MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
DEPARTAMENTO DE ENGENHARIA MECÂNICA

APERFEIÇOAMENTO DE UM PROGRAMA DE OTIMIZAÇÃO TOPOLÓGICA DE ESTRUTURAS CONTÍNUAS SUJEITAS A RESTRIÇÃO DE TENSÃO

por

Daniel de Bortoli

Monografia apresentada ao Departamento de Engenharia Mecânica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para obtenção do diploma de Engenheiro Mecânico.

Porto Alegre, dezembro de 2011

Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Engenharia Mecânica

APERFEIÇOAMENTO DE UM PROGRAMA DE OTIMIZAÇÃO TOPOLÓGICA DE ESTRUTURAS CONTÍNUAS SUJEITAS A RESTRIÇÃO DE TENSÃO

por

Daniel de Bortoli

ESTA MONOGRAFIA FOI JULGADA ADEQUADA COMO PARTE DOS REQUISITOS PARA A OBTENÇÃO DO TÍTULO DE
ENGENHEIRO MECÂNICO
APROVADA EM SUA FORMA FINAL PELA BANCA EXAMINADORA DO DEPARTAMENTO DE ENGENHARIA MECÂNICA

Prof. Arnaldo Ruben Gonzalez
Coordenador do Curso de Engenharia Mecânica

Área de Concentração: **Mecânica dos Sólidos**

Orientador: Prof. Jun Sérgio Ono Fonseca

Comissão de Avaliação:

Prof. Letícia Fleck Fadel Miguel

Prof. Herbert Martins Gomes

Prof. Juan Pablo Raggio Quintas

Porto Alegre, 05 de dezembro de 2011

DE BORTOLI, D. **Aperfeiçoamento de um Programa de Otimização Topológica de Estruturas Contínuas Sujeitas a Restrição de Tensão**. 2011. 25 p. Monografia (Trabalho de Conclusão do Curso em Engenharia Mecânica) – Departamento de Engenharia Mecânica, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011.

RESUMO

Problemas de otimização topológica são, por natureza, de grande escala, exigindo considerável esforço computacional para sua solução. Dessa maneira, é fundamental desenvolver algoritmos eficientes que resolvam em tempo hábil problemas de interesse prático. Esse trabalho trata do desenvolvimento continuado de uma ferramenta numérica de otimização topológica com restrição de tensão. Particular atenção é dada à solução de sistemas lineares, etapa crítica em termos de tempo de computação durante a solução do problema. Uma biblioteca que realiza a solução paralela de sistemas lineares esparsos utilizando o protocolo de comunicação MPI foi incorporada ao código, aprimorando sua performance. Além disso, certas rotinas do algoritmo foram paralelizadas com o uso de OpenMP. Resultados que permitem aferir o desempenho do novo código são apresentados.

PALAVRAS-CHAVE: otimização topológica, restrição de tensão, computação em paralelo, OpenMP, MPI.

DE BORTOLI, D. **Improving a Computer Code for Topological Optimization with Stress Constraints**. 2011. 25 p. Monografia (Trabalho de Conclusão do Curso em Engenharia Mecânica) – Departamento de Engenharia Mecânica, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011.

ABSTRACT

Topological optimization formulations lead to large scale problems, requiring extensive computational effort to be solved. Developing performing algorithms is of capital importance in order to be able to solve practical problems. This work deals with the improvement of a code for topological optimization with stress constraints. In particular, a library which solves sparse linear systems in parallel using MPI for communication is used. Furthermore, certain routines are made parallel using OpenMP directives. Some performance results show the speedup obtained in multi-core processors.

KEYWORDS: topological optimization, stress constraints, parallel computing, OpenMP, MPI.

ÍNDICE

Pág.

LISTA DE FIGURAS

Pag.

LISTA DE TABELAS

Pag.

1. INTRODUÇÃO

Otimização pode ser definida como a busca do melhor resultado para uma dada operação, satisfazendo certas restrições [Haftka e Gürdal, 1992]. Em particular, otimização estrutural concerne a minimização/maximização de uma característica de interesse de uma estrutura: massa, frequências naturais, flexibilidade. Os métodos de otimização permitem calcular um conjunto de parâmetros da estrutura, como as áreas de seção transversal, em vigas, considerados ótimos de acordo com certo critério e que respeitem as restrições impostas (tensão admissível no material, deslocamento máximo em um ponto...). Dessa forma, o método pode ser considerado como um aprimoramento sistemático de projetos, permitindo satisfazer os requisitos com o mínimo de recursos. No contexto de um mercado cada vez mais competitivo e exigente, essa pode ser uma ferramenta valiosa nas mãos de projetistas mecânicos.

Formulações de otimização estrutural são por natureza matematicamente complexas e sua solução numérica exige algoritmos eficientes e esforço computacional considerável. Kuckoski, 2009, aplicou a formulação apresentada por Guilherme, 2006, desenvolvendo um código que permite a resolução de problemas de minimização de massa com restrição de tensão. Embora o código resolva corretamente o problema proposto, algumas etapas dos cálculos acarretam grande tempo computacional.

Para contornar esse problema, Kuckoski estabeleceu uma interface entre seu código e o pacote comercial Ansys, tirando proveito de seus algoritmos de solução (*solvers*) de sistemas lineares. Essa abordagem limita o controle sobre os métodos e parâmetros de solução e cria uma dependência com código disponível apenas comercialmente, além de implicar em tempo computacional desperdiçado na troca de dados entre os dois aplicativos.

2. OBJETIVOS

O principal objetivo desse trabalho é acelerar a execução do código existente, aprimorando alguns pontos de seu algoritmo de cálculo. Uma solução mais eficiente permite tratar problemas mais relevantes de um ponto de vista prático, com geometrias complexas e discretizações em elementos finitos refinadas. O novo código deve ter capacidade de realizar processamento em paralelo, de modo a tirar proveito de processadores com vários núcleos (*multi-core*) ou então ser utilizado em um agrupamento de máquinas (*cluster*).

A formulação do problema de otimização topológica, apresentada no Capítulo 3, consiste na minimização da massa de uma estrutura utilizando uma restrição de tensão global. No Capítulo 4, são apresentados alguns conceitos de computação em paralelo que serão úteis para descrição das alterações realizadas no algoritmo de solução (Capítulo 5). Finalmente, resultados que ilustram o desempenho do novo código são relatados no Capítulo 6.

3. OTIMIZAÇÃO TOPOLÓGICA

Em um problema de otimização, as funções que se deseja extremizar são denominadas *funções objetivo*, e os parâmetros que podem ser alterados nessa busca são as *variáveis de projeto*. Em geral, as variáveis de projeto não podem assumir quaisquer valores, devendo satisfazer algumas *restrições*. Em um problema de otimização estrutural, podem existir tanto restrições de igualdade, como as equações de equilíbrio, quanto de desigualdade, como a restrição de tensão, que impõe como limite a tensão admissível no material da estrutura. Um conjunto de variáveis de projeto que respeite todas as restrições impostas é denominado *projeto viável*.

A otimização estrutural pode ser dividida em três categorias principais: otimização paramétrica, de forma e topológica. Na otimização paramétrica (ou dimensional), as variáveis de projeto estão diretamente ligadas a uma característica geométrica da estrutura, como a distribuição de espessuras em um modelo de placas. Nesse caso, o domínio de otimização da

estrutura é conhecido *a priori*. A otimização de forma, por sua vez, busca encontrar a forma ótima do contorno de uma estrutura [Haftka e Gürdal, 1992].

De uma maneira bastante geral, a otimização topológica de estruturas contínuas consiste em determinar se em cada ponto do domínio admissível da estrutura há material ou não [Bendsøe, 1995]. Desse modo, a topologia da estrutura não é conhecida a princípio, podendo criar-se cavidades nas regiões em que não seja necessário haver material.

O problema que se pretende resolver pode ser resumido da seguinte maneira: em um domínio admissível inicialmente preenchido com material (o *domínio de projeto*), condições de contorno são aplicadas (carregamentos, vinculações) e deseja-se saber a distribuição espacial ótima de material, no sentido de minimizar a massa da estrutura sem entretanto violar a tensão admissível do material. A seguir são apresentadas as principais hipóteses de trabalho da formulação utilizada, sem detalhamento extensivo. Algumas das equações associadas a essa formulação podem ser encontradas no Anexo II; mais informações e justificativas para a utilização das técnicas apresentadas podem ser encontradas nos trabalhos de Guilherme, 2006, e Kuckoski, 2009.

