IMPLEMENTAÇÃO PARALELA PARA ANÁLISES ESTÁTICAS LINEARES PELO MÉTODO DOS ELEMENTOS FINITOS

Gabriela Moreira Azevedo

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ESTRUTURAS

"IMPLEMENTAÇÃO PARALELA PARA ANÁLISES ESTÁTICAS LINEARES PELO MÉTODO DOS ELEMENTOS FINITOS"

Gabriela Moreira Azevedo

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Estruturas da Escola de Engenharia da Universidade Federal de Minas Gerais, como parte dos requisitos necessários à obtenção do título de "Mestre em Engenharia de Estruturas".

Comissão Examinadora:

Prof. Dr. Samuel Silva Penna
DEES - UFMG (Orientador)

Prof. Dr. Ramon Pereira da Silva
DEES - UFMG (Coorientador)

Prof. Dr. Roque Luiz da Silva Pitangueira
DEES - UFMG

Prof. Dr. Rodrigo Weber dos Santos
UFJF

Belo Horizonte, 08 de novembro de 2019

Azevedo, Gabriela Moreira.
A994i Implementação paralela para análises estáticas lineares pelo método dos elementos finitos [recurso eletrônico] /Gabriela Moreira Azevedo. - 2019.
1 recurso online (xiii, 140 f. : il., color.) : pdf.

Orientador: Samuel Silva Penna.
Coorientador: Ramon Pereira da Silva.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f.113-130.
Bibliografia: f.131-140.

Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia de Estruturas - Teses. 2. Programação paralela (Computação) - Teses. 3. Linguagem de programação orientada a objetos - Teses. 4. Modelo de métodos finitos - Teses. I. Penna, Samuel Silva. II. Silva, Ramon Pereira da. III. Universidade Federal de Minas Gerais. Escola de Engenharia. IV. Título.

CDU: 624(043)

Ficha catalográfica: Biblioteca Profº Mário Werneck, Escola de Engenharia da UFMG

Índice

Lista de Tabelas

Lista de Figuras

Resumo

Na análise estrutural por elementos finitos, a representação de estruturas e fenômenos complexos é cada vez mais recorrente e acaba exigindo uma quantidade cada vez maior de graus de liberdade dos modelos. Esta condição representa um problema significativo de desempenho e demanda de memória quando considera-se as etapas de montagem e resolução dos sistemas de equações. Dessa forma, faz-se necessário o uso de computadores de fina tecnologia e alta performance. Contudo, o uso destas máquinas especializadas é extremamente ineficiente quando considerado o seu custo e, conforme o avanço da tecnologia, a necessidade de ocasional substituição.

Visando resolver esse impasse, tem-se dado atenção para soluções que façam uso de computadores paralelos. Nesta modalidade da computação, um problema grande e complexo é dividido em pequenas porções, que são resolvidas de forma independente em diferentes computadores. Assim, a computação paralela tem como vantagens a menor demanda de capacidade em componentes como memória e processador, além da possibilidade de fácil expansão do sistema distribuído. Entretanto, a concorrência de tarefas demanda algoritmos diferentes em relação às rotinas sequenciais, principalmente quando objetiva-se a alta performance. Também, para que os resultados da computação paralela sejam superiores, a divisão das tarefas deve ser realizada de forma que todos os processos terminem seus trabalhos ao mesmo tempo, evitando a ociosidade de processadores, juntamente com o mínimo de comunicação possível.

No caso do método dos elementos finitos, devem ser utilizadas metodologias que realizem a divisão homogênea dos elementos entre os processadores ao mesmo

tempo que minimizem as fronteiras entre as divisões. Desta forma, as determinações para evitar desperdício computacional distribuindo cargas iguais de trabalho e gerar menor comunicação entre processadores é atendida.

Portanto, este trabalho tem como intuito uma implementação paralela para o método dos elementos finitos. Para tanto, a implementação foi realizada usando o sistema computacional INSANE (*INteractive Sructural ANalysis Environment*), desenvolvido na linguagem de programação Java, segundo o paradigma de Orientação à Objetos, e o padrão MPI (*Message Passing Interface*) de comunicação de dados entre computadores.

Palavras-Chave: Computação Paralela, Elementos Finitos, MPI, HPC

Abstract

In the finite element structural analysis, complex mechanisms representation in structures are increasing and requiring models with a great number of degrees of freedom. This situation represents a significant problem both in performance and memory usage on computers, particularly on phases such as assembling and solving of the equations system. Thus, there is a great need for high technology and performance computers. However, the use of these machines is very inefficient when considering its cost and, as the technology advances, the need for occasional replacement.

In order to solve this situation, attentions turned to the use of parallel computers. In this system modality, a big and complex problem is divided in smaller portions, that are individually solved in different computers. Therefore, parallel computing bring great advantages, such as smaller demand on components like memory and processor, besides the easiness to add new units on the system. Nonetheless, the concurrency demand the use of algorithms other than those from sequential computing, especially when high performance is needed. For the resulting time of a parallel computing be advantageous, the division of tasks must be performed in such way that all processes end at the same time along with minimum communication.

Considering the finite element method, procedures that perform a homogeneous division of elements among all processors and domains frontier minimization are essential. As a consequence of this, the requirement for well balanced workload and minimal communication is met.

This work addresses a finite element method parallel implementation. For this,

the INSANE (Interactive Structural Analysis Environment) system, the Java programming language, the Object Oriented paradigm and the MPI (Message Passing Interface) pattern of data exchenge and communication are used.

Keywords: Parallel Computing, Finite Element Method, MPI, HPC

Agradecimentos

Agradeço, primeiramente, a todos brasileiros e brasileiras que possibilitam a existência da pesquisa e ciência de qualidade através do financiamento das nossas Universidades Públicas. Da mesma forma, também agradeço à Universidade Federal de Minas Gerais, ao Departamento de Engenharia de Estruturas, aos meus Professores e às instituições de fomento à pesquisa, em especial a CAPES, por proporcionarem minha educação.

Aos meus professores Orientador, Samuel Silva Penna, e Coorientador, Ramon Pereira da Silva, por aceitarem participar da minha pesquisa, plenamente presentes e solícitos e absolutamente fundamentais no desenvolvimento do trabalho.

À minha família por todo apoio e compreensão, essenciais para a conclusão deste ciclo.

Ao Ricardo pela incrível parceria que só me encoraja a fazer meu melhor em todos aspectos da vida.

Aos meus queridos amigos, que todos os dias compartilham das minhas dificuldades, e dos quais sinto tanta falta.

Aos meus colegas de mestrado e do grupo INSANE pelo companheirismo e espírito de equipe.

Capítulo 1

Introdução

O surgimento e avanço da tecnologia computacional proporcionou grandes desenvolvimentos nos campos de análises e de simulações, trazendo melhorias de representação e complexidade na modelagem de fenômenos. No entanto, as áreas de pesquisa em métodos numéricos, hoje possibilitados pela era da computação, não devem limitar-se em si, tornando obrigatório estudar, também, a aplicação de novas tecnologias, dada a constante evolução computacional.

