Backward, Subst_Forward, Substitution, TPZFrontMatrix, and TPZParFrontMatrix::WriteFile.

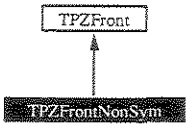
The documentation for this class was generated from the following files:

- **TPZFrontMatrix.h**
- **TPZFrontMatrix.c**

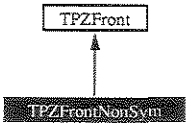
3.5 TPZFrontNonSym Class Reference

`#include <TPZFrontNonSym.h>`

Inheritance diagram for TPZFrontNonSym:



Collaboration diagram for TPZFrontNonSym:



Public Methods

- char * **GetMatrixType** ()
- **~TPZFrontNonSym** ()
- **TPZFrontNonSym** ()
- **TPZFrontNonSym** (int GlobalSize)
- void **DecomposeEquations** (int mineq, int maxeq, **TPZEqnArray** &result)
- void **SymbolicDecomposeEquations** (int mineq, int maxeq)
- void **SymbolicAddKel** (TPZVec< int > &destinationindex)
- void **Compress** ()
- void **Expand** (int largefrontsize)
- REAL & **Element** (int i, int j)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &destinationindex)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &sourceindex, TPZVec< int > &destinationindex)
- int **NFree** ()
- void **Reset** (int GlobalSize=0)
- void **AllocData** ()
- void **Print** (const char *name, ostream &out=cout)
- void **PrintGlobal** (const char *name, ostream &out=cout)
- DecomposeType **GetDecomposeType** () const

Static Public Methods

- void **main** ()

3.5.1 Detailed Description

The Front matrix itself.

It is controled by **TPZFrontMatrix** (p. 43).

TPZFrontNonSym is a non symmetrical matrix. It uses LU decomposition scheme.

Definition at line 35 of file TPZFrontNonSym.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 TPZFrontNonSym::~TPZFrontNonSym ()

Simple destructor

Definition at line 335 of file TPZFrontNonSym.c.

```
335 {}
```

3.5.2.2 TPZFrontNonSym::TPZFrontNonSym ()

Simple constructor

Definition at line 331 of file TPZFrontNonSym.c.

```
331                                 : TPZFront() {
332     fDecomposeType=ELU;
333 }
```

3.5.2.3 TPZFrontNonSym::TPZFrontNonSym (int *GlobalSize*)

Constructor with a initial size parameter

Definition at line 326 of file TPZFrontNonSym.c.

```
326                                                 : TPZFront(GlobalSize)
327 {
328         fDecomposeType=ELU;
329 }
```

3.5.3 Member Function Documentation

3.5.3.1 void TPZFrontNonSym::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *sourceindex*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Parameters:

elmat Already formed element matrix

sourceindex Source index

destinationindex Destine index on the global matrix

Definition at line 205 of file TPZFrontNonSym.c.

References Element.

```
206 {
207         int i, j, ilocal, jlocal, nel;
208         nel=sourceindex.NElements();
209         for (i = 0; i < nel; i++) {
210             // message #1.1.1 to this:TPZFront
211             ilocal = this->Local(destinationindex[i]);
212             for (j = 0; j < nel; j++) {
213                 // message #1.1.2.1 to this:TPZFront
214                 jlocal = this->Local(destinationindex[j]);
215
216                 // message #1.1.2.2 to this:TPZFront
217                 this->Element(ilocal, jlocal)+=elmat(sourceindex[i],sourceindex[j]);
218             }
219         }
220     }
```

3.5.3.2 void TPZFrontNonSym::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Parameters:

elmat Already formed element matrix

destinationindex Destine index on the global matrix

Definition at line 221 of file TPZFrontNonSym.c.

References Element.

Referenced by main.

```
222 {
223     int i, j, ilocal, jlocal, nel;
224     nel = destinationindex.NElements();
225     for(i=0;i<nel;i++){
226         ilocal = this->Local(destinationindex[i]);
227                 for(j=0;j<nel;j++) {
228             jlocal=this->Local(destinationindex[j]);
229             this->Element(ilocal, jlocal)+=elmat(i,j);
230         }
231     }
232 /*
233         output << "Dest Index " ;
234         for(i=0;i<nel;i++) output << destinationindex[i] << " ";
235         output << endl;
236         elmat.Print("Element matrix ",output);
237         PrintGlobal("After Assemb..." , output);
238 */
239 }
```

3.5.3.3 void TPZFrontNonSym::AllocData ()

Allocates data for Front

Definition at line 67 of file TPZFrontNonSym.c.

References TPZFront::fData, TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fMaxFront.

Referenced by main.

```
68 {
69         fData.Resize(fMaxFront*fMaxFront);
70         fGlobal.Fill(-1);
71         fData.Fill(0.);
72         fFront=0;
73         //fLocal.Fill(-1);
74 }
```

3.5.3.4 void TPZFrontNonSym::Compress ()

Compress data structure

Definition at line 255 of file TPZFrontNonSym.c.

References Element, TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fLocal.

Referenced by main.

```
255                             {
```

```
256 //      PrintGlobal("Before COmpress");
257 //      Print("Before Compress", cout);
258         TPZStack <int> from;
259         int nfound;
260         int i, j;
261         for(i = 0; i < fFront; i++){
262                 if(fGlobal[i] != -1) from.Push(i);
263         }
264
269         nfound = from.NElements();
270         for(i=0;i<nfound;i++) {
271                 fGlobal[i]=fGlobal[from[i]];
272                 //fGlobal[from[i]] = -1;
273                 fLocal[fGlobal[i]] = i;
274         }
275         for(;i<fGlobal.NElements();i++) fGlobal[i] = -1;
276
277         if(nfound+fFree.NElements()!=fFront) cout << "TPZFront.Compress inconsistent data structure\n";
278         int frontold = fFront;
279         fFront = nfound;
280         fFree.Resize(0);
281         fGlobal.Resize(fFront);
282         if(fData.NElements()==0) return;
283
284         for(j = 0; j < nfound; j++){
285                 for(i = 0; i < nfound; i++){
286                         Element(i,j) = Element(from[i], from[j]);
287                         if(from[i]!=i || from[j]!=j) Element(from[i],from[j])=0.;
288                 }
289 //              fGlobal[i] = fGlobal[from[i]];
290 //              fLocal[fGlobal[i]] = i;
291         }
292
293 //      Print("After Compress", cout);
294 //      PrintGlobal("After Compress",output);
295 }
```

3.5.3.5 void TPZFrontNonSym::DecomposeEquations (int *mineq*, int *maxeq*, TPZEqnArray & *result*)

Decompose these equations and put the result in eqnarray.

Default decompose method is LU

Parameters:

mineq Starting index of equations to be decomposed

maxeq Finishing index of equations to be decomposed

eqnarray Result of decomposition

Definition at line 313 of file TPZFrontNonSym.c.

References TPZEqnArray::Reset, and TPZEqnArray::SetNonSymmetric.

Referenced by main.

```
313                                                                          {
314         // message #1.1 to eqnarray:TPZEqnArray
315                 int ieq;
316
317         eqnarray.Reset();
318         eqnarray.SetNonSymmetric();
319
320         for (ieq = mineq; ieq <= maxeq; ieq++) {
321             // message #1.2.1 to this:TPZFront
322             this->DecomposeOneEquation(ieq, eqnarray);
323 //                      this->Print("Teste.txt",output);
324         }
325     }
```

3.5.3.6 REAL& TPZFrontNonSym::Element (int *i*, int *j*) [inline]

Returns the ith,jth element of the matrix. @associates <{mat(sourceindex[i],sourceindex[j])}> @semantics +=

Definition at line 88 of file TPZFrontNonSym.h.

References TPZFront::fData, and TPZFront::fMaxFront.

Referenced by AddKel, Compress, and Print.

```
88                                  {
89              return fData[fMaxFront*j + i];
90      }
```

3.5.3.7 void TPZFrontNonSym::Expand (int *largefrontsize*)

Expand the front matrix

Parameters:

larger New size of front

Definition at line 241 of file TPZFrontNonSym.c.

References TPZFront::fData, TPZFront::fFront, and TPZFront::fMaxFront.

```
241                                         {
242 //      PrintGlobal("Antes do Expande");
243         fData.Resize(larger*larger,0.);
244     int i,j;
```

```
245         for(j=fFront-1;j>=0;j--){
246                 for(i=fFront-1;i>=0;i--){
247                         fData[j*larger + i]=fData[j*fMaxFront + i];
248                         if(j) fData[j*fMaxFront+i] = 0.;
249                 }
250         }
251         fMaxFront = larger;
252 //      PrintGlobal("Depois do Expande");
253
254 }
```

3.5.3.8 DecomposeType TPZFrontNonSym::GetDecomposeType () const

Returns decomposition type.

Default LU

Definition at line 7 of file TPZFrontNonSym.c.

```
7                                                   {
8         return fDecomposeType;
9 }
```

3.5.3.9 char * TPZFrontNonSym::GetMatrixType ()

Type of matrix

Definition at line 431 of file TPZFrontNonSym.c.

```
431                                     {
432       return "Non symmetric matrix";
433 }
```

3.5.3.10 void TPZFrontNonSym::main () [static]

Static main used for testing

Reimplemented from **TPZFront** (p. 36).

Definition at line 338 of file TPZFrontNonSym.c.

References AddKel, AllocData, Compress, DecomposeEquations, TPZEqnArray::EqnBackward, TPZEqnArray::EqnForward, TPZEqnArray::Print, SymbolicAddKel, and SymbolicDecomposeEquations.

```
339 {
340         int i, j;
344         int matsize=6;
345         TPZFMatrix TestMatrix(matsize,matsize);
346         for(i=0;i<matsize;i++) {
347                 for(j=i;j<matsize;j++) {
348                         int random = rand();
349                         double rnd = (random*matsize)/0x7fff;
350                         TestMatrix(i,j)=rnd;
351                         TestMatrix(j,i)=TestMatrix(i,j);
352                         if(i==j) TestMatrix(i,j)=6000.;
353                 }
354         }
355
356         TPZFMatrix Prova;
357         Prova=TestMatrix;
358
359 //      Prova.Decompose_Cholesky();
360         Prova.Print("TPZFMatrix Cholesky");
361
362         TPZFrontNonSym TestFront(matsize);
363
364
365         TPZVec<int> DestIndex(matsize);
366         for(i=0;i<matsize;i++) DestIndex[i]=i;
367
368         TestFront.SymbolicAddKel(DestIndex);
369         TestFront.SymbolicDecomposeEquations(0,matsize-1);
370
371         char * OutFile;
372         OutFile = "TPZFrontNonSymTest.txt";
373
374         ofstream output(OutFile,ios::app);
375
376         TestFront.Compress();
377
378         TestFront.AllocData();
379
380         TestFront.AddKel(TestMatrix, DestIndex);
381         TPZEqnArray Result;
382
383         /*TestFront.DecomposeEquations(0,0,Result);
384
385         TestFront.Print(OutFile, output);
386
387         ofstream outeqn("TestEQNArray.txt",ios::app);
388         Result.Print("TestEQNArray.txt",outeqn);
389
390         TestFront.Compress();
391
392         TestFront.Print(OutFile, output);
393 */
394         TestFront.DecomposeEquations(0,matsize-1,Result);
395         ofstream outeqn("TestEQNArray.txt",ios::app);
396
```

```
397         Result.Print("TestEQNArray.txt",outeqn);
398
399
400         TPZFMatrix Load(matsize);
401
402         for(i=0;i<matsize;i++) {
403                 int random = rand();
404                 double rnd = (random*matsize)/0x7fff;
405                 Load(i,0)=rnd;
406         }
407
408         TPZFMatrix Load_2(matsize);
409         Load_2=Load;
410
411 //      Prova.Subst_Forward(&Load);
412 //      Prova.Subst_Backward(&Load);
413
414
415         DecomposeType decType = ECholesky;
416         Prova.SolveDirect(Load, decType);
417
418         Load.Print();
419         //TestFront.Print(OutFile, output);
420
421         Result.EqnForward(Load_2, decType);
422         Result.EqnBackward(Load_2, decType);
423
424         Load_2.Print("Eqn");
425
426
427
428 }
```

3.5.3.11 int TPZFrontNonSym::NFree ()

Returns the number of free equations

Reimplemented from **TPZFront** (p. 38).