3.1. Variáveis de projeto

Em princípio, a formulação de um problema de otimização topológica é de natureza inerentemente discreta: em cada ponto do domínio (ou no caso de uma discretização em elementos finitos, em cada elemento), existe material ou não. Essa formulação discreta é conhecida como problema 0-1 e sabe-se que este é um problema mal posto [Bendsøe, 1995]. Uma maneira de contornar essa limitação é introduzindo uma variável artificial densidade, variando entre 1 (material original) e 0 (vazio). Valores intermediários de densidade podem ser relacionados a diferentes microestruturas de compósitos formados a partir do material original contendo poros a nível microscópico; essa abordagem é denominada *método da homogeneização*. Nesse método, a formação de regiões sem material no domínio admissível da estrutura é realizada por intermédio da variável contínua densidade.

Os valores de densidade são diretamente relacionados às propriedades mecânicas do material compósito. Nesse trabalho, adota-se a hipótese de que o material utilizado na estrutura possua comportamento elástico linear isotrópico. Além disso, considera-se uma relação de linearidade entre as propriedades mecânicas do material e sua densidade.

3.2. Função objetivo

Como já mencionado anteriormente, o objetivo da otimização realizada nesse trabalho é a minimização da massa da estrutura, a qual está diretamente relacionada à distribuição espacial de densidades no seu domínio admissível e constitui, dessa maneira, a função objetivo do problema.

Embora teoricamente possuam sentido físico e possam ser interpretadas em termos da microestrutura do material, densidades intermediárias não são desejadas, tendo em vista as dificuldades técnicas e econômicas que a fabricação de uma peça contendo esse tipo de gradação material apresentaria. Considerando isso, a função objetivo é corrigida com alguns coeficientes de penalização que desfavorecem o aparecimento de densidades intermediárias.

3.3. Programação matemática

A área da matemática que trata da extremização de uma função sujeita a um conjunto de restrições no espaço de projeto (conjunto de todos os valores que as variáveis de projeto podem assumir) é denominada programação matemática. Em particular, quando as funções de restrição e a função objetivo são lineares, fala-se em programação linear. Como grande parte dos problemas de interesse prático, em geral formulações de otimização estrutural envolvem funções não-lineares.

Nesse caso, uma técnica comum é recorrer à *programação linear sequencial*, que consiste em linearizar as funções objetivo e restrições e resolver diversos problemas de programação linear em sequência a partir de uma estimativa inicial. A solução obtida é utilizada como novo ponto de partida em torno do qual as funções objetivo e restrições são novamente linearizadas [Haftka e Gürdal, 1992]. O processo é repetido até que se obtenha convergência segundo um critério estabelecido.

Para que a solução do problema de programação linear sequencial seja consistente com a do problema associado de programação não-linear, faz-se uso dos *limites móveis*. Como as funções objetivo e de restrição são expandidas em série de Taylor e truncadas nos termos lineares, a validade das aproximações é restrita à proximidade do ponto de expansão. Os limites móveis garantem que essa exigência seja satisfeita. A escolha adequada dos limites móveis é fator determinante para a convergência da solução; limites muito estreitos fazem o método avançar muito lentamente em direção ao ponto de ótimo. Nesse trabalho, utiliza-se uma heurística de atualização local dos limites móveis baseada no histórico da densidade de um elemento finito: se ela variou no mesmo sentido (aumento ou diminuição) por duas iterações seguidas, os limites móveis são ampliados. Caso contrário, são reduzidos.

3.4. Restrição de tensão global

A restrição de tensão admissível no material é de natureza local, uma vez que em nenhum ponto da estrutura esse limite pode ser ultrapassado. Como critério de tensão é utilizado a tensão equivalente de von Mises, calculada para cada elemento da malha de elementos finitos. Isso significa que para cada elemento existe, a princípio, uma função restrição, tornando o problema extremamente intensivo do ponto de vista computacional, pois as derivadas de cada função de restrição em relação às variáveis de projeto são necessárias na etapa de otimização das densidades (análise de sensibilidades). Na discretização por elementos finitos, toma-se a densidade como sendo constante ao longo de um elemento, o que significa que há tantas variáveis de projeto quanto elementos finitos na malha.

Dessa maneira, uma medida global de tensão é utilizada para limitar a restrição a uma única função e simplificar a análise de sensibilidades. A grande dificuldade consiste em encontrar uma função global que seja robusta e geral, atendendo satisfatoriamente a diversas espécies de carregamento [Guilherme, 2006].

3.5. Cálculo das sensibilidades

Com a linearização das funções objetivo e restrições surge a necessidade de calcular suas derivadas em relação às variáveis de projeto, numa etapa denominada análise de sensibilidades.

A sensibilidade da função objetivo é calculada analiticamente, devido à simplicidade do cálculo. Por sua vez, a sensibilidade da restrição global de tensão é calculada utilizando o método adjunto, que envolve a solução de um sistema linear adicional para cálculo dos pseudo-deslocamentos a partir das pseudo-forças [Haftka e Gürdal, 1992].

3.6. Relaxação-ε

A relaxação-ε é uma técnica que busca evitar o problema do ótimo singular que surge na otimização com restrição de tensão. Esse problema ocorre quando a solução se aproxima do ponto ótimo e a densidade de certos elementos tende a zero ao passo que a tensão nesses elementos converge para um valor finito não-nulo. O fenômeno causa oscilações na convergência do processo iterativo de otimização e pode impedir que ele atinja um valor ótimo.

Para evitar essa dificuldade, utiliza-se a técnica da relaxação-ε, que consiste em resolver uma versão perturbada do problema inicial, introduzindo um parâmetro de relaxação na restrição de tensão dependente da densidade [Guilherme, 2006]. Esse parâmetro é

reduzido sucessivamente, fazendo com que a solução do problema perturbado tenda à solução do problema original.

3.7. Filtragem das densidades

Outro problema comum na otimização topológica com restrição de tensão é a obtenção da chamada instabilidade de tabuleiro de xadrez, onde elementos contíguos apresentam um padrão alternado de altas e baixas densidades que lembra as casas de um tabuleiro de xadrez. Sabe-se que certas discretizações por elementos finitos, especialmente as de baixa ordem, apresentam esse problema [Bendsøe, 1995]. O uso de elementos finitos de ordem mais elevada minimiza a ocorrência dessa forma de instabilidade, porém eleva sensivelmente o custo computacional envolvido na solução do problema.

Uma alternativa interessante é a aplicação de um filtro sobre as variáveis de projeto, de maneira que a densidade de um elemento esteja relacionada às densidades de seus vizinhos. Além de evitar os problemas de instabilidade de tabuleiro de xadrez, essa abordagem minimiza problemas de não unicidade de soluções associados à dependência de malha, ou seja, o fato de diferentes malhas de elementos finitos levarem a soluções diferentes [Kuckoski, 2009].

4. COMPUTAÇÃO PARALELA

Tradicionalmente, os programas de computador foram feitos para serem executados em uma máquina contendo um único processador; nessa situação, uma tarefa é decomposta em uma série de instruções a serem seguidas sequencialmente, com uma única instrução sendo executada em um dado instante de tempo, como mostrado na Figura 4.1(a). Nesse contexto, um programa paralelo é basicamente aquele que foi feito para ser executado em múltiplos processadores ao mesmo tempo. Para que sua execução seja paralela, a tarefa deve ser decomposta em partes que possam ser executadas de maneira concorrente. Por sua vez, cada parte é decomposta em instruções a serem executadas sequencialmente e enviada a um processador distinto. Como ilustra a Figura 4.1(b), a cada instante de tempo há múltiplas instruções sendo executadas simultaneamente.