Desde a origem dos computadores, a busca por maior poder computacional sempre foi objeto de dedicação máxima uma vez que a demanda por mais capacidade somente aumenta. Sendo os dois principais fatores limitantes a velocidade de processamento e tamanho de memória, o rápido desenvolvimento da tecnologia de fabricação de transístores acelerou melhorias nestes dois componentes chave. Conforme a lei de Moore, o número de transístores por área de *chip* é duplicado a cada 18 meses (Tanenbaum e Austin, 2013), consequentemente, aumentando o tamanho da memória e velocidade de transmissão de dados. Entretanto, as dificuldades da atual escala nanométrica já não permitem esse acelerado avanço, principalmente devido a problemas como dissipação de calor/energia e escapamento de elétrons, efeitos estes que inviabilizam a produção de *chips* ainda menores. Como consequência direta, tem-se que as altas taxas de progresso vivenciadas anteriormente mostram-se no fim, e o limite desta tecnologia pode estar próximo. Sendo assim, os fabricantes

de computadores, hoje, buscam novos recursos para superar suas marcas de performance, como, por exemplo, usando processadores *multicor*e, processadores vetoriais e hierarquia de memórias.

Enquanto os computadores não atendem as mais altas exigências, como ainda é caso de muitos estudos na academia e, cada vez mais expressivamente, de aplicações comerciais e industriais, são utilizados os *clusters*. Este recurso já é bastante antigo, desde a década de 70 utilizam-se computadores conectados em redes. Com isto, problemas grandes demais para serem armazenados na memória de um único computador ficam distribuídos entre várias máquinas. Da mesma forma, cálculos muito extensos são fracionados para serem solucionados simultaneamente.

Nos últimos anos, tornou-se mais clara a necessidade da computação paralela, principalmente com a emergência da indústria 4.0, do aprendizado de máquina e da *Big Data*. No entanto, tanto para as novas arquiteturas de computadores *multicores* quanto para os computadores paralelos, a maior limitação ocorre no fato de que o *software* deve ser explicitamente produzido de forma a aproveitar todas as capacidades da paralelização. Williams (2008) explica que, como os compiladores falharam no quesito de produzir automaticamente códigos paralelos e eficientes, tornou-se responsabilidade dos programadores – especialistas em arquitetura computacional e compiladores – a produção de aplicações, bibliotecas e *frameworks* especializados em extrair alta performance. Nesta área, já existem diversas bibliotecas que propõem-se a obter máxima eficiência, seja pela escrita de algoritmos eficientes ou por aplicações que automaticamente encontram a melhor composição das funções para a arquitetura do computador, conhecidas como *auto-tuners*. Essas ferramentas são resultados de anos de pesquisas e testes, que geralmente são sintetizadas em bibliotecas/*frameworks*/API's livres ou proprietárias.

Nas simulações computacionais, como nas baseadas em elementos finitos, há a necessidade de um grande volume de dados e cálculos, como, por exemplo, em análises de efeitos muito pequenos e/ou concentrados, como vórtices em escoamentos

ou em descontinuidades no material, ou em problemas de escalas gigantescas, como modelagem de bacias ou da camada limite atmosférica. Nestes casos, a computação paralela pode ser um recurso indispensável tanto para agilizar o tempo de processamento quanto para viabilizar a própria montagem do problema. Diante disso, é clara a necessidade de que pesquisadores envolvidos nestas áreas também disponham-se a explorar os ramos da computação paralela e de alto desempenho.

1.1 O Projeto INSANE

O sistema INSANE (*INteractive Structural ANalysis Environment*) é um *software* de código aberto, desenvolvido e atualizado pelos professores e estudantes do núcleo de estudos de Métodos Numéricos e Computacionais para a Engenharia no Departamento de Engenharia de Estruturas da Universidade Federal de Minas Gerais. Criado com o propósito de ser uma ferramenta para o aprendizado iterativo nas disciplinas de análise estrutural, o sistema acabou evoluindo para uma plataforma de fomento à pesquisa, com o desenvolvimento de implementações em elementos finitos, elementos finitos generalizados, elementos de contorno e métodos sem malha, todos aplicados à engenharia de estruturas. Por ser instrumento utilizado na pesquisa dos discentes, metodologias de análise cada vez mais complexas são incorporadas ao *software*, de forma que espera-se ser possível a execução destes novos algoritmos. Entretanto, estas modificações geralmente surgem com uma maior exigência computacional que, se não atendida, inviabiliza o prosseguimento dos estudos. E é neste contexto que este trabalho insere-se, de forma a trazer novas perspectivas de computação de alto desempenho e computação paralela para o grupo de pesquisa.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo deste trabalho é o estudo e implementação de rotinas paralelas para a solução de análises estruturais estáticas pelo método dos elementos finitos no sistema INSANE.

1.2.2 Objetivos Específicos

Os objetivos específicos desta dissertação são:

1. refatoração das classes *Solution* para a criação de um padrão para a solução de equações nos moldes da Orientação a Objetos e outro para a utilização de bibliotecas nativas pelo JNI;
2. integração do padrão de programação MPI ao sistema INSANE;
3. implementação de métodos nativos para a partição de malhas;
4. implementação de métodos nativos para a solução paralela de equações com o MPI;
5. comparação de desempenho da solução paralela com a solução sequencial.

Capítulo 2

Computação de Alto Desempenho

A computação de alto desempenho (*High-performance computing* ou HPC) é o nicho de estudo, tanto da ciência quanto da engenharia, que busca aumentar a performance computacional pelo aprimoramento do *software* e/ou do *hardware*. No primeiro caso, a performance é melhorada através do refinamento de algoritmos e compiladores, de forma que sejam utilizadas todas as capacidades que o *hardware* dispõe. Já no segundo caso, o desempenho é buscado pelo aperfeiçoamento dos componentes eletrônicos, melhorando dispositivos ou formando novos *designs* de arquiteturas computacionais.

Desde a utilização dos transístores nos circuitos eletrônicos, o aumento da velocidade de processamento dos computadores passou a seguir, aproximadamente, a escala linear da lei de Moore. No entanto, apesar da tecnologia estar sendo constantemente aprimorada, a demanda por maior capacidade computacional incentivou a criação de computadores paralelos e supercomputadores através da tecnologia LAN (*Local Area Network*), na década de 70 (Tanenbaum e Austin, 2013). Nesta época, trabalhar no desenvolvimento de novas arquiteturas e redes de computadores era muito custoso. Entretanto, os ganhos por possuir uma tecnologia computacional em seu estado da arte possibilitou avanços exclusivos.

Nesse contexto que foi lançado o projeto de HPC na DARPA (*Defense Advance Research Projects Agency*), agência federal de defesa americana, em conjunto com grandes empresas como IBM, Cray, HP, SGI e Sun. Segundo Dongarra et al. (2008),

esta iniciativa foi formada com o objetivo de prevenir surpresas tecnológicas de outros países, como o lançamento do satélite Sputnik em 1957, e, da mesma forma, criar tecnologias imprevisíveis para superar os adversários americanos. Áreas de interesse como previsão oceânica, análise de aeronaves, navios e estruturas, biotecnologia, design de armas, criptoanálise, sistemas de inteligência/vigilância/reconhecimento são algumas das pesquisas que o projeto visava impulsionar (Dongarra et al., 2008).

A partir da década de 90, os computadores paralelos começaram a ser artigos mais acessíveis através dos *clusters*, viabilizados pelo preço cada vez mais baixo dos computadores pessoais. Nesta época, o custo *vs* performance dos microprocessadores avançaram juntamente com as redes de alta velocidade. Assim, uma aproximação efetiva ocorreu entre a computação paralela e os pesquisadores. Contudo, segundo Dongarra et al. (2008), o aparecimento dos *clusters* causou o enfraquecimento da indústria de supercomputadores, diminuindo o investimento das empresas americanas no desevolvimento desta tecnologia custosa, uma situação que poderia enfraquecer tecnologicamente o país. Por isso, a participação na pesquisa e desenvolvimento do governo americano em supercomputadores continuou imprescindível.