Definition at line 85 of file TPZFrontNonSym.c.

References TPZFront::fLocal.

```
86 {
87         int i;
88         int free_eq=0;
89         for(i=0;i<fLocal.NElements();i++)
90         {
91                 if(fLocal[i]==-1){
92                         free_eq=free_eq+1;
93                 }
94         }
95         return free_eq;
```

```
96 }
```

3.5.3.12 void TPZFrontNonSym::Print (const char * *name*, ostream & *out* = cout)

It prints **TPZFront** (p. 34) data

Reimplemented from **TPZFront** (p. 38).

Definition at line 28 of file TPZFrontNonSym.c.

References Element, TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, TPZFront::fLocal, and TPZFront::fMaxFront.

```
29 {
30   if(name) out << name << endl;
31   int i,j,loop_limit;
32
33
34   out <<  "Frontal Matrix Size          "<< fFront << endl;
35   out <<  "Maximum Frontal Matrix Size  "<< fMaxFront << endl;
36
37   out << endl;
38   out <<  "Local Indexation "<< endl;
39   out <<  "Position  "<<" Local index"<< endl;
40   for(i=0;i<fLocal.NElements();i++){
41     out <<   i <<  "          "  << fLocal[i] << endl;
42   }
43
44   out << endl;
45   out <<  "Global Indexation "<< endl;
46   out <<  "Position  "<<" Global index"<< endl;
47
48   for(i=0;i<fGlobal.NElements();i++){
49     out <<   i <<  "            "<< fGlobal[i] << endl;
50   }
51
52   out << endl;
53   out <<  "Local Freed Equations  "<< fFree.NElements() << endl;
54   out <<  "position "<<   "Local Equation "<<endl;
55   loop_limit=fFree.NElements();
56   for(i=0;i<loop_limit;i++){
57     out <<  i <<  "             " << fFree[i] << endl;
58   }
59   out <<  "Frontal Matrix "<< endl;
60   if(fData.NElements() > 0) {
61           for(i=0;i<fFront;i++){
62                   for(j=0;j<fFront;j++) out << Element(i,j) <<  " ";
63                out << endl;
64           }
65   }
66 }
```

3.5.3.13 void TPZFrontNonSym::Reset (int *GlobalSize* = 0)

Resets data structure

Parameters:
GlobalSize Initial global size to be used in reseting.

Reimplemented from **TPZFront** (p. 39).

Definition at line 75 of file TPZFrontNonSym.c.

References TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, TPZFront::fLocal, and TPZFront::fMaxFront.

```
76 {
77         fData.Resize(0);
78         fFree.Resize(0);
79         fFront=0;
80         fGlobal.Resize(0);
81         fLocal.Resize(GlobalSize);
82         fLocal.Fill(-1);
83         fMaxFront=0;
84 }
```

3.5.3.14 void TPZFrontNonSym::SymbolicAddKel (TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix using the indexes to compute the frontwidth

Reimplemented from **TPZFront** (p. 39).

Definition at line 296 of file TPZFrontNonSym.c.

References TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fMaxFront.

Referenced by main.

```
297 {
298         int i, loop_limit, aux;
299         loop_limit=destinationindex.NElements();
300         for(i=0;i<loop_limit;i++){
301                 aux=destinationindex[i];
302                 Local(aux);
303                 fFront = fGlobal.NElements();
304         }
305         fMaxFront=(fFront<fMaxFront)?fMaxFront:fFront;
306
307 }
```

3.5.3.15 void TPZFrontNonSym::SymbolicDecomposeEquations (int *mineq*, int *maxeq*)

Decompose these equations in a symbolic way and store freed indexes in fFree

Reimplemented from **TPZFront** (p. 40).

Definition at line 308 of file TPZFrontNonSym.c.

Referenced by main.

```
309 {
310         int i;
311         for(i=mineq;i<=maxeq;i++) FreeGlobal(i);
312 }
```

The documentation for this class was generated from the following files:

- **TPZFrontNonSym.h**
- **TPZFrontNonSym.c**

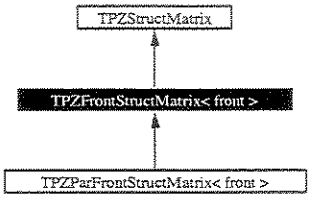
3.6 TPZFrontStructMatrix Class Template Reference

Type parameter for TPZFrontStructMatrix frontal matrix.

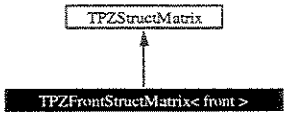
It can assume values **TPZFrontSym** (p. 79) and **TPZFrontNonSym** (p. 55) for symmetric and non symmetric matrices.

`#include <TPZFrontStructMatrix.h>`

Inheritance diagram for TPZFrontStructMatrix:



Collaboration diagram for TPZFrontStructMatrix:



Public Methods

- **TPZFrontStructMatrix** (TPZCompMesh *)
- **~TPZFrontStructMatrix** ()
- TPZMatrix * **Create** ()
- **TPZStructMatrix** * **Clone** ()
- void **AssembleNew** (TPZMatrix &stiffness, TPZFMatrix &rhs)
- void **Assemble** (TPZMatrix &stiffness, TPZFMatrix &rhs)
- void **AssembleElement** (TPZCompEl *el, TPZElementMatrix &ek, TPZElementMatrix &ef, TPZMatrix &stiffness, TPZFMatrix &rhs)
- TPZMatrix * **CreateAssemble** (TPZFMatrix &rhs)

Static Public Methods

- int **main** ()

Protected Methods

- void **GetNumElConnected** (TPZVec< int > &numelconnected)
- void **OrderElement** ()

Protected Attributes

- TPZVec< int > **fElementOrder**

3.6.1 Detailed Description

template<class front> class TPZFrontStructMatrix< front >

Type parameter for TPZFrontStructMatrix frontal matrix.

It can assume values **TPZFrontSym** (p. 79) and **TPZFrontNonSym** (p. 55) for symmetric and non symmetric matrices.

Class responsible for a interface among Finite Element Package and Matrices package

Prevents users from all the necessary information to work with all matrices classes

It facilitates considerably the use of TPZAnalysis

Definition at line 30 of file TPZFrontStructMatrix.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 template<class front> TPZFrontStructMatrix< front >::TPZFrontStructMatrix (TPZCompMesh *)

Class constructor @url `http://www.fec.unicamp.br/~longhin` @url `http://www.fec.unicamp.br/~phil`

Definition at line 52 of file TPZFrontStructMatrix.c.

```
52                                                          : TPZStructMatrix(mesh) {
53
54
55   //TPZFrontMatrix<TPZFileEqnStorage, TPZFrontNonSym> *mat = new TPZFrontMatrix<TPZFileEqnStora
56         //TPZFrontMatrix<TPZStackEqnStorage, TPZFrontNonSym> *mat = new TPZFrontMatrix<TPZStack
57         //TPZFrontMatrix<TPZStackEqnStorage> *mat = new TPZFrontMatrix<TPZStackEqnStorage>(cmes
58
59 /*  TPZVec<int> numelconnected(fMesh->NEquations(),0);
60   TPZFrontMatrix<TPZStackEqnStorage, TPZFrontNonSym> *mat = new TPZFrontMatrix<TPZStackEqnStora
61   GetNumElConnected(numelconnected);
```

```
62   mat->SetNumElConnected(numelconnected);*/
63 }
```

3.6.2.2 template<class front> TPZFrontStructMatrix< front >::~TPZFrontStructMatrix ()

Class destructor @url http://www.fec.unicamp.br/~longhin @url http://www.fec.unicamp.br/~phil

Definition at line 67 of file TPZFrontStructMatrix.c.

```
67 {}
```

3.6.3 Member Function Documentation

3.6.3.1 template<class front> void TPZFrontStructMatrix< front >::Assemble (TPZMatrix & *stiffness*, TPZFMatrix & *rhs*) [virtual]

Assemble a stiffness matrix.

Parameters:

stiffness Stiffness matrix to assembled

rhs Vector contaning loads

Reimplemented from **TPZStructMatrix** (p. 125).

Definition at line 300 of file TPZFrontStructMatrix.c.

References AssembleElement, fElementOrder, and TPZStructMatrix::fMesh.

Referenced by CreateAssemble.

```
300                                                                    {
301
302   int iel;
303   int numel = 0, nelem = fMesh->NElements();
304   TPZElementMatrix ek,ef;
305   REAL stor1[1000],stor2[1000],stor3[100],stor4[100];
306   ek.fMat = new TPZFMatrix(0,0,stor1,1000);
307   ek.fConstrMat = new TPZFMatrix(0,0,stor2,1000);
308   ef.fMat = new TPZFMatrix(0,0,stor3,100);
309   ef.fConstrMat = new TPZFMatrix(0,0,stor4,100);
310
311   TPZAdmChunkVector<TPZCompEl *> &elementvec = fMesh->ElementVec();
312
313
315   TPZVec<int> elorder(fMesh->NEquations(),0);
```

```
316
317 //  OrderElement();
318
319
320   for(iel=0; iel < nelem; iel++) {
321
322      if(fElementOrder[iel] < 0) continue;
323      TPZCompEl *el = elementvec[fElementOrder[iel]];
324      if(!el) continue;
325      //         int dim = el->NumNodes();
326
327      //Builds elements stiffness matrix
328      el->CalcStiff(ek,ef);
329      AssembleElement(el, ek, ef, stiffness, rhs);
330
331      if(!(numel%20)) cout << endl << numel;
332      cout << '*';
333      cout.flush();
334      numel++;
335
336   }//fim for iel
337
338 }
```

3.6.3.2 template<class front> void TPZFrontStructMatrix< front >::AssembleElement (TPZCompEl * *el*, TPZElementMatrix & *ek*, TPZElementMatrix & *ef*, TPZMatrix & *stiffness*, TPZFMatrix & *rhs*)

Computes element matrices.

Each computed element matrices would then be added to Stiffness matrix

Parameters:

el Actual element being computed

ek Formed element matrix

ef Formed element load matrix

stiffness Global Stiffness matrix

rhs Global load matrix

Definition at line 344 of file TPZFrontStructMatrix.c.

References TPZStructMatrix::fMesh.

Referenced by Assemble, and TPZParFrontStructMatrix::GlobalAssemble.

```
344
345
346      int destinationstore[100];
```

```
347     TPZManVector<int> destinationindex(0,destinationstore,100);
348     int sourcestore[100];
349     TPZManVector<int> sourceindex(0,sourcestore,100);
350
351     if(!el->HasDependency()) {
352         //ek.fMat->Print("stiff has no constraint",test);
353         //ef.fMat->Print("rhs has no constraint",test);
354         //test.flush();
355         destinationindex.Resize(ek.fMat->Rows());
356         int destindex = 0;
357         int numnod = ek.NConnects();
358         for(int in=0; in<numnod; in++) {
359             int npindex = ek.ConnectIndex(in);
360             TPZConnect &np = fMesh->ConnectVec()[npindex];
361             int blocknumber = np.SequenceNumber();
362             int firsteq = fMesh->Block().Position(blocknumber);
363             int ndf = fMesh->Block().Size(blocknumber);
364             for(int idf=0; idf<ndf; idf++) {
365                 destinationindex[destindex++] = firsteq+idf;
366             }
367         }
368         //ek.Print(*fMesh,cout);
369         stiffness.AddKel(*ek.fMat,destinationindex);
370         rhs.AddFel(*ef.fMat,destinationindex);                //  ??????????? Erro
371     }
372     else
373     {
374         // the element has dependent nodes
375         el->ApplyConstraints(ek,ef);
376         //ek.fMat->Print("stif no constraint",test);
377         //ek.fConstrMat->Print("stif constrained",test);
378         //ef.fMat->Print("rhs no constraint",test);
379         //ef.fConstrMat->Print("rhs constrained",test);
380         //test.flush();
381         //test << "sum of columns\n";
382         int destindex = 0;
383         int fullmatindex = 0;
384         destinationindex.Resize(ek.fConstrMat->Rows());
385         sourceindex.Resize(ek.fConstrMat->Rows());
386         int numnod = ek.fConstrConnect.NElements();
387         for(int in=0; in<numnod; in++) {
388             int npindex = ek.fConstrConnect[in];
389             TPZConnect &np = fMesh->ConnectVec()[npindex];
390             int blocknumber = np.SequenceNumber();
391             int firsteq = fMesh->Block().Position(blocknumber);
392             int ndf = fMesh->Block().Size(blocknumber);
393             if(np.HasDependency()) {
394                 fullmatindex += ndf;
395                 continue;
396             }
397
398             for(int idf=0; idf<ndf; idf++) {
399                 sourceindex[destindex] = fullmatindex++;
400                 destinationindex[destindex++] = firsteq+idf;
401             }
```

```
402           }
403
404           sourceindex.Resize(destindex);
405           destinationindex.Resize(destindex);
406           //ek.Print(*fMesh,cout);
407           stiffness.AddKel(*ek.fConstrMat,sourceindex,destinationindex);
408           rhs.AddFel(*ef.fConstrMat,sourceindex,destinationindex);
409       }
410 }
```

3.6.3.3 template<class front> void TPZFrontStructMatrix< front >::AssembleNew (TPZMatrix & *stiffness*, TPZFMatrix & *rhs*)

Assemble a stiffness matrix according to rhs

Parameters:

stiffness Stiffness matrix to assembled

rhs Matrix contaning ???