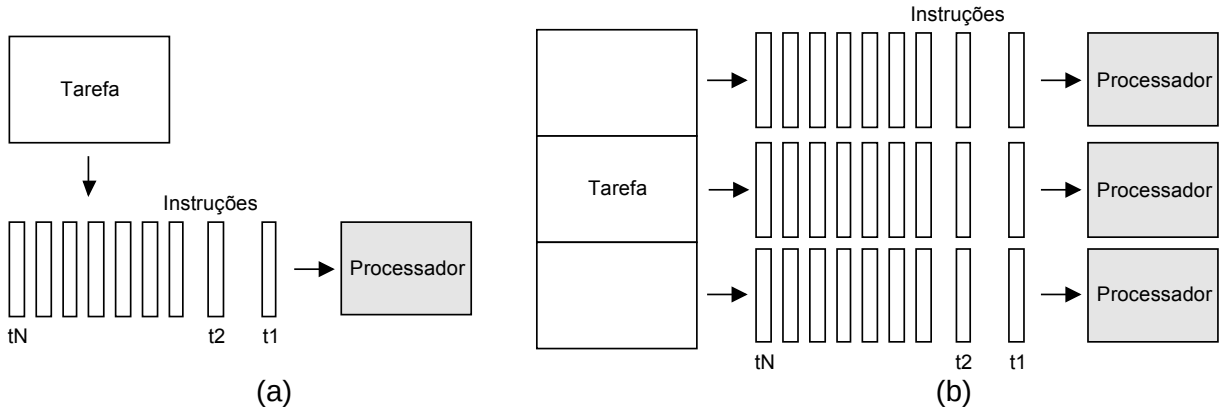

Figura 4.1 – (a) Processamento sequencial. (b) Processamento paralelo.

O interesse em utilizar a computação em paralelo é evidenciado quando se leva em consideração os limites para o desempenho de um único processador. A velocidade de um processador está diretamente atrelada à velocidade com que os dados são transmitidos nos seus circuitos, o que possui limites físicos teóricos como a velocidade da luz e a velocidade de transmissão do cobre. Aumentar a velocidade do processador implica aproximar os seus componentes e aumentar seu número, o que torna, a partir de certo ponto, a criação de processadores mais rápidos economicamente ou mesmo tecnicamente inviável. Além disso,

questões problemáticas como consumo elevado de energia e alta dissipação de calor tornam-se ainda mais críticas [Barney, 2011].

Dessa forma, nas últimas décadas, observa-se uma tendência bem estabelecida a utilizar computação em paralelo para atingir altas performances [van der Steen e Dongarra, 2003]. Mesmo nos computadores domésticos, as arquiteturas *multi-core* já estão se proliferando e sua utilização tende a se consolidar como o padrão.

Existem muitas arquiteturas de máquinas (*hardware*) projetadas para realizar computação em paralelo, assim como diversos paradigmas de programação capazes de gerenciar a execução paralela de diversos processos concorrentes. A seguir são apresentadas as principais classificações pertinentes à computação em paralelo, tanto em termos de *hardware* quanto de código (*software*).

4.1. Classificação de computadores paralelos

Uma das primeiras classificações computadores de alta performance foi proposta por Flynn em 1966, prevendo quatro categorias distintas [Foster, 1995]:

- SISD (*Single Instruction, Single Data*): este é o computador sequencial tradicional, executando apenas uma instrução por ciclo e utilizando um único fluxo de dados para tanto. Foi o primeiro tipo a ser utilizado, e ainda hoje é comum.
- SIMD (*Single Instruction, Multiple Data*): este tipo de computador paralelo envolve múltiplos processadores executando as mesmas instruções a cada ciclo, cada qual operando num fluxo de dados diferente. É adequado para problemas altamente regulares, como o processamento de imagem, de modo que é atualmente empregado por placas de processadores gráficos (GPUs).
- MISD (*Multiple Instruction, Single Data*): essa classe emprega múltiplos processadores operando independentemente em um único fluxo de dados; na prática, sua utilidade é limitada e poucos computadores pertenceram a essa categoria.
- MIMD (*Multiple Instruction, Multiple Data*): essa categoria mais geral permite que cada processador trabalhe com um fluxo de dados diferente, de modo independente. A maioria dos computadores paralelos atuais se enquadra aqui.

Um critério adicional de classificação, de importância fundamental, diz respeito à arquitetura de compartilhamento de memória empregada pela máquina.

4.1.1. Arquiteturas de memória compartilhada

Essa classe, representada na Figura 4.2(a), é definida pela capacidade de todos os processadores acessarem uma memória global, compartilhando os mesmo recursos. Desse modo, alterações em um endereço de memória são visíveis a todos os processadores. Podem ainda ser divididas em duas classes principais:

- UMA (*Uniform Memory Access*): os processadores são idênticos e possuem tempo de acesso uniforme a toda a memória. Atualmente os representantes mais comuns dessa categoria são as máquinas SMP (*Symmetric MultiProcessor*), onde todos os processadores compartilham um mesmo espaço de endereços de memória e acesso aos mesmos recursos.
- NUMA (*Non-Uniform Memory Access*): nesse caso, os tempos de acesso à memória não são uniformes, podendo variar de processador a processador. Um exemplo comum consiste em ligar dois ou mais SMPs, de modo que um possa acessar a memória do outro; como o acesso à memória de outros processadores é feito através do barramento, ele é mais lento do que entre o processador e sua própria memória.

Em geral, o fato de possuírem uma memória global facilita o trabalho do programador, que não tem de gerar a comunicação de dados entre os processadores; entretanto, o sincronismo entre os processos deve ser controlado para que o acesso à memória não gere

problemas (por exemplo, vários processadores tentando atualizar o mesmo endereço de memória simultaneamente). A principal desvantagem desse modelo é a falta de proporcionalidade entre memória e processadores, o que implica um aumento geométrico no tráfego de dados entre estes com o aumento do número de processadores [Barney, 2011]. Outras desvantagens são as dificuldades técnica e econômica de se projetar e produzir máquinas com memória compartilhada utilizando um número elevado de processadores.

4.1.2. Arquiteturas de memória distribuída

A característica comum dos processadores dessa categoria, ilustrada na Figura 4.2(b), é a necessidade de uma rede de comunicação interligando as memórias locais de cada processador. Não há um conceito de memória global e cada processador opera em sua memória local, sendo completamente independente. A arquitetura de *hardware* prevê como e quando os dados são comunicados caso um processador necessite acessar dados da memória de outro.

Como cada processador possui sua própria memória, há proporcionalidade entre a memória disponível e o número de processadores utilizados, contornando assim uma das maiores limitações da arquitetura de memória compartilhada.

Também é possível combinar as arquiteturas de memória distribuída e compartilhada, utilizando, por exemplo, uma rede com vários SMPs. Essa é a arquitetura híbrida, utilizada pelos mais potentes supercomputadores da atualidade [van der Steen e Dongarra, 2003].

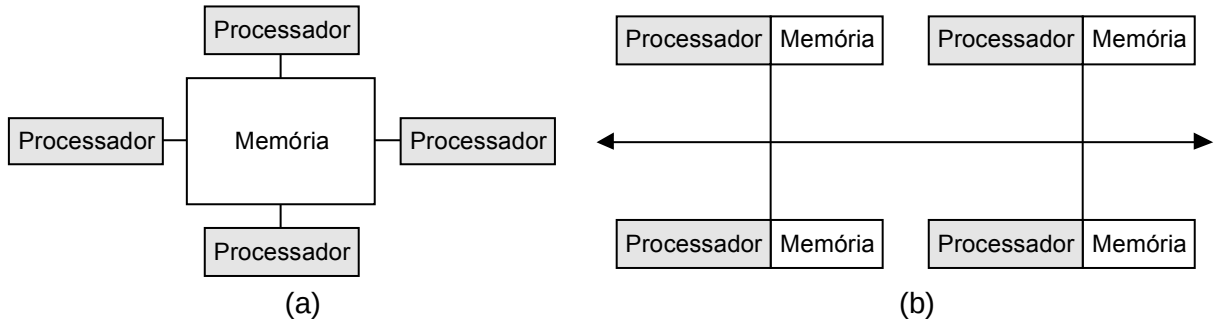

Figura 4.2 – (a) Arquitetura de memória compartilhada. (b) Arquitetura de memória distribuída.

4.2. Paradigmas de programação em paralelo

Independentemente da arquitetura de *hardware* utilizada, diversos paradigmas de programação em paralelo podem ser implementados, utilizando tanto memória compartilhada como memória distribuída. Dentre outras possibilidades, os paradigmas de maior interesse para esse trabalho são os modelos de *threads* e de *message passing*.

4.2.1. Modelo de linhas de execução (*threads*)

Trata-se de um paradigma de memória compartilhada, onde um único processo pode criar múltiplas linhas de execução (*threads*) concorrentes. Cada *thread* tem acesso à memória global do processo que o criou, compartilhando seus recursos. Um *thread* pode ser visto como uma sub-rotina de um programa que é executada de maneira concorrente com outros *threads*. A comunicação entre os *threads* ocorre por meio da memória global e requer controle de sincronização, para assegurar, por exemplo, que mais de um *thread* não atualize o mesmo endereço simultaneamente. O processo provê os recursos necessários para a inicialização e término dos *threads* até que ele tenha sido concluído.