Como pode ser visto na Figura 2.1, após o início do século XXI, os processadores tiveram taxas de ganho de performance cada vez menores. Este fato impulsionou fortemente a produção e uso de microprocessadores *multicore*. Assim, nessa década, computadores paralelos tornaram-se os computadores de uso geral através dos *chips* multiprocessados, possibilitando o desenvolvimento da computação paralela em programas de uso cotidiano.

Por último, ao final dos anos 2000, o processamento de uso geral em placas gráficas (*General Purpose Graphics Processing Unit* ou GPGPU) ganhou a atenção dos programadores, em especial devido a uma época de pesquisa e desenvolvimento extremamente interessada em *machine learning*. O grande poder computacional paralelo proporcionado pelas centenas de núcleos das placas gráficas impulsionou o desenvolvimento de uma variedade de áreas computacionalmente intensas. A partir

de então, o processamento vetorial, que somente eram realidade em supercomputadores, passaram a existir em versões menores e de alcance geral.

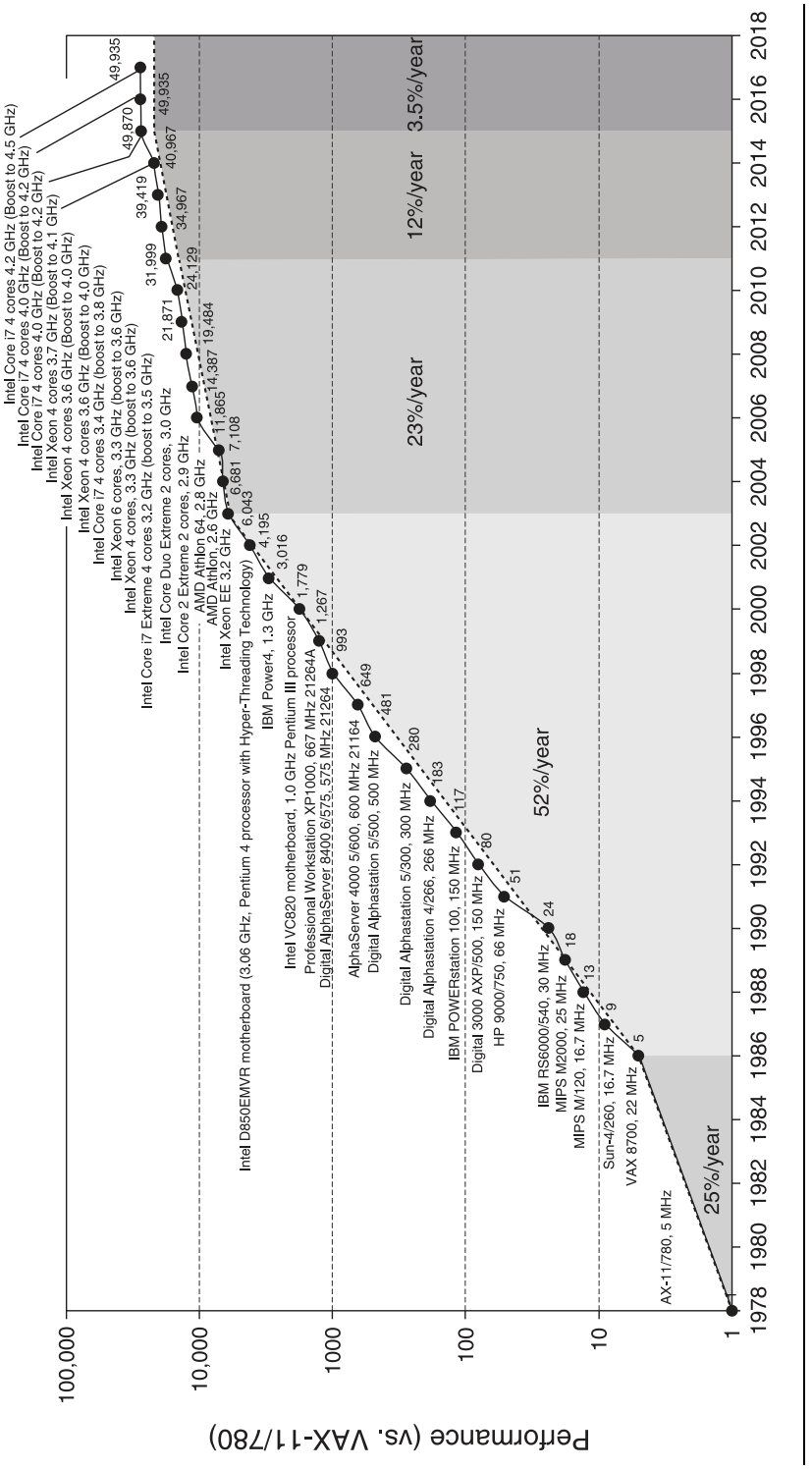


Figura 2.1: Aumento da performance de processadores nos últimos 40 anos (Autores: Hennessy e Patterson, 2017)

2.1 Computação Paralela

Foster (2003) define que um computador paralelo é um conjunto de máquinas independentes que cooperam para resolver um problema, uma definição tão ampla que permite incluir categorias como computadores massivamente paralelos, redes de computadores, multiprocessadores e sistemas embarcados. Apesar desta abrangência, cada tipo de computador paralelo possui sua singularidade, que deve ser explorada de forma a alcançar máxima eficiência. Assim, discutem-se algumas definições base para o adequado desenvolvimento de programas paralelos.

2.1.1 Tipos de Paralelismo

Muitas são as formas de classificar "paralelismo", sendo possível criar divisões e definições que muitas vezes sobrepõem-se, complementam-se ou divergem. Hennessy e Patterson (2017) começam expondo uma subdivisão para o paralelismo de programas e outra para o paralelismo de *hardware*. A seguir são descritos os dois tipos possíveis de paralelismo em aplicações.

(a) **Paralelismo de Dados (*Data-level Parallelism* ou DLP)**: manifesta-se quando existem muitos dados que podem ser manejados ao mesmo tempo utilizando as mesmas instruções;

(b) **Paralelismo de Tarefas (*Task-level Parallelism* ou TLP)**: manifesta-se quando existem muitas tarefas que podem ser executadas ao mesmo tempo devido a independência entre si;

Já as quatro formas possíveis de explorar paralelismo em *hardware*, segundo os autores, estão listadas nos itens abaixo (Hennessy e Patterson, 2017).

(a) **Paralelismo de Instruções (*Instruction-Level Paralelism* ou ILP)**: este tipo de paralelismo é obtido através da execução simultânea de múltiplas instruções provenientes de um programa. A concorrência destas somente é

possível porque para a conclusão de cada instrução são realizadas várias operações independentes, o chamado *pipelining*. Assim, várias instruções podem estar, de certa forma, simultanemanete em andamento desde que estejam em diferentes etapas nas operações do *pipeline*. Conforme Hennessy e Patterson (2017), o paralelismo de instruções pode ser explorado dinamicamente, através do *hardware*, ou estaticamente, por meio de compiladores. Desta forma, o ILP é presente em todos os computadores atuais através do ILP dinâmico, que ocorre na detecção e gerenciamento de operações que podem ser executadas paralelamente, e o ILP estático decorre na execução de um *software* devidamente compilado.