Definition at line 189 of file TPZFrontStructMatrix.c.

References fElementOrder, TPZStructMatrix::fMesh, and OrderElement.

```
189                                                                                                    {
190
191   int iel;
192   int numel = 0, nelem = fMesh->NElements();
193   TPZElementMatrix ek,ef;
194   int destinationstore[100];
195   TPZManVector<int> destinationindex(0,destinationstore,100);
196   int sourcestore[100];
197   TPZManVector<int> sourceindex(0,sourcestore,100);
198   REAL stor1[1000],stor2[1000],stor3[100],stor4[100];
199   ek.fMat = new TPZFMatrix(0,0,stor1,1000);
200   ek.fConstrMat = new TPZFMatrix(0,0,stor2,1000);
201   ef.fMat = new TPZFMatrix(0,0,stor3,100);
202   ef.fConstrMat = new TPZFMatrix(0,0,stor4,100);
203
204   TPZAdmChunkVector<TPZCompEl *> &elementvec = fMesh->ElementVec();
205
206
208   TPZVec<int> elorder(fMesh->NEquations(),0);
209
210   OrderElement();
211
212
213   for(iel=0; iel < nelem; iel++) {
214
215     if(fElementOrder[iel] < 0) continue;
```

```
216     TPZCompEl *el = elementvec[fElementOrder[iel]];
217     if(!el) continue;
218     //          int dim = el->NumNodes();
219
220     //Builds elements stiffness matrix
221     el->CalcStiff(ek,ef);
222       //ek.fMat->Print(out);
223     //ef.fMat->Print();
224     if(!(numel%20)) cout << endl << numel;
225 //    if(!(numel%20)) cout << endl;
226     cout << '*';
227     cout.flush();
228     numel++;
229
230     if(!el->HasDependency()) {
231       //ek.fMat->Print("stiff has no constraint",test);
232       //ef.fMat->Print("rhs has no constraint",test);
233       //test.flush();
234       destinationindex.Resize(ek.fMat->Rows());
235       int destindex = 0;
236       int numnod = ek.NConnects();
237       for(int in=0; in<numnod; in++) {
238          int npindex = ek.ConnectIndex(in);
239          TPZConnect &np = fMesh->ConnectVec()[npindex];
240          int blocknumber = np.SequenceNumber();
241          int firsteq = fMesh->Block().Position(blocknumber);
242          int ndf = fMesh->Block().Size(blocknumber);
243          for(int idf=0; idf<ndf; idf++) {
244            destinationindex[destindex++] = firsteq+idf;
245          }
246       }
247       //ek.Print(*fMesh,cout);
248       stiffness.AddKel(*ek.fMat,destinationindex);
249       rhs.AddFel(*ef.fMat,destinationindex);                  //  ??????????? Erro
250     }
251     else {
252       // the element has dependent nodes
253       el->ApplyConstraints(ek,ef);
254       //ek.fMat->Print("stif no constraint",test);
255       //ek.fConstrMat->Print("stif constrained",test);
256       //ef.fMat->Print("rhs no constraint",test);
257       //ef.fConstrMat->Print("rhs constrained",test);
258       //test.flush();
259       //test << "sum of columns\n";
260       int destindex = 0;
261       int fullmatindex = 0;
262       destinationindex.Resize(ek.fConstrMat->Rows());
263       sourceindex.Resize(ek.fConstrMat->Rows());
264       int numnod = ek.fConstrConnect.NElements();
265       for(int in=0; in<numnod; in++) {
266          int npindex = ek.fConstrConnect[in];
267          TPZConnect &np = fMesh->ConnectVec()[npindex];
268          int blocknumber = np.SequenceNumber();
269          int firsteq = fMesh->Block().Position(blocknumber);
270          int ndf = fMesh->Block().Size(blocknumber);
```

```
271           if(np.HasDependency()) {
272             fullmatindex += ndf;
273             continue;
274           }
275           for(int idf=0; idf<ndf; idf++) {
276             sourceindex[destindex] = fullmatindex++;
277             destinationindex[destindex++] = firsteq+idf;
278           }
279         }
280         sourceindex.Resize(destindex);
281         destinationindex.Resize(destindex);
282         //ek.Print(*fMesh,cout);
283         stiffness.AddKel(*ek.fConstrMat,sourceindex,destinationindex);
284         rhs.AddFel(*ef.fConstrMat,sourceindex,destinationindex);
285 /*
286 if(ek.fConstrMat->Decompose_LU() != -1) {
287     el->ApplyConstraints(ek,ef);
288     ek.Print(*this,check);
289     check.flush();
290 }
291 */
292     }
293
294   }//fim for iel
295   cout << endl;
296 }
```

3.6.3.4 template<class front> TPZStructMatrix * TPZFrontStructMatrix< front >::Clone () [virtual]

It clones a TPZFrontStructMatrix

Reimplemented from **TPZStructMatrix** (p. 125).

Reimplemented in **TPZParFrontStructMatrix** (p. 99).

Definition at line 85 of file TPZFrontStructMatrix.c.

```
85                                                  {
86
87     return new TPZFrontStructMatrix<front>(fMesh);
88 }
```

3.6.3.5 template<class front> TPZMatrix * TPZFrontStructMatrix< front >::Create () [virtual]

Returns a pointer to TPZMatrix

Reimplemented from **TPZStructMatrix** (p. 125).

Definition at line 72 of file TPZFrontStructMatrix.c.

References TPZStructMatrix::fMesh, GetNumElConnected, and TPZFrontMatrix::SetNumElConnected.

```
72                                                  {
73
74      TPZVec <int> numelconnected(fMesh->NEquations(),0);
75      TPZFrontMatrix<TPZFileEqnStorage, front> *mat = new TPZFrontMatrix<TPZFileEqnStorage,front>(fMesh->NEquat:
76
77      GetNumElConnected(numelconnected);
78      mat->SetNumElConnected(numelconnected);
79      return mat;
80
81      //return (0);
82 }
```

3.6.3.6 template<class front> TPZMatrix * TPZFrontStructMatrix< front >::CreateAssemble (TPZFMatrix & *rhs*) [virtual]

Returns a pointer to TPZMatrix.

This is a mandatory function, it is neded by all StructMatrix.

Except in frontal matrices, the returned matrix is not in its decomposed form.

Parameters:
rhs Load matrix

Reimplemented from **TPZStructMatrix** (p. 125).

Reimplemented in **TPZParFrontStructMatrix** (p. 100).

Definition at line 173 of file TPZFrontStructMatrix.c.

References Assemble, TPZStructMatrix::fMesh, GetNumElConnected, OrderElement, and TPZFrontMatrix::SetNumElConnected.

```
173                                                                  {
174
175      TPZVec <int> numelconnected(fMesh->NEquations(),0);
176      //TPZFrontMatrix<TPZStackEqnStorage, front> *mat = new TPZFrontMatrix<TPZStackEqnStorage, front>(fMesh->I
177
178      TPZFrontMatrix<TPZFileEqnStorage, front> *mat = new TPZFrontMatrix<TPZFileEqnStorage, front>(fMesh->NEqu:
179      GetNumElConnected(numelconnected);
180      mat->SetNumElConnected(numelconnected);
181
182      OrderElement();
183
184      Assemble(*mat,rhs);
185      return mat;
186 }
```

3.6.3.7 template<class front> void TPZFrontStructMatrix< front >::GetNumElConnected (TPZVec< int > & *numelconnected*) [protected]

Returns a vector containing all elements connected to a degree of freedom.

Parameters:

numelconnected Vector containing the number of connections for every ith dof.

Definition at line 26 of file TPZFrontStructMatrix.c.

References TPZStructMatrix::fMesh.

Referenced by Create, TPZParFrontStructMatrix::CreateAssemble, and CreateAssemble.

```
26                                                                      {
27         int ic;
28
29         cout << "Nmero de Equaes -> " << fMesh->NEquations() << endl;
30         cout.flush();
31
32         fMesh->ComputeNodElCon();
33
34         for(ic=0; ic<fMesh->ConnectVec().NElements(); ic++) {
35                 TPZConnect &cn = fMesh->ConnectVec()[ic];
36                 if(cn.HasDependency()) continue;
37                 int seqn = cn.SequenceNumber();
38                 if(seqn < 0) continue;
39                 int firsteq = fMesh->Block().Position(seqn);
40                 int lasteq = firsteq+fMesh->Block().Size(seqn);
41                 int ind;
42                 for(ind=firsteq;ind<lasteq;ind++) numelconnected[ind] = fMesh->ConnectVec()[ic]
43         }
44 /*cout << "GetNumElConnected::numelconnected : ";
45 int i;
46 for(i=0; i<numelconnected.NElements(); i++) cout << numelconnected[i] << ' ';
47 cout << endl;
48 cout.flush();*/
49 }
```

3.6.3.8 template<class front> void TPZFrontStructMatrix< front >::OrderElement () [protected]

It is applied over fElementOrder putting it in the correct order.

Definition at line 90 of file TPZFrontStructMatrix.c.