Existem duas padronizações independentes desse tipo de programação: os POSIX Threads (*Pthreads*), desenvolvidos em 1995 para C, e o OpenMP (*Open MultiProcessing*),

lançado inicialmente em 1997 para Fortran90 e em 1998 para C/C++. Os *Pthreads* requerem a paralelização completa do código, que deve ser realizada de maneira bastante explícita pelo programador. Por sua vez, o OpenMP é um padrão mais acessível, de modo que sua utilização será priorizada nesse trabalho.

O OpenMP é baseado em algumas diretivas de compilador que gerenciam boa parte do paralelismo, o que faz com que o programador não precise se concentrar nos detalhes do gerenciamento dos *threads*. O OpenMP prevê uma série de diretivas que simplificam a realização de tarefas comuns como a distribuição das iterações de um laço entre os *threads*, por exemplo. Outra característica interessante do padrão é o fato de ele permitir a paralelização parcial do código, gerando o chamado *forking* (bifurcação), ilustrado na Figura 4.3: o processo inicialmente sequencial pode ser dividido em múltiplos *threads* nas partes computacionalmente intensivas, e ao fim dessas tarefas, retornar à execução sequencial.

Informações mais detalhadas sobre as diversas funcionalidades do padrão podem ser encontradas em http://openmp.org/wp/.

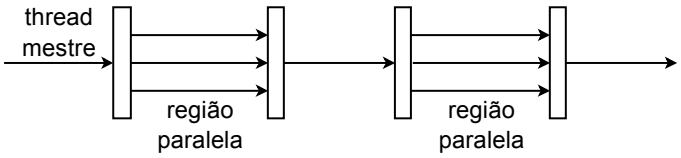

Figura 4.3 – Paralelização parcial de um código com OpenMP (*forking*).

4.2.2. Modelo de troca de mensagens (*message passing*)

Trata-se de um paradigma de programação de memória distribuída onde uma série de processos mantém sua memória local durante a execução. Essas memórias locais podem estar localizadas em uma mesma máquina ou em máquinas distintas. Os processos trocam dados por meio do envio e recepção de mensagens. Normalmente, isso é feito com sub-rotinas que o programador deve chamar no código fonte para gerenciar o paralelismo.

Atualmente, existe um padrão de *message passing* cuja utilização é praticamente unânime: o MPI (*Message Passing Interface*), lançado em 1994 e atualmente em sua segunda versão. Contrariamente ao OpenMP, com o MPI o programador deve gerar o paralelismo entre os processos de maneira bastante explícita, utilizando para tanto funções definidas pelo padrão, que contemplam basicamente o envio e a recepção de mensagens para troca de dados entre os processos. Informações adicionais sobre o padrão MPI bem como suas diversas implementações, existentes para praticamente todas as plataformas de computação em paralelo, podem ser encontradas em http://www.mcs.anl.gov/research/projects/mpi/.

4.2.3. Modelo híbrido

Além dos dois paradigmas apresentados, também merece menção o modelo híbrido, que combina mais de um paradigma. Uma opção bastante utilizada é a combinação do MPI com o modelo de *threads*, implementado com o OpenMP. Desse modo, os *threads* realizam operações na memória compartilhada de um SMP, por exemplo, e comunicam-se com outros processadores na rede utilizando o MPI, uma combinação adequada para *clusters* de processadores *multi-core*.

É importante constatar que os paradigmas de programação são uma abstração independente do *hardware*, não estando a ele vinculados [Barney, 2011]. Por exemplo, seria perfeitamente possível utilizar um paradigma de programação por troca de mensagens em um processador com memória compartilhada, embora fosse mais natural utilizar *threads* nesse caso.

5. ALGORITMO UTILIZADO

5.1. Principais modificações efetuadas

O desenvolvimento de algoritmos paralelos envolve uma série de considerações complexas que não serão abordadas nesse trabalho, como a divisão, a comunicação e o gerenciamento das dependências entre os dados dos diferentes processos, o sincronismo e o balanceamento da carga atribuída a cada processador.

Ao invés de desenvolver um algoritmo completamente paralelo, optou-se por manter o código sequencial existente, desenvolvido em C, e paralelizar somente as regiões críticas do algoritmo, ou seja, as regiões computacionalmente intensivas que podem receber um incremento de velocidade razoável com a paralelização. Uma inspeção dos tempos de execução do código original revela que a região mais crítica do algoritmo é o cálculo dos deslocamentos e dos pseudo-deslocamentos, que envolvem a resolução de sistemas lineares.

Em seu trabalho, Kuckoski, 2009, resolveu esses sistemas lineares com uma variante do método da eliminação gaussiana, a decomposição de Cholesky, empregando o armazenamento de matrizes de perfil variável (método *skyline*). Esse método de armazenamento é propício para matrizes que podem ser armazenadas em banda, o que se aplica à matriz de rigidez da estrutura [Bathe, 1996]. Entretanto, sua eficiência é extremamente dependente da numeração nodal empregada, como será visto adiante, o que não foi considerado no trabalho anterior e explica a ineficiência do método.

Uma constatação importante, justificada posteriormente, é que a matriz de rigidez $\mathbf{K}$ associada a uma malha de elementos finitos tende a ser extremamente esparsa. Métodos que armazenem a matriz em formato esparso e tirem proveito dessa propriedade para a solução do sistema linear tendem a ser muito eficientes nessa situação. Nesse trabalho, priorizou-se métodos diretos de solução de sistemas lineares pela sua robustez e generalidade, em contraste com métodos iterativos, cuja convergência pode ser difícil de ser obtida em certos casos [Pellegrini, 2009]. Métodos diretos consistem na decomposição da matriz associada ao sistema linear em um produto de fatores que, uma vez obtidos, tornam a solução do sistema um mero processo de substituição.

No algoritmo em uso, esses métodos apresentam uma vantagem adicional: a mesma decomposição pode ser utilizada para resolver os dois sistemas lineares (solução para deslocamentos e pseudo-deslocamentos, ver Figura 5.1). Isso ocorre pois a matriz de rigidez só é alterada no final de cada iteração da programação linear sequencial, quando as densidades são atualizadas. Em particular, como a matriz de rigidez é simétrica positiva definida, opta-se pela fatoração de Cholesky, que possui a vantagem de ser numericamente estável [Golub e Van Loan, 1996]. Entretanto, e eficiência da solução de sistemas lineares esparsos por métodos diretos depende sensivelmente do preenchimento (*fill-in*) gerado durante a fatoração. Esse problema é evitado aplicando-se uma técnica de renumeração nodal adequada.

Além disso, existem algumas partes do algoritmo, como o cálculo dos filtros e das sensibilidades, que também podem ser feitos em paralelo. Isso pode ser realizado com diretivas OpenMP de laço paralelo, uma vez que esses cálculos consistem em operações independentes realizadas elemento a elemento.

5.2. Detalhamento do novo algoritmo

O funcionamento geral do novo código é mostrado no fluxograma da Figura 5.1. Manteve-se a opção pela linguagem C, modificando-se as rotinas onde necessário, por exemplo, para utilização das bibliotecas externas descritas nos próximos subcapítulos.

Basicamente, o código recebe como dados de entrada uma malha de elementos finitos previamente gerada, propriedades de materiais e parâmetros de otimização. Caso os filtros estejam ativos, determina-se uma lista de elementos vizinhos baseada no raio de filtragem especificado.

Então, inicia-se o processo iterativo de otimização utilizando programação linear sequencial. Em uma primeira etapa, utiliza-se a biblioteca Scotch para realizar a renumeração nodal, garantindo assim a eficiência da fatoração direta da matriz de rigidez. As matrizes de rigidez dos elementos finitos são então montadas e o campo de deslocamentos é encontrado com a ajuda da biblioteca MUMPS, que fatora e resolve o sistema $\mathbf{Ku} = \mathbf{f}$ para encontrar o vetor de deslocamentos $\mathbf{u}$. De posse do vetor $\mathbf{u}$, determinam-se as pseudo-forças necessárias no método adjunto de cálculo de sensibilidades e reutiliza-se a fatoração feita pelo MUMPS para determinar os pseudo-deslocamentos. Então, procede-se à análise de sensibilidade, atualizando em seguida os limites móveis e finalmente as densidades, utilizando para tanto uma rotina da biblioteca SLATEC, disponível gratuitamente em http://www.netlib.org (um repositório de programas matemáticos). Ela permite resolver de maneira eficiente problemas de programação linear com matrizes de restrição esparsas utilizando um algoritmo *simplex* primal-dual. O processo anterior é repetido até que o critério de convergência estabelecido seja satisfeito ou o número limite de iterações seja atingido.