(b) **Processamento Vetorial**: esta categoria explora o paralelismo de dados, ou seja, efetua simultaneamente uma instrução em um conjunto de dados. Por isso, este tipo de processamento é muito conveniente em aplicações de computação intensiva e repetitiva, como operações em matrizes e vetores. Já que a tendência dos computadores atuais é de conter diferentes unidades de processamento especializadas, é comum haver unidades de processamento vetorial nos processadores de uso geral. Contudo, as unidades de processamento gráfico (*Graphics Processing Unit* ou GPU) são atualmente o exemplo mais representativo desta categoria.

(c) **Paralelismo de *Threads***: enquanto um computador está trabalhando, várias atividades estão sendo executadas e requisitadas pelo usuário ao mesmo tempo. Porém, quando existem mais tarefas que processadores, não é possível uma execução simultânea, sendo necessário alternar o tempo de processamento. Claro que esse revezamento ocorre em um intervalo de tempo tão rápido que cria-se a sensação de simultaneidade. Contudo, toda vez que uma tarefa tem sua execução interrompida é necessário que sejam guardadas informações do seu estado atual e suas futuras instruções em espera. As *threads* são uma

forma de abstração criada para explicar o funcionamento desse planejamento e execução de programas nos computadores. Portanto, cada *thread* é uma pequena tarefa de um processo em execução, sendo gerenciada pelo Sistema Operacional (SO), que deve ser processada pela(s) CPU(s). Esta categoria pode desempenhar tanto o paralelismo de tarefas quanto o paralelismo de dados (Hennessy e Patterson, 2017). Isso, porque é possível que várias *threads* estejam trabalhando ao mesmo tempo em um conjunto de dados (DLP) ou em diferentes tipos de tarefas (TLP). Contudo, para que o DLP seja eficiente – principalemente em comparação ao processamento vetorial – as tarefas a serem paralelizadas devem ter uma certa complexidade. Caso contrário, a divisão do paralelismo entre as diversas *threads* pode custar mais que o ganho de tempo da própria paralelização (Hennessy e Patterson, 2017).

(d) **Paralelismo de Requisição**: Este paralelismo ocorre exclusivamente em servidores gigantescos formados por enormes redes de computadores para acesso remoto de serviços. Estes computadores devem atender as requisições feitas por milhões de usuários, sendo imprescindível a paralelização destas operações.

2.1.2 Arquiteturas Paralelas

Da seção anterior, percebe-se que é difícil falar de tipos de paralelismo sem entrar em detalhes da organização de computadores paralelos. Por isto, um ponto de partida bastante utilizado para agrupar as diferentes arquiteturas conforme suas características é a taxonomia de Flynn, desenvolvida em 1966 e 1972 por Michael Flynn (Flynn, 2011). Ao longo do desenvolvimento de computadores e sistemas paralelos, diversos projetos foram concebidos de forma a aumentar a performance utilizando diferentes recursos. Desta forma, conforme Flynn (2011) explica, sua taxonomia é uma proposta criada para elucidar os tipos de paralelismo possíveis tanto no *hardware*, por processos do sistema, quanto em uma aplicação. Para facilitar, Flynn não

criou uma divisão conforme os tipos de máquinas, mas classificou segundo o conceito de fluxo de instruções (*instruction stream*) e fluxo de dados (*data stream*). O termo *stream*, traduzido aqui por fluxo, refere-se a uma sequência de dados ou instruções operados pelo computador na execução de um processo. Dessa forma, pode-se dizer que uma sequência de instruções a ser executada pela CPU forma o Fluxo de Instruções e uma sequência de dados requerida para a execução das instruções forma o Fluxo de Dados. Portanto, relacionam-se quatro combinações possíveis.

(a) **Um Fluxo de Instrução e Um Fluxo de Dados (*Single Instruction Stream Single Data Stream* ou SISD)**: a categoria SISD são os computadores mais simples, pois possuem apenas um processador e somente podem realizar uma instrução por vez. Apesar da sua aparência sequencial, estas arquiteturas podem ser paralelas por causa do paralelismo de instruções. Nesta categoria estão os processadores escalares, superescalares e VLIW (*Very Long Instruction Word*).

(b) **Um Fluxo de Instrução e Múltiplos Fluxos de Dados (*Single Instruction Stream Multiple Data Stream* ou SIMD)**: os computadores de uma instrução e múltiplos dados são assim referidos devido ao seu design de uma única unidade de controle (UC) para várias unidades de lógica e aritmética (ULA) no mesmo processador, aproveitando basicamente o DLP. Desta forma, uma instrução repassada pela UC é aplicada em múltiplos dados pelas ULAs. Sua grande vantagem é que cada instrução de leitura na memória – uma operação bastante custosa – é realizada ao mesmo tempo para todos os dados. Exemplos de computadores com esse tipo de arquitetura são os computadores vetoriais, matriciais e, mais recentemente, as placas gráficas (GPU ou *manycore*).

(c) **Multiplos Fluxos de Instruções e Um Fluxo de Dados (*Multiple Instruction Stream Single Data Stream* ou MISD)**: a arquitetura MISD

teoricamente refere-se à classe de computadores que executam várias instruções em um único conjunto de dados. Contudo, não existe um consenso quanto à sua existência (Tanenbaum e Austin, 2013), podendo ser considerada como uma arquitetura impossível de ser produzida. Entretanto, Flynn (2011) argumenta que as primeiras tentativas de criação de um processador vetorial eram um tipo de arquitetura MISD, assim como arquiteturas sistólicas (*systolic array*), arquiteturas de *dataflow* (*dataflow architecture*) e antigas placas gráficas.

(d) **Múltiplos Fluxos de Instruções e Múltiplos Fluxos de Dados (*Multiple Instruction Stream Multiple Data Stream* ou MIMD)**: os computadores MIMD devem possuir pelo menos dois processadores completos (uma unidades de controle para cada unidade de lógica e aritmética) que comunicam-se por alguma forma de conexão (Flynn, 2011). Como consequência, cada processador pode executar independentemente tarefas distintas. Porém, quando ocorre comunicação entre os processadores, é obrigatoria uma sincronização, causando um certo custo de tempo. Nesta categoria basicamente são classificados os *multicores*, multiprocessadores e multicomputadores.

Duncan (1992), contudo, criou outra taxonomia de computadores paralelos. Em seu trabalho, ele discorda que computadores com apenas paralelismo de baixo nível deveriam ser incluídos na classificação, pois, senão, a maioria dos computadores poderia ser considerados paralelos. Para o autor, paralelismo de baixo nível seriam funcionalidades como o *pipelining* de instruções e unidades de processamento especializadas, como de I/O (*Input/Output*) ou gráfico. Também, ele destaca a falta de classificação apropriada dos computadores vetoriais que, em sua visão, não podem ser classificados como SIMD, pois não executam-se exatamente as mesmas instruções nos *pipelines*, e, também, não existe uma autonomia assíncrona de tarefas como a categoria MIMD. Como a taxonomia de Duncan aprofunda-se muito na constituição

dos diferentes tipos de computadores paralelos e suas especifidades, fugindo do propósito deste capítulo, decidiu-se que já é suficiente para o entendimento do escopo deste trabalho a taxonomia de Flynn e, por isso, não são explorados os aspectos da obra de Duncan (1992).

Por último, para exemplificar com mais detalhes as principais arquiteturas paralelas, Tanenbaum e Austin (2013) mostram os diferentes níveis de paralelismos presentes nos diversos componentes dos computadores.

2.1.2.1 Paralelismo de *Chip*

O *chip* é o menor componente que pode-se desenvolver o paralelismo, ou seja, criar mecanismos para que sejam realizadas mais funções ao mesmo tempo. Assim, são três formas de paralelizar as atividades realizadas no *chip* de um processador.