Referenced by AssembleNew, TPZParFrontStructMatrix::CreateAssemble, and CreateAssemble.

```
91 {
92         int numelconnected = 0;
93         int nconnect = fMesh->ConnectVec().NElements();
94         int ic;
95         //firstelconnect contains the first element index in the elconnect vector
96         TPZVec<int> firstelconnect(nconnect+1);
97         firstelconnect[0] = 0;
98         for(ic=0; ic<nconnect; ic++) {
99                 numelconnected += fMesh->ConnectVec()[ic].NElConnected();
100                 firstelconnect[ic+1] = firstelconnect[ic]+fMesh->ConnectVec()[ic].NElConnected();
101         }
102 //cout << "numelconnected " << numelconnected << endl;
103 //cout << "firstelconnect ";
104 //  for(ic=0; ic<nconnect; ic++) cout << firstelconnect[ic] << ' ';
105         TPZVec<int> elconnect(numelconnected);
106         int el;
107         TPZCompEl *cel;
108         for(el=0; el<fMesh->ElementVec().NElements(); el++) {
109                 cel = fMesh->ElementVec()[el];
110                 if(!cel) continue;
111                 TPZStack<int> connectlist(100);
112                 cel->BuildConnectList(connectlist);
113                 int nc = connectlist.NElements();
114                 int ic;
115                 for(ic=0; ic<nc; ic++) {
116                         int cindex = connectlist[ic];
117                         elconnect[firstelconnect[cindex]] = el;
118                         firstelconnect[cindex]++;
119                 }
120         }
121 //  for(ic=0; ic<numelconnected; ic++) cout << elconnect[ic] << endl;
122         firstelconnect[0] = 0;
123         for(ic=0; ic<nconnect; ic++) {
124                 firstelconnect[ic+1] = firstelconnect[ic]+fMesh->ConnectVec()[ic].NElConnected();
125         }
126   //cout << "elconnect\n";
127   int no;
128 //  for(no=0; no< fMesh->ConnectVec().NElements(); no++) {
129         //cout << "no numero " << no << ' ' << " seq num " << fMesh->ConnectVec()[no].SequenceNumber() << ' '
130 //       for(ic=firstelconnect[no]; ic<firstelconnect[no+1];ic++) cout << elconnect[ic] << ' ';
131         //cout << endl;
132 //  }
133         fElementOrder.Resize(fMesh->ElementVec().NElements(),-1);
134         fElementOrder.Fill(-1);
135         TPZVec<int> nodeorder(fMesh->ConnectVec().NElements(),-1);
136         firstelconnect[0] = 0;
137         for(ic=0; ic<nconnect; ic++) {
138                 int seqnum = fMesh->ConnectVec()[ic].SequenceNumber();
139                 if(seqnum >= 0) nodeorder[seqnum] = ic;
140         }
141 //  cout << "nodeorder ";
142 /*  for(ic=0; ic<fMesh->ConnectVec().NElements(); ic++) cout << nodeorder[ic] << ' ';
143   cout << endl;
144   cout.flush();*/
145         int seq;
```

```
146         int elsequence = 0;
147         TPZVec<int> elorderinv(fMesh->ElementVec().NElements(),-1);
148         for(seq=0; seq<nconnect; seq++) {
149                 ic = nodeorder[seq];
150                 if(ic == -1) continue;
151                 int firstind = firstelconnect[ic];
152                 int lastind = firstelconnect[ic+1];
153                 int ind;
154                 for(ind=firstind; ind<lastind; ind++) {
155                         el = elconnect[ind];
156                         if(elorderinv[el]==-1) elorderinv[el] = elsequence++;
157                 }
158         }
159 //  cout << "elorderinv ";
160 //  for(seq=0;seq<fMesh->ElementVec().NElements();seq++) cout << elorderinv[seq] << ' ';
161 //  cout << endl;
162         elsequence = 0;
163         for(seq=0;seq<fMesh->ElementVec().NElements();seq++) {
164                 if(elorderinv[seq] == -1) continue;
165                 fElementOrder[elorderinv[seq]] = seq;
166         }
167 //  cout << "elorder" << endl;
168 //  for(ic=0; ic<fMesh->ElementVec().NElements(); ic++) cout << elorder[ic] << endl;
169
170 }
```

3.6.4 Member Data Documentation

3.6.4.1 template<class front> TPZVec<int> TPZFrontStructMatrix::fElementOrder [protected]

This vector contains an ordered list.

The elements must be asssembled in that order so the frontal works on its best performance

Definition at line 38 of file TPZFrontStructMatrix.h.

Referenced by Assemble, AssembleNew, TPZParFrontStructMatrix::ElementAssemble, and TPZParFrontStructMatrix::GlobalAssemble.

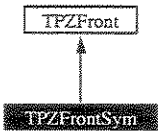
The documentation for this class was generated from the following files:

- **TPZFrontStructMatrix.h**
- **TPZFrontStructMatrix.c**

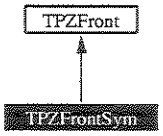
3.7 TPZFrontSym Class Reference

`#include <TPZFrontSym.h>`

Inheritance diagram for TPZFrontSym:



Collaboration diagram for TPZFrontSym:



Public Methods

- char * **GetMatrixType** ()
- **~TPZFrontSym** ()
- **TPZFrontSym** ()
- **TPZFrontSym** (int GlobalSize)
- void **DecomposeEquations** (int mineq, int maxeq, **TPZEqnArray** &result)
- void **SymbolicDecomposeEquations** (int mineq, int maxeq)
- void **SymbolicAddKel** (TPZVec< int > &destinationindex)
- void **Compress** ()
- void **Expand** (int largefrontsize)
- REAL & **Element** (int i, int j)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &destinationindex)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &sourceindex, TPZVec< int > &destinationindex)
- int **NFree** ()
- void **Reset** (int GlobalSize=0)
- void **AllocData** ()
- void **Print** (const char *name, ostream &out=cout)
- void **PrintGlobal** (const char *name, ostream &out=cout)
- DecomposeType **GetDecomposeType** () const

Static Public Methods

- void **main** ()

3.7.1 Detailed Description

The Front matrix itself.

It is controled by **TPZFrontMatrix** (p. 43).

TPZFrontSym is a symmetrical matrix.

It uses a Cholesky decomposition scheme.

Definition at line 39 of file TPZFrontSym.h.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 TPZFrontSym::~TPZFrontSym ()

Simple destructor

Definition at line 329 of file TPZFrontSym.c.

```
329 {}
```

3.7.2.2 TPZFrontSym::TPZFrontSym ()

Simple constructor

Definition at line 326 of file TPZFrontSym.c.

```
326                         {
327         fDecomposeType=ECholesky;
328 }
```

3.7.2.3 TPZFrontSym::TPZFrontSym (int *GlobalSize*)

Constructor with a initial size parameter

Definition at line 322 of file TPZFrontSym.c.

```
322                                         : TPZFront(GlobalSize)
323 {
324         fDecomposeType=ECholesky;
325 }
```

3.7.3 Member Function Documentation

3.7.3.1 void TPZFrontSym::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *sourceindex*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Definition at line 201 of file TPZFrontSym.c.

References Element.

```
202 {
203         int i, j, ilocal, jlocal, nel;
204         nel=sourceindex.NElements();
205         for (i = 0; i < nel; i++) {
206             // message #1.1.1 to this:TPZFront
207             ilocal = this->Local(destinationindex[i]);
208             for (j = i; j < nel; j++) {
209                 // message #1.1.2.1 to this:TPZFront
210                 jlocal = this->Local(destinationindex[j]);
211
212                 // message #1.1.2.2 to this:TPZFront
213                 this->Element(ilocal, jlocal)+=elmat(sourceindex[i],sourceindex[j]);
214             }
215         }
216     }
```

3.7.3.2 void TPZFrontSym::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Definition at line 217 of file TPZFrontSym.c.

References Element.

Referenced by main.

```
218 {
219      int i, j, ilocal, jlocal, nel;
220      nel = destinationindex.NElements();
221      for(i=0;i<nel;i++){
222           ilocal = this->Local(destinationindex[i]);
223           for(j=i;j<nel;j++) {
224                jlocal=this->Local(destinationindex[j]);
225                this->Element(ilocal, jlocal)+=elmat(i,j);
226           }
227      }
228 /*
229         output << "Dest Index " ;
```

```
230         for(i=0;i<nel;i++) output << destinationindex[i] << " ";
231         output << endl;
232         elmat.Print("Element matrix ",output);
233         PrintGlobal("After Assemb..." , output);
234 */
235 }
```

3.7.3.3 void TPZFrontSym::AllocData ()

Allocates data for Front

Definition at line 80 of file TPZFrontSym.c.

References TPZFront::fData, TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fMaxFront.

Referenced by main.

```
81 {
82         fData.Resize(fMaxFront*(fMaxFront+1)/2);
83         fGlobal.Fill(-1);
84         fData.Fill(0.);
85         fFront=0;
86         //fLocal.Fill(-1);
87 }
```

3.7.3.4 void TPZFrontSym::Compress ()

Compress data structure

Definition at line 250 of file TPZFrontSym.c.

References Element, TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fLocal.

Referenced by main.

```
250                         {
251 //      PrintGlobal("Before COmpress",output);
252 //      Print("Before Compress", cout);
253         TPZStack <int> from;
254         int nfound;
255         int i, j;
256         for(i = 0; i < fFront; i++){
257                 if(fGlobal[i] != -1) from.Push(i);
258         }
259
264         nfound = from.NElements();
265         for(i=0;i<nfound;i++) {
266                 fGlobal[i]=fGlobal[from[i]];
```

```
267                 //fGlobal[from[i]] = -1;
268                 fLocal[fGlobal[i]] = i;
269         }
270         for(;i<fGlobal.NElements();i++) fGlobal[i] = -1;
271
272         if(nfound+fFree.NElements()!=fFront) cout << "TPZFront.Compress inconsistent data structure\n";
273         int frontold = fFront;
274         fFront = nfound;
275         fFree.Resize(0);
276         fGlobal.Resize(fFront);
277         if(fData.NElements()==0) return;
278
279         for(j = 0; j < nfound; j++){
280                 for(i = 0; i <= j; i++){
281                         Element(i,j) = Element(from[i], from[j]);
282                 }
283 //              fGlobal[i] = fGlobal[from[i]];
284 //              fLocal[fGlobal[i]] = i;
285         }
286         for(;j<frontold;j++) {
287                 for(i=0;i<= j; i++) Element(i,j) = 0.;
288         }
289
290 //      Print("After Compress", cout);
291 //      PrintGlobal("After Compress",output);
292 }
```

3.7.3.5 void TPZFrontSym::DecomposeEquations (int *mineq*, int *maxeq*, TPZEqnArray & *result*)

Decompose these equations and put the result in eqnarray Default decompose method is Cholesky

Parameters:

mineq Starting index of equations to be decomposed

maxeq Finishing index of equations to be decomposed

eqnarray Result of decomposition

Definition at line 310 of file TPZFrontSym.c.

References TPZEqnArray::Reset, and TPZEqnArray::SetSymmetric.

Referenced by main.

```
310                                                                                 {
311         // message #1.1 to eqnarray:TPZEqnArray
312         int ieq;
313         eqnarray.Reset();
314         eqnarray.SetSymmetric();
315
```

```
316         for (ieq = mineq; ieq <= maxeq; ieq++) {
317             // message #1.2.1 to this:TPZFront
318             this->DecomposeOneEquation(ieq, eqnarray);
319 //                      this->Print("Teste.txt",output);
320         }
321     }
```

3.7.3.6 REAL& TPZFrontSym::Element (int *i*, int *j*) [inline]

Returns ith, jth element of matrix. @associates <{mat(sourceindex[i],sourceindex[j])}> @semantics +=

Definition at line 92 of file TPZFrontSym.h.

References TPZFront::fData.

Referenced by AddKel, Compress, and Print.

```
92                                      {
93         if(i>j){
94                 int i_temp=i;
95                 i=j;
96                 j=i_temp;
97                 //cout << "Changing row column indexes !"<<endl;
98         }
99         return fData[(j*(j+1))/2+i];
100 }
```

3.7.3.7 void TPZFrontSym::Expand (int *largefrontsize*)

Expand the front matrix

Definition at line 246 of file TPZFrontSym.c.

References TPZFront::fData, and TPZFront::fMaxFront.

```
246                                         {
247         fData.Resize(larger*(larger+1)/2,0.);
248         fMaxFront = larger;
249 }
```

3.7.3.8 DecomposeType TPZFrontSym::GetDecomposeType () const

Returns decomposition type

Definition at line 20 of file TPZFrontSym.c.

```
21 {
22         return fDecomposeType;
23 }
```

3.7.3.9 char * TPZFrontSym::GetMatrixType ()

Returns its type

Definition at line 424 of file TPZFrontSym.c.

```
424                                 {
425     return "Symmetric matrix";
426 }
```

3.7.3.10 void TPZFrontSym::main () [static]

Static main used for testing

Reimplemented from **TPZFront** (p. 36).

Definition at line 331 of file TPZFrontSym.c.