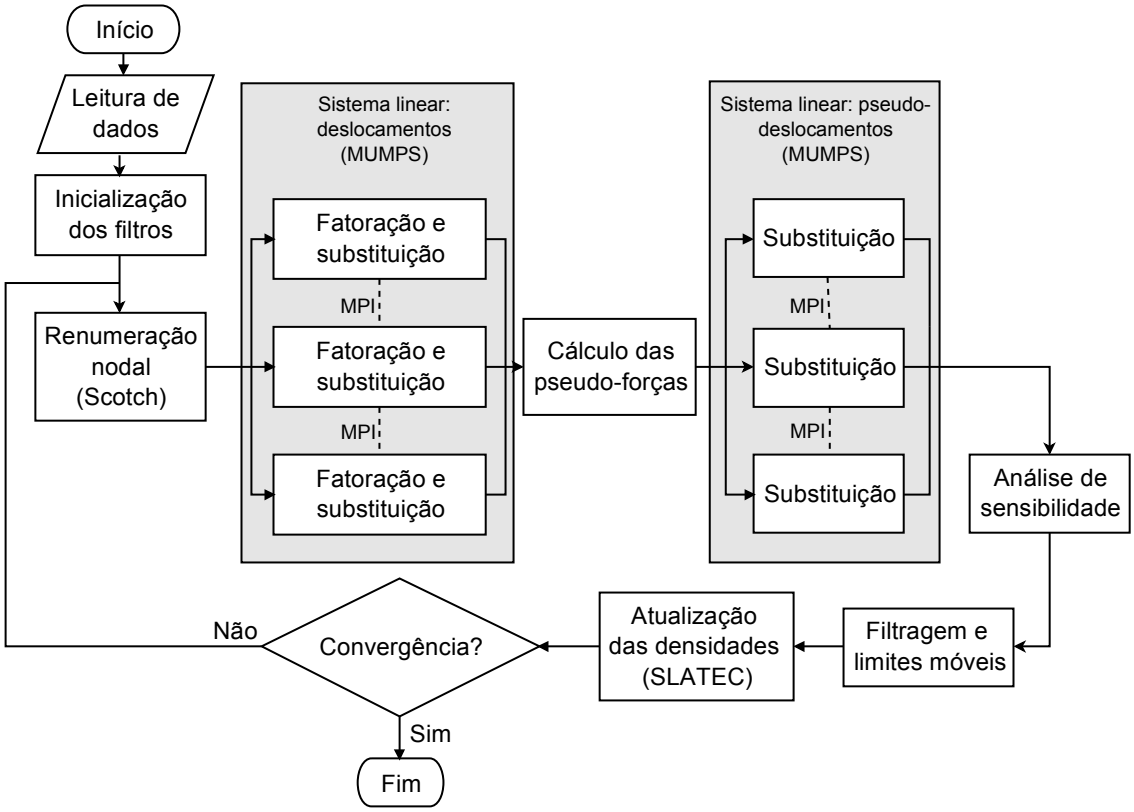

Figura 5.1 – Fluxograma simplificado da rotina de otimização topológica. O número de processos em paralelo é meramente ilustrativo.

A Figura 5.1 ilustra também as principais regiões do algoritmo que podem trabalhar em paralelo: a resolução do sistema linear para obtenção dos deslocamentos e pseudo-deslocamentos. Ressalta-se que o número de processos no fluxograma é apenas ilustrativo, podendo variar de um único ao número de processadores disponíveis.

As bibliotecas utilizadas bem como os algoritmos por elas empregados são detalhados nos próximos Subcapítulos.

5.3. Álgebra linear esparsa

A densidade de uma matriz pode ser definida como a razão entre o seu número de elementos não-nulos e seu número total de elementos [Pellegrini, 2009]. Em geral, os sistemas lineares oriundos de uma discretização por elementos finitos são representados por matrizes esparsas, ou seja, que apresentam uma baixa densidade. Isso ocorre pois o componente k_{ij} da matriz de rigidez $\mathbf{K}$ é não-nulo apenas se os graus de liberdade i e j estiverem ligados por meio de um elemento*. Como grande parte dos nós de uma malha não estão ligados diretamente uns aos outros, há um número relativamente pequeno de elementos não-nulos na matriz de rigidez. Em geral, quanto maior for a malha, menor será a densidade de $\mathbf{K}$.

Assim, métodos que explorem a esparsidade da matriz de rigidez, armazenando e operando apenas com seus coeficientes não-nulos tendem a ser eficientes. Para tirar proveito dessa vantagem, existem diversos esquemas de armazenamento de matrizes esparsas. Os métodos de interesse para esse trabalho são apresentados no Anexo I; para mais detalhes e outras possibilidades, ver Dongarra, 2000.

5.3.1. Fatoração de Cholesky

Um método de solução de sistemas lineares válido para matrizes simétricas positivo-definidas é a fatoração de Cholesky, uma variação do método da eliminação gaussiana onde a matriz é fatorada como $\mathbf{A} = \mathbf{LDL}^{\mathbf{T}}$, sendo $\mathbf{L}$ uma matriz triangular inferior e $\mathbf{D}$ uma matriz diagonal. Dessa forma, o sistema linear $\mathbf{Ax} = \mathbf{b}$ é resolvido em duas etapas: uma vez fatorada a matriz, resolve-se $\mathbf{LDy} = \mathbf{b}$, e em seguida, $\mathbf{L}^{\mathbf{T}}\mathbf{x} = \mathbf{y}$ (substituição) [Golub e Van Loan, 1996]. Esses dois sistemas são simples de se resolver pois as matrizes envolvidas são triangulares, bastando uma substituição sucessiva das variáveis já resolvidas para encontrar a solução.

5.3.2. A biblioteca MUMPS

MUMPS (*"a MUltifrontal Massively Parallel sparse direct Solver"*) é uma biblioteca voltada para a solução paralelizada de sistemas lineares esparsos por métodos diretos. O projeto vem sendo desenvolvido desde 1996 e seu código é de domínio público. A biblioteca é desenvolvida em Fortran 90 mas também possui interfaces em C e Matlab [MUMPS, 2011].

A biblioteca possui algoritmos de solução diretos para sistemas lineares esparsos, em versões específicas para matrizes que não são simétricas, matrizes simétricas gerais e matrizes simétricas positivo-definidas, que é o caso de interesse nesse trabalho. Além disso, é capaz de lidar tanto com matrizes de números reais quanto de números complexos, podendo trabalhar em precisão simples ou em precisão dupla.

Em matrizes simétricas positivo-definidas, a biblioteca realiza a decomposição $\mathbf{A} = \mathbf{LDL}^{\mathbf{T}}$, explorando o paralelismo oriundo de técnicas especiais de fatoração da matriz esparsa [Amestoy *et al.*, 2006].

Para realizar os cálculos paralelamente, além do protocolo de troca de mensagens MPI, MUMPS utiliza funcionalidades dos seguintes pacotes, todos disponíveis gratuitamente em http://www.netlib.org:

- BLAS (*Basic Linear Algebra Subprograms*): conjunto de rotinas que realiza operações básicas de álgebra linear (produto interno, multiplicação de matrizes...) de modo extremamente eficiente. Para maximizar o desempenho, existem versões especificamente desenvolvidas certas arquitetura de processadores.

* A não ser em casos especiais, como na aplicação de *multipoint constraints* (relação entre diversos graus de liberdade) [Bathe, 1996].

- BLACS (*Basic Linear Algebra Communication Subprograms*): uma série de rotinas de comunicação que permitem paralelizar a implementação do BLAS. Existem versões baseadas em diversos protocolos de comunicação, em particular, uma versão MPI.
- ScaLAPACK (*Scalable Linear Algebra PACKage*): um conjunto de rotinas desenvolvidas em Fortran 90 baseadas no BLAS/BLACS que permite realizar de maneira paralela uma série de tarefas como a solução de sistemas lineares e de problemas de valores próprios.