2.1.2.1.1 Paralelismo de Instrução

Como explicado brevemente na seção anterior, o ILP é obtido quando mais de uma instrução de um processo é executada ao mesmo tempo. Isto acontece pela antecipação das próximas instruções do fluxo nos demais estágios do *pipeline*, como ilustrado na Figura 2.2. A Figura 2.2a exemplifica um *pipeline* de 5 estágios e na Figura 2.2b mostra-se a execução deste *pipeline* ao longo de 9 ciclos. Percebe-se que a partir do segundo ciclo o *pipeline* começa a processar as instruções 1 e 2 nos estágios S1 e S2, respectivamente, e que no quinto ciclo o *pipeline* está completamente ocupado com as instruções de número 1 a 5. Estes tipos de processadores são chamados de escalares e classificam-se como arquitetura SISD, pois apenas um fluxo de instruções e um fluxo de dados, provenientes de um único processo, estão sendo operados pelo processador.

Para aumentar ainda mais o número de instruções por segundo, é possível adicionar *pipelines* ao processador, arquitetura essa conhecida como superescalar. Assim,

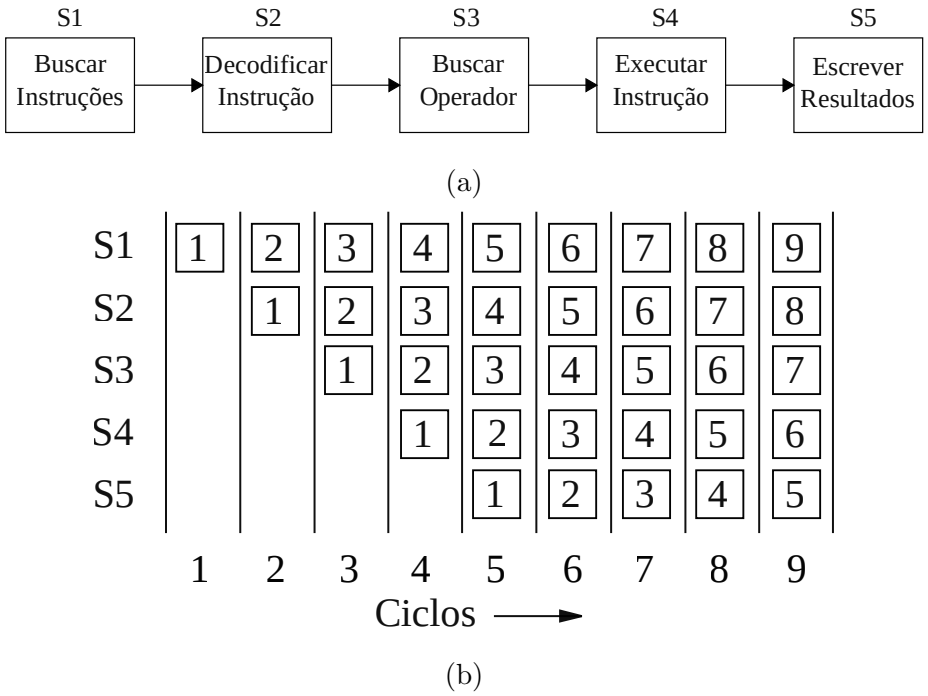


Figura 2.2: Paralelismo de instruções no pipeline: (a) exemplo de um *pipeline* com cinco operações e (b) execução das instruções pelo *pipeline* ao longo de nove ciclos (Adaptado de Tanenbaum e Austin, 2013)

a execução das instruções não fica limitada apenas ao número de estágios do *pipeline*. Estas classificam-se também como processadores SISD pois as instruções processadas ainda provém de apenas um único fluxo.

2.1.2.1.2 *Chip Multithreaded*

O progresso de velocidade dos processadores ocorreu mais rapidamente que a velocidade de transferência de dados da memória. Essa diferença, muitas vezes, ocasiona paradas no processamento em consequência da espera para carregar dados e instruções da memória. Para amenizar este tempo em suspenso, usa-se o *multithreading*, de forma que outras tarefas de outras *threads* sejam executadas durante a espera. Segundo Ungerer et al. (2003), os requisitos mínimos para um processador admitir o *multithreading* são: um mecanismo para diferenciar instruções de diferentes *threads*, um mecanismo que aciona a troca de *threads* e ter dois ou mais

contadores de programas independentes para guardar o contexto das diferentes *threads*. Entretanto, a existência de muitas *threads* também pode causar ineficiências devido à necessidade do gerenciamento destas pelo Sistema Operacional. Uma comparação visual pode ser feita entre a Figura 2.3a e 2.3c, com duas arquiteturas que comportam apenas uma *thread*, e as Figuras 2.3b, 2.3d e 2.3e, de arquiteturas que usam o *multithreading*.

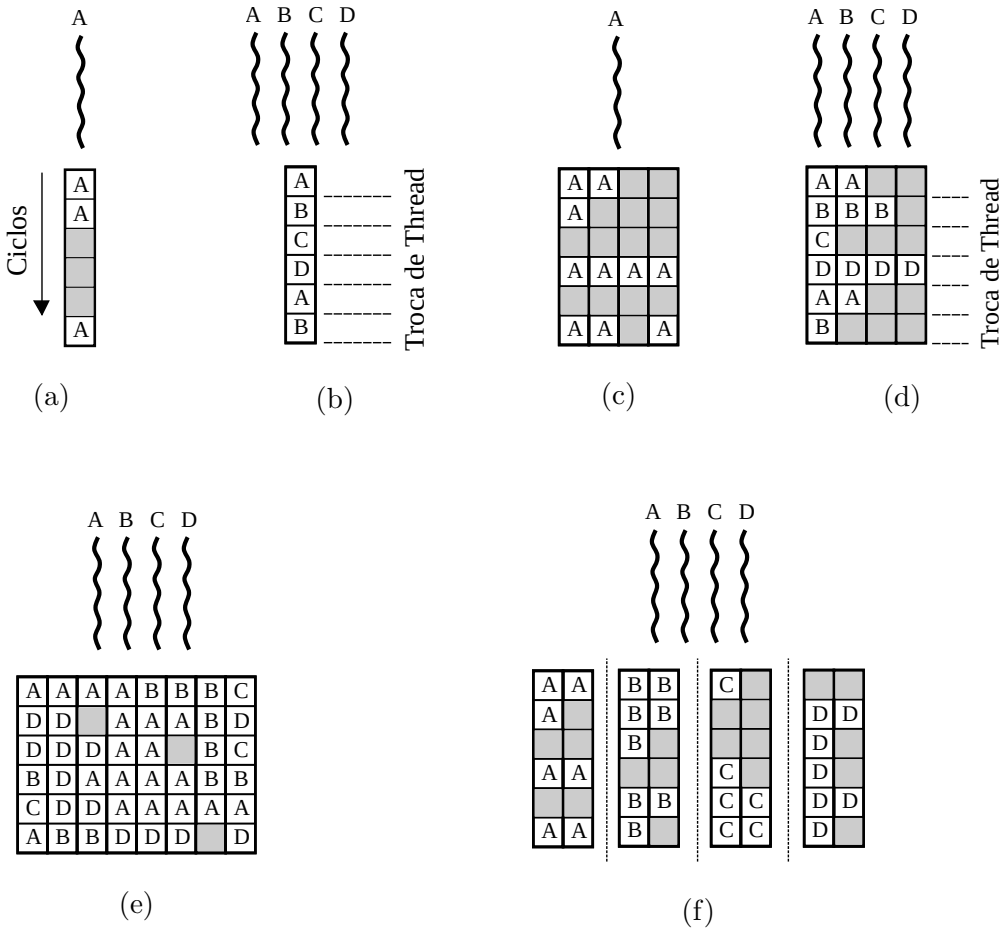


Figura 2.3: Ilustração do funcionamento da execução de *threads* em diferentes arquiteturas de computadores: (a) arquitetura escalar com uma *thread*, (b) arquitetura escalar com quatro *threads*, (c) arquitetura superescalar com uma *thread*, (d) arquitetura superescalar com quatro *threads* intercaladas, (e) arquitetura superescalar com quatro *threads* simultâneas e (f) multiprocessador superescalar em chip com 4 *threads*. (Adaptado de Ungerer et al., 2003)