References AddKel, AllocData, Compress, DecomposeEquations, TPZEqnArray::EqnBackward, TPZEqnArray::EqnForward, TPZEqnArray::Print, SymbolicAddKel, and SymbolicDecomposeEquations.

```
332 {
333         int i, j;
337         int matsize=6;
338         TPZFMatrix TestMatrix(matsize,matsize);
339         for(i=0;i<matsize;i++) {
340                 for(j=i;j<matsize;j++) {
341                         int random = rand();
342                         double rnd = (random*matsize)/0x7fff;
343                         TestMatrix(i,j)=rnd;
344                         TestMatrix(j,i)=TestMatrix(i,j);
345                         if(i==j) TestMatrix(i,j)=6000.;
346                 }
347         }
348
349         TPZFMatrix Prova;
350         Prova=TestMatrix;
351
352 //      Prova.Decompose_Cholesky();
353         Prova.Print("TPZFMatrix Cholesky");
354
355         TPZFrontSym TestFront(matsize);
356
357
358         TPZVec<int> DestIndex(matsize);
```

```
359         for(i=0;i<matsize;i++) DestIndex[i]=i;
360
361         TestFront.SymbolicAddKel(DestIndex);
362         TestFront.SymbolicDecomposeEquations(0,matsize-1);
363
364         char * OutFile;
365         OutFile = "TPZFrontSymTest.txt";
366
367         ofstream output(OutFile,ios::app);
368
369         TestFront.Compress();
370
371         TestFront.AllocData();
372
373         TestFront.AddKel(TestMatrix, DestIndex);
374         TPZEqnArray Result;
375
376         /*TestFront.DecomposeEquations(0,0,Result);
377
378         TestFront.Print(OutFile, output);
379
380         ofstream outeqn("TestEQNArray.txt",ios::app);
381         Result.Print("TestEQNArray.txt",outeqn);
382
383         TestFront.Compress();
384
385         TestFront.Print(OutFile, output);
386 */
387         TestFront.DecomposeEquations(0,matsize-1,Result);
388         ofstream outeqn("TestEQNArray.txt",ios::app);
389
390         Result.Print("TestEQNArray.txt",outeqn);
391
392
393         TPZFMatrix Load(matsize);
394
395         for(i=0;i<matsize;i++) {
396                 int random = rand();
397                 double rnd = (random*matsize)/0x7fff;
398                 Load(i,0)=rnd;
399         }
400
401         TPZFMatrix Load_2(matsize);
402         Load_2=Load;
403
404 //      Prova.Subst_Forward(&Load);
405 //      Prova.Subst_Backward(&Load);
406
407
408         DecomposeType decType = ECholesky;
409         Prova.SolveDirect(Load, decType);
410
411         Load.Print();
412         //TestFront.Print(OutFile, output);
413
```

```
414         Result.EqnForward(Load_2, decType);
415         Result.EqnBackward(Load_2, decType);
416
417         Load_2.Print("Eqn");
418
419
420
421 }
```

3.7.3.11 int TPZFrontSym::NFree ()

Returns the number of free equations

Reimplemented from **TPZFront** (p. 38).

Definition at line 98 of file TPZFrontSym.c.

References TPZFront::fLocal.

```
99 {
100         int i;
101         int free_eq=0;
102         for(i=0;i<fLocal.NElements();i++)
103         {
104                 if(fLocal[i]==-1){
105                         free_eq=free_eq+1;
106                 }
107         }
108         return free_eq;
109 }
```

3.7.3.12 void TPZFrontSym::Print (const char * *name*, ostream & *out* = cout)

It prints **TPZFront** (p. 34) data

Reimplemented from **TPZFront** (p. 38).

Definition at line 41 of file TPZFrontSym.c.

References Element, TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, TPZFront::fLocal, and TPZFront::fMaxFront.

```
42 {
43   if(name) out << name << endl;
44   int i,j,loop_limit;
45
46
47   out <<  "Frontal Matrix Size          "<< fFront << endl;
48   out <<  "Maximum Frontal Matrix Size  "<< fMaxFront << endl;
```

```
49
50   out << endl;
51   out <<  "Local Indexation "<< endl;
52   out <<  "Position  "<<" Local index"<< endl;
53   for(i=0;i<fLocal.NElements();i++){
54     out <<   i <<  "          "  << fLocal[i] << endl;
55   }
56
57   out << endl;
58   out <<  "Global Indexation "<< endl;
59   out <<  "Position  "<<" Global index"<< endl;
60
61   for(i=0;i<fGlobal.NElements();i++){
62     out <<   i <<  "            "<< fGlobal[i] << endl;
63   }
64
65   out << endl;
66   out <<  "Local Freed Equatoins  "<< fFree.NElements() << endl;
67   out <<  "position "<<   "Local Equation "<<endl;
68   loop_limit=fFree.NElements();
69   for(i=0;i<loop_limit;i++){
70     out <<  i <<  "             " << fFree[i] << endl;
71   }
72   out <<  "Frontal Matrix "<< endl;
73   if(fData.NElements() > 0) {
74           for(i=0;i<fFront;i++){
75                   for(j=0;j<fFront;j++) out << ((i<j) ? Element(i,j) : Element(j,i)) <<  " ";
76                 out << endl;
77           }
78   }
79 }
```

3.7.3.13 void TPZFrontSym::Reset (int *GlobalSize* = 0)

Resets data structure

Reimplemented from **TPZFront** (p. 39).

Definition at line 88 of file TPZFrontSym.c.

References TPZFront::fData, TPZFront::fFree, TPZFront::fFront, TPZFront::fGlobal, TPZFront::fLocal, and TPZFront::fMaxFront.

```
89 {
90         fData.Resize(0);
91         fFree.Resize(0);
92         fFront=0;
93         fGlobal.Resize(0);
94         fLocal.Resize(GlobalSize);
95         fLocal.Fill(-1);
96         fMaxFront=0;
97 }
```

3.7.3.14 void TPZFrontSym::SymbolicAddKel (TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix using the indexes to compute the frontwidth

Reimplemented from **TPZFront** (p. 39).

Definition at line 293 of file TPZFrontSym.c.

References TPZFront::fFront, TPZFront::fGlobal, and TPZFront::fMaxFront.

Referenced by main.

```
294 {
295         int i, loop_limit, aux;
296         loop_limit=destinationindex.NElements();
297         for(i=0;i<loop_limit;i++){
298                 aux=destinationindex[i];
299                 Local(aux);
300                 fFront = fGlobal.NElements();
301         }
302         fMaxFront=(fFront<fMaxFront)?fMaxFront:fFront;
303
304 }
```

3.7.3.15 void TPZFrontSym::SymbolicDecomposeEquations (int *mineq*, int *maxeq*)

Decompose these equations in a symbolic way and store freed indexes in fFree

Parameters:

mineq Initial equation index

maxeq Final equation index

Reimplemented from **TPZFront** (p. 40).

Definition at line 305 of file TPZFrontSym.c.

Referenced by main.

```
306 {
307         int i;
308         for(i=mineq;i<=maxeq;i++) FreeGlobal(i);
309 }
```

The documentation for this class was generated from the following files:

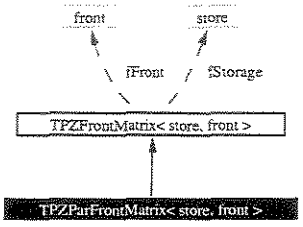
- **TPZFrontSym.h**
- **TPZFrontSym.c**

3.8 TPZParFrontMatrix Class Template Reference

`#include <TPZParFrontMatrix.h>`

Inheritance diagram for TPZParFrontMatrix:

Collaboration diagram for TPZParFrontMatrix:



Public Methods

- **~TPZParFrontMatrix** ()
- **TPZParFrontMatrix** ()
- **TPZParFrontMatrix** (int globalsize)
- void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &sourceindex, TPZVec< int > &destinationindex)
- virtual void **AddKel** (TPZFMatrix &elmat, TPZVec< int > &destinationindex)
- void **FinishWriting** ()

Static Public Methods

- void * **WriteFile** (void *t)

3.8.1 Detailed Description

template<class store, class front> class TPZParFrontMatrix< store, front >

FrontMatrix with parallel techniques included.

Is derived from **TPZFrontMatrix** (p. 43).

As its base class it is also a template class. The parameters store and front can assume the values **TPZFileEqnStorage** (p. 22) or **TPZStackEqnStorage** (p. 120) for store and **TPZFrontSym** (p. 79) or **TPZFrontNonSym** (p. 55) for front.

Definition at line 33 of file TPZParFrontMatrix.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 template<class store, class front> TPZParFrontMatrix< store, front >::~TPZParFrontMatrix ()

Simple Destructor

Definition at line 60 of file TPZParFrontMatrix.c.

```
60                                                      {
61 }
```

3.8.2.2 template<class store, class front> TPZParFrontMatrix< store, front >::TPZParFrontMatrix ()

Simple Constructor

Definition at line 27 of file TPZParFrontMatrix.c.

```
27                                                      :
28     fFinish(0)
29 {
30     fEqnStack.Resize(0);
31     pthread_mutex_t mlocal = PTHREAD_MUTEX_INITIALIZER;
32     fwritelock = mlocal;
33     pthread_cond_t clocal = PTHREAD_COND_INITIALIZER;
34     fwritecond = clocal;
35 /*     fFront.Reset();
36        fStorage.Reset();
37        fNumElConnected.Resize(0);
38        fLastDecomposed = -1;
```

```
39          fNumEq=0;
40          */
41 }
```

3.8.2.3 template<class store, class front> TPZParFrontMatrix< store, front >::TPZParFrontMatrix (int *globalsize*)

Constructor with a globalsize parameter

Parameters:

globalsize Indicates initial global size

Definition at line 44 of file TPZParFrontMatrix.c.

```
44
45      TPZFrontMatrix<store, front>(globalsize),
46      fFinish(0)
47 {
48      fEqnStack.Resize(0);
49      pthread_mutex_t mlocal = PTHREAD_MUTEX_INITIALIZER;
50      fwritelock = mlocal;
51      pthread_cond_t clocal = PTHREAD_COND_INITIALIZER;
52      fwritecond = clocal;
53 /*      fFront.Reset(globalsize);
54         fStorage.Reset();
55         fNumElConnected.Resize(0);
56         fLastDecomposed = -1;
57         fNumEq=globalsize;*/
58 }
```

3.8.3 Member Function Documentation

3.8.3.1 template<class store, class front> void TPZParFrontMatrix< store, front >::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *destinationindex*) [virtual]

Add a contribution of a stiffness matrix putting it on destination indexes position

Parameters:

elmat Member stiffness matrix beeing added

destinationindex Positioning of such members on global stiffness matrix

Reimplemented from **TPZFrontMatrix** (p. 46).

Definition at line 64 of file TPZParFrontMatrix.c.

References TPZFrontMatrix::CheckCompress, TPZFrontMatrix::EquationsToDecompose, TPZFrontMatrix::fFront, and FinishWriting.

```
65 {
66
67          // message #1.3 to fFront:TPZFront
68          fFront.AddKel(elmat, destinationindex);
69     /*     cout << "destination index" << endl;
70          int i;
71          for(i=0;i<destinationindex.NElements();i++) cout << destinationindex[i] << " ";
72          cout << endl;
73          cout.flush();
74          elmat.Print("Element Matrix");
75          */
76                int mineq, maxeq;
77                EquationsToDecompose(destinationindex, mineq, maxeq);
78                TPZEqnArray *AuxEqn = new TPZEqnArray;
79                if(maxeq >= mineq) {
80 //              if(!(maxeq%10)){
81 //                    cout << (100*maxeq/fNumEq) << " % Decomposed" << endl;
82 //                    cout.flush();
83  //             }
84
85                     fFront.DecomposeEquations(mineq,maxeq,*AuxEqn);
86                     CheckCompress();
87                     pthread_mutex_lock(&fwritelock);
88                     fEqnStack.Push(AuxEqn);
89                     if(maxeq == Rows()-1){
90                    cout << "Decomposition finished" << endl;
91                 cout.flush();
92                     FinishWriting();
93                            //fStorage.ReOpen();
94                     }
95                     pthread_mutex_unlock(&fwritelock);
96                     pthread_cond_signal(&fwritecond);
97                }
98                fDecomposed = fFront.GetDecomposeType();
99 }
```

3.8.3.2 template<class store, class front> void TPZParFrontMatrix< store, front >::AddKel (TPZFMatrix & *elmat*, TPZVec< int > & *sourceindex*, TPZVec< int > & *destinationindex*)

Add a contribution of a stiffness matrix

Parameters:

elmat Member stiffness matrix beeing added

sourceindex Sorce position of values on member stiffness matrix

destinationindex Positioning of such members on global stiffness matrix

Reimplemented from **TPZFrontMatrix** (p. 45).

Definition at line 101 of file TPZParFrontMatrix.c.