O processo de solução do sistema linear realizado pelo MUMPS pode ser divido em três grandes etapas:

- Análise: engloba operações como reordenamento, fatoração simbólica, estimativa do número de operações e da memória necessária para a fatoração. O pacote conta com diversos algoritmos de reordenamento, bem como uma interface para outras bibliotecas como a Scotch, que será utilizada nesse trabalho. Ao final dessa etapa, a matriz de entrada $\mathbf{A}$ é transformada na matriz pré-processada $\mathbf{A}_{\mathbf{pre}}$.
- Fatoração: a matriz a ser fatorada é distribuída entre os vários processos disponíveis de acordo com a árvore de eliminação, que expressa a interdependência entre as tarefas. Esse processo para realizar $\mathbf{A}_{\mathbf{pre}} = \mathbf{LDL}^{\mathbf{T}}$ é chamado de *abordagem multifrontal* [Amestoy *et al.*, 2001]. O algoritmo utilizado emprega mapeamento dinâmico de tarefas e comunicação assíncrona entre processos, o que permite que ele se adapte durante a sua execução, distribuindo as tarefas da maneira mais adequada entre os processadores disponíveis.
- Solução: resolve efetivamente o sistema linear $\mathbf{LDL}^{\mathbf{T}}\mathbf{x}_{\mathbf{pre}} = \mathbf{b}_{\mathbf{pre}}$, onde $\mathbf{x}_{\mathbf{pre}}$ e $\mathbf{b}_{\mathbf{pre}}$ são os vetores de deslocamentos e de forças, respectivamente, pré-processados de acordo com $\mathbf{A}_{\mathbf{pre}}$. Isso é feito com a etapa da substituição, onde se resolve $\mathbf{LDy} = \mathbf{b}_{\mathbf{pre}}$, seguida da retro-substituição, onde se resolve $\mathbf{L}^{\mathbf{T}}\mathbf{x}_{\mathbf{pre}} = \mathbf{y}$, encontrando o vetor de deslocamentos. O vetor solução alterado é então transformado de volta a sua forma original, $\mathbf{x}$.

A biblioteca oferece a possibilidade de o processador principal (*host*) trabalhar junto nas etapas de análise e fatoração, o que aumenta seu desempenho, a não ser que a matriz seja distribuída entre muitos processos e um processador exclusivo para gerenciar a distribuição de dados e a compilação dos resultados seja necessário.

5.4. Renumeração nodal

5.4.1. *Fill-in* na fatoração de matrizes esparsas

O *fill-in* ocorre durante a fatoração de uma matriz quando elementos que eram inicialmente nulos tornam-se não nulos em um dos fatores[†]. Evidentemente, esse fenômeno é indesejado quando se trabalha com matrizes armazenadas em forma esparsa, uma vez que incorre em espaço adicional de armazenamento e em operações adicionais nos termos extras.

Basicamente, um grafo é uma estrutura composta de um conjunto de vértices ligados por arestas. O grafo de adjacência de uma matriz simétrica $\mathbf{A}$ de ordem n é composto por n vértices ligados por arestas apenas se o elemento $a_{i,j}$ (e consequentemente $a_{j,i}$) for não-nulo. Em suma, ele consiste numa representação da topologia da matriz e é útil para desenvolver algoritmos que operem nas propriedades estruturais de matrizes esparsas [Pellegrini, 2009].

[†] O elemento criado pelo *fill-in* pode ser numericamente igual a zero, mas será alocado da mesma maneira na memória do computador durante a solução do sistema linear esparso. Esse caso é chamado de *fill-in* simbólico, em contraste com o *fill-in* numérico.

Devido à natureza da eliminação gaussiana, um elemento $a_{i,j}$ de uma matriz incorrerá em *fill-in* durante sua fatoração quando existe em seu grafo de adjacência um caminho entre os vértices v_i e v_j passando apenas por vértices que tenham numeração menor que $\min(i, j)$ [Pellegrini, 2009]. Desse modo, a quantidade de *fill-ins* durante a fatoração da matriz de rigidez é extremamente dependente da numeração nodal empregada na malha de elementos finitos. Como isso influencia diretamente a quantidade de operações realizadas pelo algoritmo de solução de sistemas lineares esparsos e portanto sua eficiência em termos de memória e custo computacional, é imperativo buscar numerações nodais que minimizem o *fill-in*. No caso geral, deve-se ter cuidado na permutação de linhas e colunas da matriz, uma vez que essas podem introduzir instabilidades numéricas na solução; entretanto, esse problema não se manifesta quando se utiliza a decomposição $\mathbf{LDL^T}$ ao invés da decomposição $\mathbf{LU}$ tradicional, pois a decomposição de Cholesky é numericamente estável independentemente do ordenamento [Golub e Van Loan, 1996]. Isso leva à importante consequência que as etapas de fatoração e reordenamento podem ser consideradas independentemente.

5.4.2. Secção recursiva

Um método eficiente para encontrar ordenamentos que minimizem o *fill*-in é o da secção recursiva (*recursive nested dissection*), empregado pela biblioteca Scotch [Pellegrini, 2010]. Resumidamente, o método consiste em determinar um conjunto S de vértices do grafo separando-o em duas partes A e B. Aos vértices do conjunto S são associados os maiores números nodais disponíveis, garantindo que entre quaisquer pares formados entre um vértice do conjunto A e um do conjunto B não haja um caminho passando unicamente por vértices com numeração nodal menor do que os números do par. Dessa forma, impede-se a formação de *fill-in* entre os dois conjuntos de nós. O processo de divisão e numeração da fronteira é repetido recursivamente, até que os conjuntos sejam todos menores do que um valor limite [Pellegrini, 2009].

Este algoritmo é por natureza paralelizável, já que quando um grafo distribuído entre p processos é separado, cada uma das partes criadas pode ser tratada por $\frac{p}{2}$ processos de maneira independente.

5.4.3. As bibliotecas Scotch e PT-Scotch

Scotch é uma biblioteca que realiza tarefas como partição de grafos, mapeamento estático, reordenamento de matrizes esparsas e partição de malhas. A biblioteca é distribuída livremente com código fonte aberto, sob uma licença que garante aos seus usuários o direito de modificá-la e redistribuí-la gratuitamente observando algumas condições [Pellegrini, 2010].

Nesse trabalho, interessa-se apenas à sua capacidade de reordenamento de matrizes esparsas utilizando o algoritmo do *nested dissection* recursivo anteriormente descrito. A biblioteca usa como dados de entrada o grafo de adjacência associado à matriz. No entanto, também existe a possibilidade de utilizar um arquivo representando a malha de elementos finitos, onde constam as suas conectividades. A partir desse arquivo, a biblioteca cria o grafo de adjacência da matriz e nele opera, produzindo como resultado uma lista contendo a nova numeração nodal.

Também está disponível a biblioteca PT-Scotch, que é uma versão com rotinas paralelas da Scotch. Essa versão emprega uma série de técnicas para particionar o grafo de adjacência e aplicar o *nested dissection* com vários processos simultâneos [Pellegrini, 2010]. Entretanto, nesse trabalho restringiu-se apenas à versão sequencial da biblioteca, pois o desempenho da versão paralela foi inferior. Isso provavelmente se deve ao tamanho dos grafos processados, que continham no máximo algumas dezenas de milhares de vértices. Nesse caso, a renumeração é um processo extremamente rápido e aumento de velocidade devido à paralelização é compensado pela sobrecarga devido à comunicação entre os processos.

6. RESULTADOS E DISCUSSÃO

O principal resultado desse trabalho é a medida do ganho de velocidade obtido com as técnicas empregadas de renumeração nodal e solução de sistemas lineares esparsos. Para diversos exemplos de resultados que podem ser obtidos com a formulação de otimização específica empregada nesse trabalho, ver Guilherme, 2006, e Kuckoski, 2009.

Após implementar a interface do código anterior em C com as bibliotecas escolhidas, testou-se extensivamente a consistência dos novos resultados, comparando-os com as soluções do antigo *solver skyline* e confirmando sua validade.

Em seguida, alguns testes com malhas de diferentes tamanhos permitem comparar o desempenho do MUMPS com o *solver skyline*. Em todos os casos, utilizam-se malhas regulares feitas com o elemento hexaédrico trilinear isoparamétrico, que possui oito nós, cada um com três graus de liberdade de deslocamento (nas três dimensões). As funções de interpolação desse elemento e outros detalhes de sua formulação podem ser encontrados em Bathe, 1996, e Guilherme, 2006. Além disso, utiliza-se integração numérica com dois pontos de Gauss por direção e as densidades são tomadas como constantes em cada elemento.