Para finalizar este contexto, deve-se citar o *multithreading* simultâneo, chamado por *hyperthreading* pela fabricante de *chips* Intel. Observando a Figura 2.3d percebe-se a presença de alguma ociosidade nos *pipelines*, pois as *threads* estão sendo executadas intercaladamente para evitar problemas com dependência de dados. Todavia, quando os dados são independentes, pode-se aproveitar boa parte da capacidade ociosa se for permitida a execução simultânea de *threads*, como na Figura 2.3e. Consequentemente, é possível aumentar o número de instruções por segundo sem aumentar o número de componentes no *chip*.

2.1.2.1.3 *Chip* Multiprocessado

Quando o *chip* possui mais unidades de processamento, chamados *cores*, existe efetivamente a computação paralela de processos, e não a alternância de execução entre os processos ou o ILP. Na Figura 2.3d 2.3f pode-se contrastar a diferença entre um processador superescalar sem *multithreading* simultâneo e um multiprocessador superescalar. Já na comparação entre a Figura 2.3e 2.3f, os dois sistemas são bastante parecidos e, no fim, podem dar resultados semelhantes. Porém, o primeiro caso é um sistema de dificil escalabilidade, pois a detectção do ILP de muitas *threads* resulta em circuitos cada vez maiores e mais complexos.

Para melhorar ainda mais a capacidade dos *chips multicore* criaram-se hierarquias de memórias *cache*. Na Figura 2.4a está representado uma CPU (*Central Process Unit*) de dois *cores* com memória *cache* privada, L1, e uma memória *cache* compartilhada, L2. Quanto mais próxima do processador for a memória, menor e mais rápida ela será. Desse modo, as instruções e valores podem ser "bufferizadas" nos diferentes níveis de cache até que cheguem ao registrador do processador. Esse controle da memória, desempenhado pelo controle de memória (*Memory Chip Controller* ou MCC), pode ser feito por um *chip* especializado, localizado na placa mãe, ou pode estar diretamente integrado à CPU, como é o caso dos processadores *multicore*.

Ademais, a Figura 2.4 mostra que a memória RAM (*Random Access Memory*) também é compartilhada, significando que a comunicação entre dois processadores pode ser feita pela escrita e leitura tanto na memória *cache* como na RAM, sendo uma grande vantagem em termos de tempo de comunicação. Porém, como o uso do mesmo espaço de memória não é restrito, alterações indesejadas do conteúdo da memória podem ocorrer, causando resultados imprevisíveis se o *software* for produzido incorretamente.

2.1.2.2 Coprocessadores

Coprocessadores são processadores especializados em realizar uma tarefa específica e, dessa forma, liberam o processador principal de executar tarefas secundárias ao Sistema Operacional. Como visto anteriormente, Duncan (1992) argue que o conceito de computador paralelo não deve estender-se a coprocessadores. Entretanto, muitos aspectos já mudaram na computação desde seu trabalho, ficando impossível negar a importância da capacidade computacional de um processador gráfico aplicada à computação científica. Seguem dois exemplos de coprocessadores.

2.1.2.2.1 Processadores de Rede

Segundo Tanenbaum e Austin (2013), a evolução das tecnologias de rede permitiram a conexão quase da totalidade de computadores existentes, além de oferecer cada vez mais velocidade na transmissão de dados. Dependendo da conexão e do pacote de dados enviados, muitas operações precisam ser realizadas. Devido a isto, o uso de coprocessadores para lidar com o processamento das tarefas da conexão com uma rede beneficia claramente a eficiência do computador.

2.1.2.2.2 Processadores Gráficos

O processamento de uma imagem pode consistir em operações bastante simples, como *scaling* ou rotação, mas é imprescindível uma capacidade computacional

grande para lidar com tantos *pixels* sendo modificados ao mesmo tempo. Com a expansão do mercado de jogos, as unidades gráficas de processamento tiveram um grande investimento em sua tecnologia de fabricação. Todavia, inusitadamente, as GPUs atraíram o interesse da comunidade científica e de engenharia, devido sua eficiência em lidar com cálculos de matrizes e vetores, transformando as placas gráficas praticamente em unidades de processamento vetorial especializados. Devido ao sucesso desses empreendimentos, surgiram padrões de programação, como OpenCL (*Open Computing Language*), OpenACC (*Open Accelerators*) e CUDA (*Compute Unified Device Architecture*), que visam oferecer um modelo de programação de alto nível que facilite a aplicação de placas gráficas no uso científico. Em virtude disto, hoje, usam-se *clusters* de placas gráficas e algoritmos específicos de processamento vetorial.

2.1.2.3 Multiprocessadores de Memória Compartilhada

Ao passo que procuram-se novas formas de aumentar o número de *cores* nos *chips*, pode-se adicionar mais processadores a uma mesma placa mãe. Neste tipo de multiprocessamento a única memória dividida entre os processadores é a RAM, como pode ser visto na Figura 2.4b. Por isso, os mesmos problemas de compartilhar os mesmos recursos de memória em *chips* multicore também ocorrem aqui. Além disto, como todas as unidades de processamento dividem a mesma memória RAM, todos "enxergam" a mesma cópia do Sistema Operacional (Tanenbaum e Austin, 2013).

Uma dificuldade que ocorre tanto em multiprocessadores quanto *multicore* é o limite no número de CPUs que podem compartilhar uma mesma memória RAM. Inicialmente, as arquiteturas computacionais possuíam um único controle da memória e barramento, conhecidas como UMA (*Uniform Memory Access*). Esta configuração comportava o acesso uniforme da memória por todas as unidades de processamento, ou seja, todos os processadores levavam o mesmo intervalo de tempo para acessar

um valor da memória. Conforme Tanenbaum e Austin (2013) explicam, para poucos processadores, dois ou três, a divisão do barramento é manejável, contudo esta organização fica insustentável com 32 ou mais processadores.

Então surgiram as arquiteturas NUMA (*Non Uniform Memory Access*) que viabilizaram o uso de mais processadores sem que o acesso a memória fosse um impedimento. Para isto foram criadas as memórias locais e memórias remotas. As memórias locais possuem acesso rápido e mais direto a um determinado processador ou grupo de processadores. Já as memórias remotas são as que não estão diretamente ligadas com um processador. Assim, as diferentes memórias são gerenciadas por diferentes controladores de memória e barramentos. Portanto, quando um processador solicita um dado contido em uma memória remota, este deve requisitar ao controlador relativo a memória que possui as informações buscadas.