References TPZFrontMatrix::CheckCompress, TPZFrontMatrix::EquationsToDecompose, TPZFrontMatrix::fFront, and FinishWriting.

```
102 {
103         fFront.AddKel(elmat, sourceindex, destinationindex);
104 //      EquationsToDecompose(destinationindex);
105  //         cout << "AddKel::destination index 2" << endl;
106      int i;
107 //          for(i=0;i<destinationindex.NElements();i++) cout << destinationindex[i] << " ";
108  //         cout << endl;
109  //         cout.flush();
110 //          elmat.Print("AddKel: Element Matrix 2");
111         int mineq, maxeq;
112         EquationsToDecompose(destinationindex, mineq, maxeq);
113         TPZEqnArray *AuxEqn = new TPZEqnArray;
114         if(maxeq >= mineq) {
115 //           if(!(maxeq%10)){
116 //               cout << (100*maxeq/fNumEq) << " % Decomposed" << endl;
117 //               cout.flush();
118 //          }
119
120                 fFront.DecomposeEquations(mineq,maxeq,*AuxEqn);
121                 CheckCompress();
122                 //fStorage.AddEqnArray(&AuxEqn);
123                 //adds an equation to a stack!!!
124                 //some sort of lock here
125 //          fEqnStack->Push(&AuxEqn);
126                 pthread_mutex_lock(&fwritelock);
127                 fEqnStack.Push(AuxEqn);
128                 if(maxeq == Rows()-1){
129             //check if writeing is over and closes file
130                cout << endl << "Decomposition finished" << endl;
131                cout.flush();
132                         FinishWriting();
133                         fFront.Reset(0);
134                         //fStorage.ReOpen();
135                 }
136                 pthread_mutex_unlock(&fwritelock);
137         pthread_cond_signal(&fwritecond);
138         }
139         fDecomposed = fFront.GetDecomposeType();
140 }
```

3.8.3.3 template<class store, class front> void TPZParFrontMatrix< store, front >::FinishWriting ()

Sets the flag fFinish to its true value

Reimplemented from **TPZFrontMatrix** (p. 48).

Definition at line 143 of file TPZParFrontMatrix.c.

Referenced by AddKel.

```
143                                                    {
144     cout << endl << "FinishWriting" << endl;
145     cout.flush();
146     fFinish = 1;
147 }
```

3.8.3.4 template<class store, class front> void * TPZParFrontMatrix< store, front >::WriteFile (void * t) [static]

Used in an independent thread to write decomposed equations to a binary file

Definition at line 150 of file TPZParFrontMatrix.c.

References fEqnStack, fFinish, TPZFrontMatrix::fStorage, fwritecond, and fwritelock.

Referenced by TPZParFrontStructMatrix::CreateAssemble.

```
150                                                    {
151     TPZParFrontMatrix<store, front> *parfront = (TPZParFrontMatrix<store, front>*) t;
152     cout << endl << "Entering Decomposition" << endl;
153     cout.flush();
154     int nlocal=0;
155     while(1){
156         TPZStack<TPZEqnArray *> local;
157         pthread_mutex_lock(&parfront->fwritelock);
158         if(parfront->fEqnStack.NElements() == 0){
159             if(parfront->fFinish == 1) {
160                 cout << "Leaving WHILE" << endl;
161                 cout.flush();
162                 break;
163             }
164             pthread_cond_wait(&parfront->fwritecond, &parfront->fwritelock);
165         }
166
167         local = parfront->fEqnStack;
168         parfront->fEqnStack.Resize(0);
169         pthread_mutex_unlock(&parfront->fwritelock);
170         int neqn = local.NElements();
171
172 /*          nlocal++;
173         if(!(nlocal%200)) cout << endl << "         Decomposing  " << neqn << " " << nlocal << " on thread '
174         if(!(nlocal%20)) cout << nlocal << endl;
175         cout << '#';
176         cout.flush();
177 */
178         int eq;
```

```
179           for(eq=0; eq<neqn; eq++) {
180                parfront->fStorage.AddEqnArray(local[eq]);
181                delete local[eq];
182           }
183
184
185      }
186      parfront->fStorage.FinishWriting();
187      parfront->fStorage.ReOpen();
188      parfront->fFinish = 0;
189      pthread_mutex_unlock(&parfront->fwritelock);
190      return (0);
191 }
```

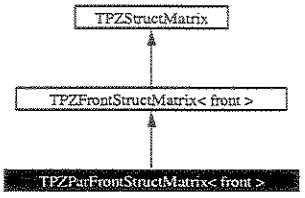
The documentation for this class was generated from the following files:

- **TPZParFrontMatrix.h**
- **TPZParFrontMatrix.c**

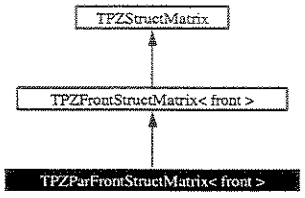
3.9 TPZParFrontStructMatrix Class Template Reference

`#include <TPZParFrontStructMatrix.h>`

Inheritance diagram for TPZParFrontStructMatrix:



Collaboration diagram for TPZParFrontStructMatrix:



Public Methods

- void **SetNumberOfThreads** (int nthreads)
- **TPZStructMatrix** * **Clone** ()
- **TPZParFrontStructMatrix** (TPZCompMesh *mesh)
- TPZMatrix * **CreateAssemble** (TPZFMatrix &rhs)

Static Public Methods

- int **main** ()
- void * **ElementAssemble** (void *t)
- void * **GlobalAssemble** (void *t)

3.9.1 Detailed Description

template<class front> class TPZParFrontStructMatrix< front >

TPZParFrontStructMatrix is derived fron **TPZFrontStructMatrix** (p. 67).

Is a Structural matrix with parallel techniques included It uses **TPZParFrontMatrix** (p. 91) as its FrontalMatrix

Definition at line 39 of file TPZParFrontStructMatrix.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 template<class front> TPZParFrontStructMatrix< front >::TPZParFrontStructMatrix (TPZCompMesh * *mesh*)

Constructor passing as parameter a TPZCompMesh

Parameters:
mesh Mesh to refer to

Definition at line 59 of file TPZParFrontStructMatrix.c.

```
59                                                                  : TPZFrontStructMatrix<
60 {
61     fMaxStackSize = 500;
62     //fNThreads = 1;
63 }
```

3.9.3 Member Function Documentation

3.9.3.1 template<class front> TPZStructMatrix * TPZParFrontStructMatrix< front >::Clone () [virtual]

It clones a **TPZStructMatrix** (p. 125)

Reimplemented from **TPZFrontStructMatrix** (p. 74).

Definition at line 82 of file TPZParFrontStructMatrix.c.

References SetNumberOfThreads.

```
82                                                  {
83     TPZParFrontStructMatrix<front> * mat = new TPZParFrontStructMatrix<front>(fMesh);
84     mat->SetNumberOfThreads(fNThreads);
85     //return new TPZParFrontStructMatrix<front>(fMesh);
86     return (TPZStructMatrix*)mat;
87
88 }
```

3.9.3.2 template<class front> TPZMatrix * TPZParFrontStructMatrix< front >::CreateAssemble (TPZFMatrix & *rhs*) [virtual]

Returns a poniter to TPZMatrix

Parameters:
rhs Load matrix

Reimplemented from **TPZFrontStructMatrix** (p. 75).

Definition at line 343 of file TPZParFrontStructMatrix.c.

References TPZStructMatrix::fMesh, TPZFrontStructMatrix::GetNumElConnected, TPZFrontStructMatrix::OrderElement, TPZFrontMatrix::SetNumElConnected, and TPZParFrontMatrix::WriteFile.

```
343                                                                                                                    {
347     pthread_attr_init(&attr);
348     pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
349     int nthreads;
350     //cout << "Number of Threads " << endl;
351     //cin >> nthreads;
352     //fNThreads = nthreads;
353     cout << "Number of Threads " << fNThreads << endl;
354     nthreads = fNThreads;
355     cout.flush();
356         //int nthreads = fNThreads+1;
357
358     pthread_t *allthreads = new pthread_t[nthreads];
359     int *res = new int[nthreads];
360     int i;
361
362     TPZVec <int> numelconnected(fMesh->NEquations(),0);
363     //TPZFrontMatrix<TPZStackEqnStorage, front> *mat = new TPZFrontMatrix<TPZStackEqnStorage, front>(fMesh->I
364
365     //TPZFrontMatrix<TPZFileEqnStorage, front> *mat = new TPZFrontMatrix<TPZFileEqnStorage, front>(fMesh->NE
366     TPZParFrontMatrix<TPZFileEqnStorage, front> *mat = new TPZParFrontMatrix<TPZFileEqnStorage, front>(fMesh
367     GetNumElConnected(numelconnected);
368     mat->SetNumElConnected(numelconnected);
369
370     fNElements = fMesh->NElements();
371
372     OrderElement();
373     fStiffness = mat;
374     fRhs = &rhs;
375     fCurrentElement = 0;
376     fCurrentAssembled = 0;
377
381     //pthread_create(&allthreads[fNThreads-1],NULL,this->GlobalAssemble, this);
382     // try{
383          res[nthreads-1] = pthread_create(&allthreads[nthreads-1],NULL,this->GlobalAssemble, this);
384          //res[nthreads-1] = pthread_create(&allthreads[nthreads-1],&attr, this->GlobalAssemble, this);
```

```
385           if(!res[nthreads-1]){
386                   cout << "GlobalAssemble Thread created Successfuly "<< allthreads[nthreads-
387                   cout.flush();
388           }else{
389                   cout << "GlobalAssemble Thread Fail "<< allthreads[nthreads-1] << endl;
390                cout.flush();
391          //          exit;
392           }
393           //res[nthreads-2] = pthread_create(&allthreads[nthreads-2],NULL,mat->WriteFile, mat
394           res[nthreads-2] = pthread_create(&allthreads[nthreads-2],&attr,mat->WriteFile, mat)
395           if(!res[nthreads-2]){
396                   cout << "WriteFile Thread created Successfuly "<< allthreads[nthreads-2] <<
397                   cout.flush();
398           }else{
399                   cout << "WriteFile Thread Fail "<< allthreads[nthreads-2] << endl;
400                   cout.flush();
401          //          exit;
402           }
403
404           for(i=0;i<nthreads-2;i++){
405                   res[i] = pthread_create(&allthreads[i],NULL,this->ElementAssemble, this);
406                   if(!res[i]){
407                        cout << "ElementAssemble Thread "<< i+1 <<  " created Successfuly "<< 
408                     cout.flush();
409                   }else{
410                        cout << "ElementAssemble Thread "<< i+1 << " Fail " << allthreads[i] <
411                        cout.flush();
412            //               exit;
413                   }
414           }
415       //}
416 /*      catch(TPZDecompErr){
417           cout << "something wrong inside here" << endl;
418       }*/
419      //for(i=0;i<nthreads-1;i++) res[i] = pthread_create(&allthreads[i],NULL,this->ElementAsse
420
421
422
423 //     Assemble(*mat,rhs);
424 //     for(i=0;i<nthreads-1;i++) pthread_join(allthreads[i], NULL);
425      for(i=0;i<nthreads;i++) pthread_join(allthreads[i], NULL);
426
427      delete allthreads;// fThreadUsed, fDec;
428      delete res;
429
430 //     Assemble(*mat,rhs);
431
432      fStiffness = 0;
433      fRhs = 0;
434      return mat;
435 }
```

3.9.3.3 template<class front> void * TPZParFrontStructMatrix< front >::ElementAssemble (void * t) [static]

It computes element matrices in an independent thread.

It is passed as a parameter to the pthread_create() function.

It is a 'static void *' to be used by pthread_create

Definition at line 93 of file TPZParFrontStructMatrix.c.