A Tabela 6.1 sintetiza os tempos de solução obtidos com diversas malhas, assim como diversas de suas características relevantes para a solução. Todos os cálculos foram realizados em um processador Athlon II X4 645, que possui quatro núcleos de frequência 3.10 GHz, em um computador que dispunha de 4 GB de memória RAM. Os tempos de execução mostrados são a média de cinco execuções distintas. Os resultados do MUMPS foram realizados utilizando os quatro núcleos do processador, ao passo que o algoritmo *skyline*, por ser puramente sequencial, utiliza um único núcleo.

Tabela 6.1 – Comparação entre os tempos de solução com o MUMPS e o algoritmo *skyline*.

Ordem da matriz	Número de elementos não-nulos	Densidade (%)	Tamanho do vetor *skyline* (sem renumeração nodal)	Tamanho do vetor *skyline* (com renumeração nodal)	Tempo de solução - *skyline* (sem renumeração) (s)	Tempo de solução *skyline* (com renumeração) (s)	Tempo de solução MUMPS (s)
45	655	63,3	739	946	< 0,01[‡]	<0,01	1,09
1080	32049	5,49	184357	217000	0,14	0,22	1,12
33952	1224531	2,12	178307903	57434966	> 3600[§]	607	5,4
65520	2386237	0,111	604987741	137425770	> 3600	1815	11,3
67432	2455846	0,108	668870902	144605811	> 3600	1912	11,7

Note-se que para matrizes de ordem muito pequena, a solução com o MUMPS é mais lenta do que a do método *skyline*. Isso se deve provavelmente ao tempo gasto na criação e comunicação entre os diversos processos gerados em paralelo pelo MUMPS, que nesse caso não são compensados pelo aumento de velocidade de processamento. Entretanto, as matrizes de ordem maior possuem densidade extremamente reduzida, e o MUMPS resolve os sistemas lineares de maneira extremamente eficiente.

A simples comparação de velocidades entre o MUMPS e o algoritmo *skyline* é um tanto injusta, uma vez que a renumeração nodal aplicada não possui a função específica de melhorar a eficiência desse tipo de algoritmo, apesar de melhorá-la substancialmente. Infelizmente, não se dispõe dos tempos computacionais empregados na solução com o Ansys utilizada por Kuckoski, 2009, para uma comparação mais apurada.

[‡] Nesse caso, a solução é tão rápida que as rotinas utilizadas para medir o tempo empregado não conseguem captar o tempo de solução.

[§] Os cálculos não foram terminados nesses casos.

Outro resultado interessante diz respeito à escalabilidade da solução com o MUMPS, ou seja, a relação entre o ganho de velocidade pela paralelização e o número de processadores em uso, mostrada na Tabela 6.2. Apesar do ganho de velocidade entre a utilização de 3 e 4 núcleos ter sido desprezível, ele é considerável em relação à execução em um único núcleo, comprovando assim a utilidade de se trabalhar com todos os processadores disponíveis.

Na execução de alguns exemplos com o código, percebeu-se que as partes sequenciais potencialmente paralelizáveis com a utilização do OpenMP que mais tomavam tempo de execução eram os cálculos de elementos vizinhos para utilização dos filtros. Apesar de serem executados uma única vez no início do algoritmo, apresentaram um ganho considerável de velocidade ao serem paralelizados, como mostra a Tabela 6.2. O fato de 4 núcleos terem apresentado um aumento do tempo em relação ao uso de 3 núcleos provavelmente se deve ao grande número de *threads* criados e finalizados no processo de paralelização dos laços que encontram os vizinhos.

Tabela 6.2 – Tempo de solução do sistema linear utilizando o MUMPS e tempo de inicialização dos filtros em função do número de núcleos empregados no cálculo.

Número de núcleos em uso	Tempo de solução do sistema linear (s)	Tempo de inicialização dos filtros (s)
1	26,6	134,1
2	15,5	69,5
3	13,1	54,4
4	11,7	112,8

Nos exemplos executados, o cálculo de sensibilidade, uma vez obtidos os pseudo-deslocamentos, era bastante rápido, de maneira que não se julgou necessário paralelizá-lo. Eventualmente para a solução de problemas com número muito maior de elementos, eles podem tomar tempo considerável e nessa situação teriam um benefício ao serem paralelizados. Nesse caso, os ganhos representados na última coluna da Tabela 6.2 fornecem uma estimativa da ordem de magnitude esperada para o aumento de velocidade.

7. CONCLUSÕES

Pode-se afirmar que o trabalho cumpriu seu objetivo principal de acelerar o código de solução de problemas de otimização topológica com restrição de tensão.

Apesar de sua generalidade, o método ainda está longe de ser consolidado, uma vez que seus resultados são dependentes de muitos parâmetros determinados heuristicamente, exigindo um grande esforço do usuário para conseguir bons resultados [Kuckoski, 2009 e Guilherme, 2006]. Como continuidade do trabalho realizado, pode-se enriquecer a ferramenta com outras formulações de elementos finitos, modelos materiais, ou técnicas de otimização.

Além disso, de um ponto de vista computacional, outra tendência recente em termos de computação de alto desempenho é a utilização de GPUs (processadores de placas gráficas) para realizar cálculos científicos. Esse tipo de placa é extremamente eficiente para computação em paralelo, pois possui dezenas de processadores e uma grande quantidade de memória disponível. Uma maneira de acelerar ainda mais o algoritmo utilizado seria com o uso de um *solver* desenvolvido com o intuito de operar nesse tipo de processador. Atualmente existem alguns *solvers* gratuitos desenvolvidos em CUDA (uma extensão de C/C++ lançada pela NVIDIA que permite o processamento utilizando suas GPUs), como o MAGMA (*Matrix Algebra on GPU and Multicore Architectures*), disponível em http://icl.eecs.utk.edu/magma/. Também pode-se mencionar o OpenCL (*Open Computing Language*), uma linguagem de programação livre para programação paralela em sistemas heterogêneos (por exemplo, CPUs + GPUs), disponível em http://www.khronos.org/opencl/.

REFERÊNCIAS BIBLIOGRÁFICAS

Amestoy, P.R., Guermouche, A., L'Excellent, J.-Y., Pralet, S.; **"Hybrid Scheduling for the Parallel Solution of Linear Systems"**, Parallel Computing, Vol. 32 (2), pp 136-156, 2006.

Amestoy, P.R., Duff, I.S., Koster, J., L'Excellent, J.-Y.; **"A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling"**, SIAM Journal of Matrix Analysis and Applications, Vol 23, No 1, pp 15-41, 2001.

Barney, B.; **"Introduction to Parallel Computing"**, Lawrence Livermore National Laboratory, disponível em https://computing.llnl.gov/tutorials/parallel_comp/, 2011. Acessado em 09/11/2011.

Bathe, K.J.; **"Finite Element Procedures"**, Prentice Hall, 1996.

Bendsøe, M.P.; **"Optimization of Structural Topology, Shape and Material"**, Springer, 1995.

Dongarra, J.J.; **"Sparse Matrix Storage Formats"**, disponível em http://web.eecs.utk.edu/~dongarra/etemplates/node372.html, 2000. Acessado em 09/11/2011.

Guilherme, C.H.M.; **"Otimização Topológica e Cálculo do Gradiente de Forma para Estruturas Submetidas à Restrição de Fadiga"**, Tese de Doutorado – Departamento de Engenharia Mecânica, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2006.

Foster, I.; **"Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering"**, Addison-Wesley, 1995.

Golub, G.H., Van Loan, C.F.; **"Matrix Computations"**, The Johns Hopkins University Press, 3ª edição, 1996.

Haftka, R.T., Gürdal, Z.; **"Elements of Structural Optimization"**, Kluwer Academic Publishers, 3ª edição, 1992.

Kuckoski, A.; **"Otimização Topológica de Estruturas com Restrição de Falha"**, Trabalho de Conclusão do Curso de Engenharia Mecânica – Departamento de Engenharia Mecânica, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2009.

MUMPS; **"MUMPS: a Multifrontal Massively Parallel Sparse Direct Solver"**, versão 4.10.0, disponível em http://graal.ens-lyon.fr/MUMPS/, 2011. Acessado em 09/11/2011.