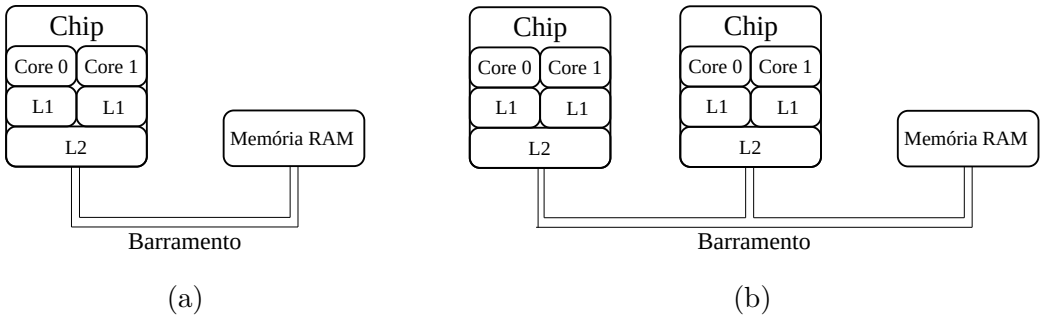


Figura 2.4: Diferença de arquitetura entre (a) um uniprocessador *multicore* e (b) um multiprocessador *multicore*

Apesar da nova arquitetura NUMA, ainda assim o aumento no número de processadores requer circuitos mais complexos e caros. Por este motivo é que torna-se tão difícil aumentar indefinidamente o número de unidades de processamento em uma arquitetura multiprocessada. Placas mãe multiprocessadas são um artigo quase que exclusivamente de servidores devido ao seu custo alto.

2.1.2.4 Multicomputadores

Quando não é mais possível adicionar processadores em uma mesma máquina, a única forma de aumentar a capacidade do sistema é conectando diferentes computadores a uma rede. Por consequência, cada computador, também chamado de nó, têm sua própria memória e Sistema Operacional para gerir seus recursos e processos, portanto, o nome computadores de memória distribuída. Assim, os vários níveis de paralelismo, já discutidos, podem ocorrer em multicomputadores. Na Figura 2.5 esquematiza-se um multicomputador de quatro nós, cada um tendo dois processadores *multicore* (Distributed Shared Memory ou DSM).

Como não existe uma memória compartilhada entre os nós, a comunicação entre computadores somente pode ocorrer pela rede. Devido a isto, a capacidade e configuração da rede são aspectos extremamente importantes para a eficiência destes sistemas. Para isso, existem duas alternativas, através da passagem de mensagens pela rede ou através do sistema de memória compartilhada distribuída (*Distributed Shared Memory* ou DSM).

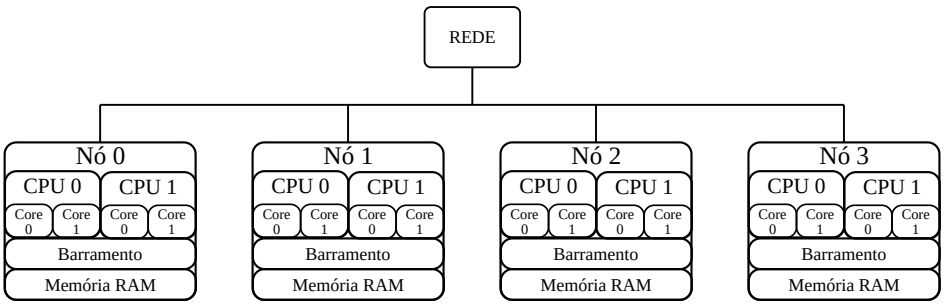


Figura 2.5: Exemplo de uma rede de computadores multiprocessados e *multicore*

A primeira opção é bastante simples, quando um computador necessita de informações que não estão contidas em sua memória, ele deve solicitar o envio destas ao computador que as possui. Desta forma, estas tarefas devem ser adicionadas ao programa, que acaba tendo sua complexidade aumentada, quando comparada com um computador de memória compartilhada.

Já a segunda alternativa é possível ao simular um ambiente de memória compartilhada através do Sistema Operacional. Ao invés de mensagens serem passadas pela rede através da implementação do programa, o SO é que detecta quando as informações estão contidas em outra máquina e envia através da rede.

2.1.3 Padrões de Programação Paralela

A medida que os computadores paralelos foram disseminando-se, os desenvolvedores buscaram formas acessíveis de integrar suas funcionalidades nos *softwares*. Em computadores de memória compartilhada, a principal e mais vantajosa forma de paralelismo é através do *multithreading*. Hoje, devido a importância e aumento dos computadores *multicore*, algumas linguagens de programação já incorporaram funções para o *multithreading*, como é o caso do Java, simplificando a criação de programas paralelos. Nas linguagens nativas C e C++ não existe tipo de suporte integrado, mas a API OpenMP (*Open Multi-Processing*) é uma opção bastante utilizada, além do Pthread (*POSIX Threads*), que é exclusiva para os sistemas operacionais do padrão POSIX (*Portable Operating System Interface*).

No caso dos multicomputadores, para possibilitar que diversos componentes trabalhem de forma conjunta e transparente para o usuário, o comando do sistema distribuído é entregue aos *middlewares* (Kshemkalyani e Singhai, 2008). A estes é incubida a responsabilidade de ser uma camada de abstração entre o usuário, o Sistema Operacional e as unidades de entrada e saída. São exemplos: CORBA (*Common Object Request Broker Architecture*), DCOM (*Distributed Component Object Model*), RMI (*Remote Method Invocation*), Apache Spark, MPI (*Message Passing Interface*) e PVM (*Parallel Virtual Machine*).

Para a linguagem de programação Java, identificaram-se quatro alternativas para a comunicação entre diferentes máquinas.

(a) **RMI ou TCP/IP**: é possível utilizar as próprias bibliotecas da linguagem para a comunicação na rede. O *framework* RMI é um modelo servidor-cliente

e não possui uma implementação específica para comunicações coletivas. O mesmo problema ocorre também para o protocolo TCP/IP que, apesar de ser uma especificação bastante conhecida para a transmissão de dados, também não possui uma infraestrutura mínima para comunicação distribuída. Por causa disto, é responsabilidade do desenvolvedor a criação das conexões entre os computadores e dos métodos de comunicações coletivas.

(b) **Apache Spark**: é o *framework* mais utilizado para computação paralela na indústria *big data*, provavelmente por proporcionar um alto nível de abstração para as tarefas paralelas. No entanto, sabe-se que seu desempenho é significantemente inferior ao MPI, principalemnte em operações de computação intensiva, como mostrado nos trabalhos de Gittens et al. (2016), Reyes-Ortiz et al. (2015) e Jha et al. (2014).

(c) **MPI para Java**: existem implementações do padrão MPI em Java puro, como MPJ-Express (Carpenter et al., 2000), PCJ (M. Nowicki, 2012) e FastMPJ (Expósito et al., 2012), que evitam as desvantagens do uso de código nativo. Todas estas bibliotecas mostram bons resultados, no entanto, dentre essas três opções, apenas a PCJ está com desenvolvimento ativo, tendo atualizações nos últimos meses.

(d) **MPI nativo**: a interface entre Java e C possui grandes facilidades através do JNI (*Java Native Interface*). Como as comunidades que produzem o MPI para C, como OpenMPI e MPICH, são bastante ativas e disponibilizam atualizações e correções constantemente, a chamada nativa destas bibliotecas é uma boa alternativa. Além disto, existem diversas bibliotecas de solucionados paralelos escritas com o padrão MPI e disponibilizadas livremente em C. No caso da biblioteca OpenMPI, já é disponibilizado um *binding* JNI de seus métodos. Entretanto, deve-se considerar que usar funcionalidades nativas prejudicam o princípio de portabilidade da linguagem Java.