References fCurrentElement, fefstack, fekstack, TPZFrontStructMatrix::fElementOrder, felnum, fMaxStackSize, TPZStructMatrix::fMesh, and fNElements.

```
93                                                                    {
94
95   TPZParFrontStructMatrix<front> *parfront = (TPZParFrontStructMatrix<front> *) t;
96
97   TPZElementMatrix *ek,*ef;
98
99
100   TPZAdmChunkVector<TPZCompEl *> &elementvec = parfront->fMesh->ElementVec();
101
102
103
104   while(parfront->fCurrentElement < parfront->fNElements) {
105      ek = new TPZElementMatrix;
106      ek->fMat = new TPZFMatrix(0,0);
107      ek->fConstrMat = new TPZFMatrix(0,0);
108      ef = new TPZElementMatrix;
109      ef->fMat = new TPZFMatrix(0,0);
110      ef->fConstrMat = new TPZFMatrix(0,0);
111
120      //Lock a mutex and get an element number
121
122      pthread_mutex_lock(&mutex_element_assemble);
123
124      //Stack is full and process must wait here!
125      if(parfront->felnum.NElements()==parfront->fMaxStackSize){
126 /*          cout << "    Stack full" << endl;
127           cout << "    Waiting" << endl;
128           cout.flush();*/
129           //cout << "Mutex unlocked on Condwait" << endl;
130           pthread_cond_wait(&stackfull,&mutex_element_assemble);
131           //cout << "Mutex LOCKED leaving Condwait" << endl;
132
133      }
134
135      //cout << "Locking mutex_element_assemble" << endl;
136      //cout.flush();
137      int local_element = parfront->fCurrentElement;
138      if(local_element==parfront->fNElements) return 0;
139 /*          cout << "All element matrices assembled" << endl;
140           return 0;
```

```
141     }
142     */
143     parfront->fCurrentElement++;
144     //Unlock mutex and / or send a broadcast
145     //cout << "Computing Element " << parfront->fCurrentElement << endl;
146     //cout << "Unlocking mutex_element_assemble" << endl;
147     //cout.flush();
148     pthread_mutex_unlock(&mutex_element_assemble);
149
150
151     if(parfront->fElementOrder[local_element] < 0) continue;
152     TPZCompEl *el = elementvec[parfront->fElementOrder[local_element]];
153     if(!el) continue;
154     //         int dim = el->NumNodes();
155
156     //Builds elements stiffness matrix
157     el->CalcStiff(*ek, *ef);
158     //Locks a mutex and adds element contribution to frontmatrix
159     //if mutex is locked go to condwait waiting for an specific condvariable
160     // este mutex deve ser outro mutex -> mutexassemble
161
162     pthread_mutex_lock(&mutex_global_assemble);
163     //cout << "Locking mutex_global_assemble" << endl;
164     //cout << "Pushing variables to the stack" << endl;
165     //cout.flush();
166
167     // colocar ek, ef e o element_local no stack
168     parfront->felnum.Push(local_element);
169     parfront->fekstack.Push(ek);
170     parfront->fefstack.Push(ef);
171
172     // Outro thread procura se stack contem o proximo elemento
173     // qdo nao encontra entra em condwait de acordo com condassemble
174     // isso ocorre num outro processo
175     // Uma vez que uma nova ek foi adicionada ao stack
176     // chame broadcast para acordar o thread que faz assemblagem global
177
178     //cout << "Unlocking mutex_global_assemble" << endl;
179     //cout << "Broadcasting condassemble" << endl;
180     //cout.flush();
181
182 /*     if(!(parfront->fCurrentElement%20)){
183           cout << endl << "Computing " << parfront->fCurrentElement << " on thread " << pthrea
184           cout << " " << (100*parfront->fCurrentElement/parfront->fNElements) << "% Elements c
185     }
186     cout << '*';
187     cout.flush();
188 */
189 //Alterado cond_broadcast para cond_signal
190 //invertendo a sequncia das chamadas
191      pthread_cond_broadcast(&condassemble);
192      pthread_mutex_unlock(&mutex_global_assemble);
193
194     // o thread de assemblagem utiliza mutexassemble
195     // e feito em outro thread     AssembleElement(el, ek, ef, stiffness, rhs);
```

```
196
197
198   }//fim for iel
199
200 }
```

3.9.3.4 template<class front> void * TPZParFrontStructMatrix< front >::GlobalAssemble (void * t) [static]

It assembles element matrices in the global stiffness matrix, it is also executed in an independent thread.

It is passed as a parameter to the pthread_create() function.

It is a 'static void *' to be used by pthread_create

Definition at line 204 of file TPZParFrontStructMatrix.c.

References TPZFrontStructMatrix::AssembleElement, fCurrentAssembled, fCurrentElement, fefstack, fekstack, TPZFrontStructMatrix::fElementOrder, felnum, fMaxStackSize, TPZStructMatrix::fMesh, fNElements, fRhs, and fStiffness.

```
204                                                                    {
205 //void *TPZParFrontStructMatrix<front>::GlobalAssemble(void *t){
206   //cout << "Entering GlobalAssemble" << endl;
207   //cout.flush();
208
209 //  int iel;
210 //  int numel = 0;
211 //  int nelem = fMesh->NElements();
212   TPZParFrontStructMatrix<front> *parfront = (TPZParFrontStructMatrix<front> *) t;
213   //TPZElementMatrix *ek,ef;
214
215 //  ef.fMat = new TPZFMatrix(0,0);
216 //  ef.fConstrMat = new TPZFMatrix(0,0);
217
218   TPZAdmChunkVector<TPZCompEl *> &elementvec = parfront->fMesh->ElementVec();
219
220
221
222   //cout << "Entering FIRST while" << endl;
223   //cout.flush();
224
225   while(parfront->fCurrentAssembled < parfront->fNElements) {
226
227
228     cout << "*";
229     cout.flush();
230     if(!(parfront->fCurrentAssembled%20)){
231       if(parfront->fCurrentElement!=parfront->fNElements){
232         cout << " " << (100*parfront->fCurrentElement/parfront->fNElements) << "% Elements computed "
```

```
233                cout.flush();
234           }else{
235                cout << " " << (100*parfront->fCurrentAssembled/parfront->fNElements) << "% El
236                cout.flush();
237           }
238      }
239
240      //cout << "Executing FIRST while" << endl;
241      //cout.flush();
242 /*     ek = new TPZElementMatrix;
243      ek->fMat = new TPZFMatrix(0,0);
244      ek->fConstrMat = new TPZFMatrix(0,0);
245 */
251      //Lock a mutex and get an element number
252      int local_element = parfront->fCurrentAssembled;
253      parfront->fCurrentAssembled++;
254
255      if(parfront->fElementOrder[local_element] < 0) continue;
256      TPZCompEl *el = elementvec[parfront->fElementOrder[local_element]];
257      if(!el) continue;
258      //          int dim = el->NumNodes();
259
260      //Searches for next element
261      int i=0;
262      int aux = -1;
263      TPZElementMatrix *ekaux, *efaux;
264      pthread_mutex_lock(&mutex_global_assemble);
265      //cout << "Global Assemble Locked 'mutex_global_assemble' aux = " << aux << endl;
266      //cout << "Global Assemble local_element = " << local_element << endl;
267      //cout.flush();
268
269      while(aux != local_element){
270           //cout << "Executing SECOND while" << endl;
271           //cout.flush();
272           while(i < parfront->felnum.NElements()) {
273                //cout << "Entering THIRD while" << i << "  " << parfront->felnum[i] << endl;
274                //cout.flush();
275                if(parfront->felnum[i] == local_element){
276                     //Assemble global matrix with local_element contribution
277                     //cout << "Found element " << local_element << endl;
278                     //cout.flush();
279
280                     TPZElementMatrix *ektemp, *eftemp;
281
282                     aux = parfront->felnum[i];
283                     ekaux = parfront->fekstack[i];
284                     efaux = parfront->fefstack[i];
285
286                     int itemp = parfront->felnum.Pop();
287                     //cout << "itemp " << itemp << endl;
288                     //cout.flush();
289                     ektemp = parfront->fekstack.Pop();
290                     eftemp = parfront->fefstack.Pop();
291                     //Restarts threads waiting !!!
292
```

```
293                     /*if(!(parfront->fCurrentAssembled%20)){
294                        if(parfront->felnum.NElements()<parfront->fMaxStackSize){
295                             cout << "Stack unloaded" << endl;
296                             cout.flush();
297                             pthread_cond_broadcast(&stackfull);
298                        }
299
300                     } */
301                     if(parfront->felnum.NElements()<parfront->fMaxStackSize){
302                          /*cout << "Stack unloaded" << endl;
303                          cout.flush();*/
304                          pthread_cond_broadcast(&stackfull);
305                     }
306
307                     if(i < parfront->felnum.NElements()) {
308
309                          parfront->felnum[i] = itemp;
310                          parfront->fekstack[i]=ektemp;
311                          parfront->fefstack[i]=eftemp;
312                     }
313                     break;
314                }
315                i++;
316           }
317           if(aux!=local_element){
318                i=0;
319                //cout << "Going to Cond_Wait on 'condassemble' and 'mutex_global_assemble'" << endl;
320                //cout.flush();
321                pthread_cond_wait(&condassemble, &mutex_global_assemble);
322                //cout << "Waking on 'condassemble' and 'mutex_global_assemble'" << endl;
323                //cout.flush();
324           }
325      }
326                //unlock
327      pthread_mutex_unlock(&mutex_global_assemble);
328      parfront->AssembleElement(el, *ekaux, *efaux, *parfront->fStiffness, *parfront->fRhs);
329
330      delete ekaux;
331      delete efaux;
332 //     return (0);
333      /* cout << endl << "                                        Assembling " << parfront->fCurrentAssembled << "
334      cout << '#';
335      cout.flush(); */
336
337   }//fim for iel
338
339 }
```

3.9.3.5 template<class front> int TPZParFrontStructMatrix< front >::main () [static]

Used only for testing

Reimplemented from **TPZFrontStructMatrix** (p. 67).

Definition at line 440 of file TPZParFrontStructMatrix.c.