Pellegrini, F.; **"Contributions au partitionnement de graphes parallèle multi-niveaux"**, Habilitation à Diriger des Recherches – Laboratoire Bordelais de Recherche en Informatique, Université de Bordeaux I, 2009.

Pellegrini, F.; **"Scotch and LibScotch 5.1 User's Guide"**, INRIA Bordeaux Sud-Ouest, Université de Bordeaux I, disponível em http://www.labri.fr/perso/pelegrin/scotch/, 2010. Acessado em 09/11/2011.

van der Steen, A.J., Dongarra, J.J.; **"Overview of Recent Supercomputers"**, disponível em http://www.phys.uu.nl/~steen/web03/overview.html, 2003. Acessado em 09/11/2011.

ANEXO I. ARMAZENAMENTO DE MATRIZES EM FORMATO ESPARSO

I.1. Armazenamento de matrizes em formato *skyline*

Além de ser uma matriz esparsa, a matriz de rigidez é simétrica positiva definida (quando as condições de contorno estão devidamente aplicadas) e pode ser armazenada em banda [Bathe, 1996]. O formato *skyline* baseia-se nessas propriedades para armazenar a matriz como dois vetores: cada coluna de uma das metades da matriz (triangular inferior ou triangular superior) é armazenada entre sua diagonal principal e o elemento não-nulo mais distante. O vetor A_{sk} contém os componentes de cada coluna até a altura necessária (último elemento não-nulo) e END_A contém índices referenciando os elementos do primeiro vetor onde cada coluna começa. Dessa maneira, os índices de linha e coluna de cada elemento da matriz são facilmente deduzidos e não precisam ser armazenados diretamente. A Figura I.1 elucida com um exemplo simples a descrição anterior.

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

$$A_{sk} = [\times,\times,\times,\times,0,\times,\times,0,\times,\times,0,0,\times] \qquad A_{sp} = [\times,\times,\times,\times,\times,\times,\times,\times,\times]$$

$$IRN_A = [1,2,1,3,1,4,2,5,2]$$

$$END_A = [1,2,4,7,10,14] \qquad JCN_A = [1,2,2,3,3,4,4,5,5]$$

Figura I.1 – Armazenamento de uma matriz simétrica em formato *skyline* e em formato esparso (à direita). O símbolo $\times$ representa um elemento não-nulo arbitrário.

Como se pode perceber, a eficiência desse tipo de armazenamento é extremamente dependente da numeração nodal da malha de elementos finitos, uma vez que isso determinará a altura das colunas e consequentemente, o número de zeros armazenados no vetor A_{sk}.

I.2. Armazenamento de matrizes em formato esparso

O esquema de armazenamento esparso utilizado nesse trabalho é mostrado na Figura I.1: emprega-se simplesmente três vetores para armazenar os valores não-nulos da matriz (A_{sp}), seus respectivos índices de linha (IRN_A) e de coluna (JCN_A). Embora nesse exemplo o armazenamento esparso empregue mais elementos, em geral ele é muito mais eficiente do que o formato *skyline* para matrizes de ordem maior. Isso ocorre pois para malhas com grande número de nós, torna-se cada vez mais difícil encontrar numerações nodais eficientes que evitem o armazenamento de elementos nulos no formato *skyline*, ao passo que o armazenamento esparso não apresenta essa dependência da numeração nodal.

Embora os *solvers* que utilizem o armazenamento esparso da matriz do sistema linear sejam mais complexos, em geral apresentam melhor desempenho, tendo em vista que o número de operações envolvendo elementos nulos é minimizado.

ANEXO II. DETALHAMENTO DA FORMULAÇÃO EMPREGADA

Qualquer problema de otimização com uma única função objetivo $f(\mathbf{x})$ pode ser expresso na forma padrão [Haftka e Gürdal, 1992]:

$$\begin{aligned} &\text{Minimizar } f(\mathbf{x}) \\ &\text{Sujeito a } \begin{matrix} g_j(\mathbf{x}) \geq 0, & j = 1,\ldots,n_g \\ h_k(\mathbf{x}) = 0, & k = 1,\ldots,n_e \end{matrix} \end{aligned}$$

onde $\mathbf{x} = (x_1, x_2, x_3, \ldots, x_n)$ denota o vetor contendo as n variáveis de projeto, $g_j(\mathbf{x})$ são as n_g restrições de desigualdade e $h_k(\mathbf{x})$ são as n_e restrições de igualdade.

A relação adotada entre as variáveis de projeto (densidades) e as propriedades mecânicas do material é linear:

$$\begin{aligned} &\mathbf{E} = \rho(x,y,z)\mathbf{E}^0 \\ &\int_\Omega \rho(x,y,z)\, d\Omega \leq V \\ &0 < \rho(x,y,z) \leq 1 \end{aligned}$$

onde $\mathbf{E}^0$ representa o tensor constitutivo de quarta ordem do material de origem (sem poros), $\mathbf{E}$ o tensor constitutivo do material levando em conta sua densidade $\rho(x,y,z)$ em um ponto do domínio Ω, de volume V.

A função objetivo adotada, $f(\mathbf{x}) = f(\rho)$, é dada por:

$$f(\rho) = \int_\Omega \left[\rho^P + \alpha\rho(1-\rho)\right] d\Omega = \sum_{e=1}^{N} \left[\rho^P + \alpha\rho(1-\rho)\right] v_e$$

onde v_e é o volume do elemento e, P é o coeficiente de penalização de densidades intermediárias e α um fator que determina a influência do segundo termo da equação na função objetivo, também com o objetivo de penalizar densidades intermediárias.

A forma particular de restrição global de tensão foi proposta por Guilherme [Guilherme, 2006]:

$$g = \left\{ \sum_{e=1}^{N} \left[\frac{v_e}{V} \max\left(0; \frac{\sigma_{VM,e}}{\sigma_l} + \varepsilon^{\rho_e} - \varepsilon\right) \right]^P \right\}^{\frac{1}{P}} \leq 1$$

onde N é o número total de elementos da malha, $\sigma_{VM,e}$ é a tensão equivalente de von Mises no elemento e, σ_l é a tensão limite admissível no material, V é o volume total da estrutura, ρ_e é a densidade do elemento e, $0 < \varepsilon \leq 1$ é o coeficiente empregado na técnica de relaxação-ε e $\max(\ )$ é a função que retorna o valor máximo de uma lista de argumentos. Essa equação

reflete o método das restrições ativas: apenas contribuições de sinal positivo são contabilizadas na função restrição global [Guilherme, 2006].

Para que se resolva o problema de otimização com um algoritmo de programação linear sequencial, a forma padrão do problema de otimização apresentada acima é linearizada como:

$$\text{Minimizar } f(\mathbf{x}) = f(\mathbf{x}^*) + \sum_{e=1}^{N} \left(x_e - x_e^*\right) \left.\frac{\partial f}{\partial x_e}\right|_{x^*}$$

$$\text{Sujeito a } g_j(\mathbf{x}) = g_j(\mathbf{x}^*) + \sum_{e=1}^{N} \left(x_e - x_e^*\right) \left.\frac{\partial g_j}{\partial x_e}\right|_{x^*} \geq 0, \qquad j = 1, \ldots, n_g$$

$$h_k(\mathbf{x}) = h_k(\mathbf{x}^*) + \sum_{e=1}^{N} \left(x_e - x_e^*\right) \left.\frac{\partial h_k}{\partial x_e}\right|_{x^*} = 0, \qquad k = 1, \ldots, n_e$$

$$x_e^L \leq x_e - x_e^* \leq x_e^U$$

onde $\mathbf{x}^*$ é um ponto na vizinhança de $\mathbf{x}$, N é o número de variáveis de projeto (que nesse caso é igual ao número de elementos da malha), x_e^L e x_e^U representam os limites móveis, garantindo que $\mathbf{x}$ e $\mathbf{x}^*$ estejam próximos o suficiente para que a linearização seja válida.

Os valores dos limites móveis não são constantes, uma vez que essa abordagem pode dificultar a convergência do método iterativo [Guilherme, 2006]. A cada iteração, os limites são atualizados de acordo com o seguinte esquema:

$$x_e^L = x_e - A|x_e| \qquad x_e^U = x_e + A|x_e|$$

onde A possui valor inicial de 10%. Caso as densidades de um elemento aumentem em duas iterações subsequentes, seu valor é aumentado em 5%; caso as densidades do elemento diminuam por duas iterações subsequentes, seu valor é diminuído em 5%, sempre observando os limites $5\% \leq A \leq 20\%$.