References SetNumberOfThreads.

```
440                                                {
441
442     int refine=5;
443     int order=5;
444
445     TPZGeoMesh gmesh;
446     TPZCompMesh cmesh(&gmesh);
447     double coordstore[4][3] = {{0.,0.,0.},{1.,0.,0.},{1.,1.,0.},
448                                {0.,1.,0.}};
449
450     int i,j;
451     TPZVec<REAL> coord(3,0.);
452     for(i=0; i<4; i++) {
453       // initializar as coordenadas do no em um vetor
454       for (j=0; j<3; j++) coord[j] = coordstore[i][j];
455
456       // identificar um espao no vetor onde podemos armazenar
457       // este vetor
458       int nodeindex = gmesh.NodeVec ().AllocateNewElement ();
459
460       // initializar os dados do n
461       gmesh.NodeVec ()[i].Initialize (i,coord,gmesh);
462     }
463     int el;
464     TPZGeoEl *gel;
465     for(el=0; el<1; el++) {
466
467       // initializar os indices dos ns
468       TPZVec<int> indices(4);
469       for(i=0; i<4; i++) indices[i] = i;
470       // O proprio construtor vai inserir o elemento na malha
471       gel = new TPZGeoElQ2d(el,indices,1,gmesh);
472     }
473     gmesh.BuildConnectivity ();
474
475     TPZVec<TPZGeoEl *> subel;
476     //gel->Divide(subel);
477
478
479
480     //cout << "Refinement ";
481     //cin >> refine;
482
483     cout << refine << endl;
484
485     UniformRefine(refine,gmesh);
486
487
488     TPZGeoElBC gelbc(gel,4,-4,gmesh);
```

```
489     TPZMat2dLin *meumat = new TPZMat2dLin(1);
490     TPZFMatrix xk(1,1,1.),xc(1,2,0.),xf(1,1,1.);
491     meumat->SetMaterial (xk,xc,xf);
492     cmesh.InsertMaterialObject(meumat);
493
494     TPZFMatrix val1(1,1,0.),val2(1,1,0.);
495     TPZMaterial *bnd = meumat->CreateBC (-4,0,val1,val2);
496     cmesh.InsertMaterialObject(bnd);
497
498
499
500     cout << "Interpolation order ";
501     // cin >> order;
502     cout << order << endl;
503
504     TPZCompEl::gOrder = order;
505
506     cmesh.AutoBuild();
507     // cmesh.AdjustBoundaryElements();
508     cmesh.InitializeBlock();
509
510     ofstream output("outputPar.dat");
511     // ofstream output2("outputNon.dat");
512     //cmesh.Print(output);
513     TPZAnalysis an(&cmesh,output);
514     // TPZAnalysis an2(&cmesh,output);
515
516     TPZVec<int> numelconnected(cmesh.NEquations(),0);
517     int ic;
518     //cout << "Nmero de Equaes -> " << cmesh.NEquations() << endl;
519     //cout.flush();
520
521     ofstream out("cmeshBlock_out.txt");
522     // cmesh.Print(out);
523     // cmesh.Block().Print("Block",out);
524     for(ic=0; ic<cmesh.ConnectVec().NElements(); ic++) {
525       TPZConnect &cn = cmesh.ConnectVec()[ic];
526       if(cn.HasDependency()) continue;
527       int seqn = cn.SequenceNumber();
528       if(seqn < 0) continue;
529       int firsteq = cmesh.Block().Position(seqn);
530       int lasteq = firsteq+cmesh.Block().Size(seqn);
531       int ind;
532       int temp = cmesh.ConnectVec()[ic].NElConnected();
533       for(ind=firsteq;ind<lasteq;ind++) {
534         numelconnected[ind] = temp;//cmesh.ConnectVec()[ic].NElConnected();
535       }
536     }
537     // //cout << "nequations " << numelconnected.NElements();
538     // for(ic=0;ic<numelconnected.NElements(); ic++) //cout << numelconnected[ic] <<' ';
539     // //cout << endl;
540     // //cout.flush();
541
542     // TPZFrontMatrix<TPZFileEqnStorage, TPZFrontNonSym> *mat = new TPZFrontMatrix<TPZFileEqnStorage, TPZFrc
543     //TPZFrontMatrix<TPZStackEqnStorage, TPZFrontNonSym> *mat = new TPZFrontMatrix<TPZStackEqnStorage, TPZFr
```

```
544     //TPZFrontMatrix<TPZStackEqnStorage> *mat = new TPZFrontMatrix<TPZStackEqnStorage>(cmesh
545
546     TPZParFrontStructMatrix<TPZFrontSym> mat(&cmesh);
547
548     //   TPZFStructMatrix mat2(&cmesh);
549     //  mat->SetNumElConnected(numelconnected);
550     //mat = CreateAssemble();
551     int threads=3;
552     cout << "Number of Threads  ";
553     // cin >> threads;
554     cout << threads << endl;
555
556     mat.SetNumberOfThreads(threads);
557     //mat.SetNumberOfThreads(1);
558
559     an.SetStructuralMatrix(mat);
560     // an2.SetStructuralMatrix(mat2);
561
562     TPZStepSolver sol;
563     // sol.SetDirect(ELU);
564     sol.SetDirect(ECholesky);
565     // TPZStepSolver sol2;
566     // sol2.SetDirect(ECholesky);
567     // sol.SetDirect(ELU);
568
569
570     an.SetSolver(sol);
571     //     an2.SetSolver(sol2);
572     // mat->SetNumElConnected(numelconnected);
573     // mat->SetFileName("longhin.bin");
574     // an.Solver().SetDirect(ELU);
575     // mat->FinishWriting();
576     //  mat->SetFileName('r',"longhin.bin");
577     // //cout << "*********************************************************************************
578     an.Run(output);
579     //an.Print("solution of frontal solver", output);
580     // //cout << "*********************************************************************************
581     // an2.Run(output2);
582     // an2.Print("solution of frontal solver", output2);
583     /*
584       TPZVec<char *> scalnames(1);
585       scalnames[0] = "state";
586
587       TPZVec<char *> vecnames(0);
588
589       TPZDXGraphMesh graph(&cmesh,2,meumat,vecnames,scalnames);
590       ofstream *dxout = new ofstream("poisson.dx");
591       graph.SetOutFile(*dxout);
592       graph.SetResolution(0);
593
594       //an.DefineGraphMesh(2, scalnames, vecnames, plotfile);
595       //an.Print("FEM SOLUTION ",output);
596       //an.PostProcess(1);
597       int istep = 0,numstep=1;
598
```

```
599         graph.DrawMesh(numstep+1);
600         graph.DrawSolution(0,0);
601
602         TPZAnalysis an2(&cmesh,output);
603         TPZFMatrix *full = new TPZFMatrix(cmesh.NEquations(),cmesh.NEquations(),0.);
604         an2.SetMatrix(full);
605         an2.Solver().SetDirect(ELU);
606         an2.Run(output);
607         an2.Print("solution of full matrix", output);
608
609         //       full->Print("full decomposed matrix");
610         */
611       output.flush();
612       cout.flush();
613       return 0;
614
615 }
```

3.9.3.6 template<class front> void TPZParFrontStructMatrix< front >::SetNumberOfThreads (int *nthreads*)

Sets number of threads to be used in frontal process

Parameters:

nthreads Number of threads to be used

Definition at line 65 of file TPZParFrontStructMatrix.c.

Referenced by Clone, and main.

```
66 {
67       if(nthreads > 2)
68       {
69            fNThreads = nthreads;
70            cout << "Number of Threads set to " << fNThreads << endl;
71            cout.flush();
72       }else{
73            cout << "At least '3' threads are necessary !" << endl;
74            cout << "Setting Number of Threads to 3 !!" << endl;
75            cout.flush();
76            fNThreads = 3;
77       }
78 }
```

The documentation for this class was generated from the following files:

- **TPZParFrontStructMatrix.h**
- **TPZParFrontStructMatrix.c**

3.10 TPZStackEqnStorage Class Reference

`#include <TPZStackEqnStorage.h>`

Public Methods

- void **ReOpen** ()
- void **FinishWriting** ()
- **TPZStackEqnStorage** (char option, const char *name)
- **~TPZStackEqnStorage** ()
- **TPZStackEqnStorage** ()
- void **AddEqnArray** (**TPZEqnArray** *EqnArray)
- void **Print** (const char *name, ostream &out)
- void **Reset** ()
- void **Backward** (TPZFMatrix &f, DecomposeType dec) const
- void **Forward** (TPZFMatrix &f, DecomposeType dec) const
- void **OpenGeneric** (char option, const char *name)
- void **ReadBlockPositions** ()
- char * **GetStorage** ()

Static Public Methods

- void **main** ()

3.10.1 Detailed Description

Responsible for storing arrays of equations (mostly in a decomposed form) It has methods for operating over a set of equations The arrays of equations are in the form of a Stack of EqnArrays

Definition at line 13 of file TPZStackEqnStorage.h.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 TPZStackEqnStorage::TPZStackEqnStorage (char *option*, const char * *name*)

Only to make both possible templates similar in terms of methods and constructors

Definition at line 59 of file TPZStackEqnStorage.c.

```
60 {
61
62 }
```

3.10.2.2 TPZStackEqnStorage::~TPZStackEqnStorage ()

Simple Destructor

Definition at line 51 of file TPZStackEqnStorage.c.

```
52 {
53 }
```

3.10.2.3 TPZStackEqnStorage::TPZStackEqnStorage ()

Simple Constructor

Definition at line 47 of file TPZStackEqnStorage.c.

```
48 {
49 }
```

3.10.3 Member Function Documentation

3.10.3.1 void TPZStackEqnStorage::AddEqnArray (TPZEqnArray * *EqnArray*)

Adds an EqnArray to EqnStack object

Parameters:
EqnArray Pointer to EqnArray to be added to the Stack

Parameters:
EqnArray Pointer to EqnArray to be added to the Stack

Definition at line 42 of file TPZStackEqnStorage.c.

```
43 {
44         fEqnStack.Push(*EqnArray);
45 }
```

3.10.3.2 void TPZStackEqnStorage::Backward (TPZFMatrix & *f*, DecomposeType *dec*) const

Executes a Backward substitution Stack object

Parameters:

f Matrix to apply Backward substitution on

dec Decomposition type of f, depends on what decomposition method was used to decompose f

Definition at line 23 of file TPZStackEqnStorage.c.

```
24 {
25         int i, stack_size;
26         stack_size=fEqnStack.NElements();
27         for(i=stack_size-1;i>=0;i--){
28                 fEqnStack[i].EqnBackward(f, dec);
29         }
30
31 }
```

3.10.3.3 void TPZStackEqnStorage::FinishWriting ()

It closes the opened binary file.

Definition at line 66 of file TPZStackEqnStorage.c.

```
66 {}
```

3.10.3.4 void TPZStackEqnStorage::Forward (TPZFMatrix & *f*, DecomposeType *dec*) const

Executes a Forward substitution Stack object

Parameters:

f Matrix to apply Forward substitution on

dec Decomposition type of f Depends on what decomposition method was used to decompose f

Definition at line 32 of file TPZStackEqnStorage.c.

```
33 {
34         int i, stack_size;
35         stack_size=fEqnStack.NElements();
```

```
36          for(i=0;i<stack_size;i++){
37                  fEqnStack[i].EqnForward(f, dec);
38          }
39
40 }
```

3.10.3.5 char * TPZStackEqnStorage::GetStorage ()

Name of Storage

Definition at line 67 of file TPZStackEqnStorage.c.

```
67 {return "Stack Storage";}
```

3.10.3.6 void TPZStackEqnStorage::main () [static]

Static main for testing

Definition at line 55 of file TPZStackEqnStorage.c.

```
56 {
57 }
```

3.10.3.7 void TPZStackEqnStorage::OpenGeneric (char *option*, const char * *name*)

Only to make both possible templates similar in terms of methods and constructors

Definition at line 64 of file TPZStackEqnStorage.c.

```
64 {}
```

3.10.3.8 void TPZStackEqnStorage::Print (const char * *name*, ostream & *out*)

It prints TPZEqnStorage data.

Parameters:

name file title to print to

out object type file

Definition at line 13 of file TPZStackEqnStorage.c.

```
13                                                              {
14     int i, loop_limit;
15         loop_limit=fEqnStack.NElements();
16     out <<  "Number of entries on EqnStack  "<< fEqnStack.NElements() << endl;
17     for(i=0;i<loop_limit;i++) fEqnStack[i].Print(name, out);
18 }
```

3.10.3.9 void TPZStackEqnStorage::ReadBlockPositions ()

Only to make both possible templates similar in terms of methods and constructors

Definition at line 65 of file TPZStackEqnStorage.c.

```
65 {}
```

3.10.3.10 void TPZStackEqnStorage::Reset ()

Resets data structure

Definition at line 19 of file TPZStackEqnStorage.c.

```
20 {
21         fEqnStack.Resize(0);
22 }
```

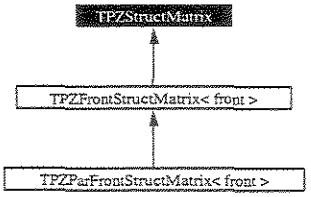
The documentation for this class was generated from the following files:

- **TPZStackEqnStorage.h**
- **TPZStackEqnStorage.c**

3.11 TPZStructMatrix Class Reference

`#include <pzstrmatrix.h>`

Inheritance diagram for TPZStructMatrix:



Public Methods

- **TPZStructMatrix** (TPZCompMesh *)
- virtual ~**TPZStructMatrix** ()
- virtual TPZMatrix * **Create** ()
- virtual TPZMatrix * **CreateAssemble** (TPZFMatrix &rhs)
- virtual TPZStructMatrix * **Clone** ()
- virtual void **Assemble** (TPZMatrix &mat, TPZFMatrix &rhs)

Protected Attributes

- TPZCompMesh * **fMesh**

3.11.1 Detailed Description

Is responsible for a interface among Matrix and Finite Element classes.

Definition at line 14 of file pzstrmatrix.h.

3.11.2 Member Data Documentation

3.11.2.1 TPZCompMesh* TPZStructMatrix::fMesh [protected]

@supplierCardinality 1

Definition at line 37 of file pzstrmatrix.h.

Referenced by TPZFrontStructMatrix::Assemble, TPZFrontStructMatrix::AssembleElement, TPZFrontStructMatrix::AssembleNew, TPZFrontStructMatrix::Create, TPZParFrontStructMatrix::CreateAssemble, TPZFrontStructMatrix::CreateAssemble, TPZParFrontStructMatrix::ElementAssemble, TPZFrontStructMatrix::GetNumElConnected, and TPZParFrontStructMatrix::GlobalAssemble.

The documentation for this class was generated from the following files:

- pzstrmatrix.h
- pzstrmatrix.c