Em componentes mais específicos, como as placas gráficas, não existem, na linguagem Java, pacotes para o seu uso. Para isso deve-se utilizar bibliotecas nativas, como JOCL, JCUDA, JAVACL, que fazem, também, uso do JNI. No caso das linguagens C e C++ empregam-se as bibliotecas comentadas na seção 2.1.2.2.

2.1.3.1 MPI

No caso da computação científica, fica clara a inclinação pelo uso do padrão de passagem de mensagens. Isto ocorre devido a origem do padrão MPI, que iniciou-se pela união de pesquisadores e fabricantes de computadores para formular os princípios dos quais esse deveria seguir. Em Message Passing Interface Forum (2015) o grupo explica a visão geral do MPI (tradução nossa):

> "MPI (*Message-Passing Interface*) é uma *especificação de uma biblioteca de interface com passagem de mensagens*, sendo todas estas propriedades significantes. O MPI aborda primariamente o modelo de programação paralelo de passagem de mensagens, do qual dados são movimentados de um espaço de endereçamento de um processo para outro através de operações cooperativas. Extensões deste modelo clássico de passagem de mensagens são proporcionadas através de operações coletivas, operações remotas de memória, criação dinâmica de processos e I/O paralelo. MPI é uma *especificação*, não uma implementação; existem diversas implementações do MPI. [...] MPI não é uma linguagem, e todas operações MPI são expressadas em funções, subrotinas ou métodos, de acordo com a linguagem de implementação [...]. Este padrão foi definido através de um processo aberto à comunidade de vendedores de computadores paralelos, cientistas da computação e desenvolvedores de aplicações. [...]
>
> Um dos objetivos do MPI é garantir a portabilidade do código fonte. Isto é, um programa escrito utilizando MPI que cumpre os padrões relevantes da linguagem deve ser portável e não deve requerer qualquer

alteração do código quando utilizada em diferentes sistemas."

Portanto, todas as rotinas aprovadas pelo grupo seguem o mesmo padrão de chamada para que seja possível a reutilização do código independentemente da biblioteca disponível. Este ponto é extremamente importante quando considerada a utilização de *clusters*. Muitas vezes um usuário eventual de um *cluster* deve utilizar uma implementação MPI específica daquele computador, podendo ser diferente da empregada durante a criação do programa. Desta forma, um código escrito em MPI deve ter o mesmo resultado final quando utilizado em compiladores diferentes.

Ainda em Message Passing Interface Forum (2015), explica-se o que é um programa MPI (tradução nossa):

> "Um programa MPI consiste de vários processos autônomos, executando seu próprio código, da mesma forma que ocorre em um computador MIMD. Os códigos executados por todos processos não precisam ser idênticos. Estes processos comunicam-se através da chamada dos comunicadores do MPI. Tipicamente, cada processo tem sua execução em seu próprio espaço de endereçamento, ainda que implementações de memória compartilhada sejam possíveis"

O padrão MPI impõe que todos os processos devem primeiramente chamar o comando MPI_INIT, antes de utilizar um de seus métodos, pois este tem a função de inicializar o ambiente para as chamadas da biblioteca. Da mesma forma, quando não forem mais utilizadas rotinas MPI, deve-se finalizar o ambiente paralelo através do $MPI\_FINALIZE$. Apesar desta imposição, segundo Message Passing Interface Forum (2015), o requisito de portabilidade do código não define como um programa MPI deve ser inicializado nem quais configurações o usuário deve realizar para executar o programa. Portanto, é obrigação do usuário fazer a chamada de tantos programas quanto desejar em todos os computadores que pretender usar. Deste modo, muitas implementações passaram a disponibilizar o comando *mpirun* que,

além de facilitar a inicialização de todos os programas, também estabelece algumas configurações iniciais necessárias ao ambiente paralelo. Percebendo as vantagens de portabilidade que um comando de inicialização pode proporcionar, o grupo decidiu por adotar o seguinte padrão (Message Passing Interface Forum, 2015).

$$mpiexec \;\; \text{-}n \;\; \langle número\ de\ processadores \rangle \;\; \langle programa \rangle$$

No caso de serem vários programas com diferentes configurações deve-se realizar a inicialização através do comando abaixo.

$$mpiexec \;\; \text{-}n \;\; \langle número\ de\ processadores \rangle \;\; \langle programa\ 1 \rangle \;\; : \\ \text{-}n \;\; \langle número\ de\ processadores \rangle \;\; \langle programa\ 2 \rangle \; ...$$

Desta forma, o usuário deve informar de antemão o número de processos que deseja utilizar. Assim, tornou-se praticamente obrigatório (por questões de facilidade e não exigência da especificação) o lançamento dos programas pelo *mpiexec* (ou *mpirun*), sendo uma prática fortemente indicada pelos desenvolvedores destas ferramentas.

Por fim, para facilitar a identificação e manipulação de cada processo autônomo, estes são numerados através do identificador chamado de $rank$, que varia de 0 a $n - 1$. Da mesma forma, existe a variável $size$ para que cada programa conheça o número de processos que deve trabalhar em conjunto. Por isso, em casos que todos os processos executam apenas um único código, geralmente diferencia-se o fluxo de comandos de cada processo através do seu número de $rank$.

2.2 Métricas da Paralelização

Quando um programa tem suas tarefas executadas de forma paralela, geralmente pode-se validar a implementação analisando sua performance através de diferentes parâmetros – número de instruções executadas, tempo de execução, uso de memória,

transferência de dados, etc. – conforme for o objeto do estudo. A metodologia mais empregada para quantificar o efeito do aumento de processadores em um programa é o cálculo do *speedup*. Dado o tempo de execução do programa sequencial, ou seja, apenas um processador, T_s, e o tempo executado pelo programa paralelo com N processadores, T_N, o *speedup* é dado pela equação a seguir (Eager et al., 1989):

$$S(N) = \frac{T_s}{T_N} \tag{2.1}$$

Desta forma, esta equação tem como propósito verificar o ganho, ou perda, com a paralelização. Consequentemente, quando um *speedup* maior que 1 for alcançado, o programa paralelo apresenta ganhos de tempo de processamento. Da mesma forma, *speedups* entre 0 e 1 indicam que o tempo de processamento aumentou com a paralelização. O *speedup* idealizado pelos desenvolvedores deve ter o valor de $S(n) = n$, representando o caso em que a carga de processamento é perfeitamente dividida entre os processadores e o tempo reduzido proporcionalmente ao número de processadores. Na maioria dos casos o que ocorre é que a comunicação entre processos ou as porções do código que não foram paralelizadas acabam afetando negativamente esta medida.

Também, pode-se avaliar a eficiência da implementação fazendo uma média da utilização dos processadores, chamada eficiência, com a Equação 2.2 (Eager et al., 1989). É claro que esta equação somente faz sentido no caso de todos os processadores possuírem igual capacidade.

$$E(N) = \frac{S(N)}{N} \tag{2.2}$$

Apesar da simplicidade destas equações, a complexidade ocorre na obtenção destes tempos e o que eles efetivamente representam da implementação. Isso porque a ação de medir tempos, de certa forma, causa uma certa perturbação dos resultados, o chamado efeito observador. A forma mais direta de extrair a duração de uma aplicação é requisitando diretamente a medida de tempo do sistema. Também, é

possível a utilização de ferramentas de *profiling*, que fazem a coleta automática de dados do programa em um intervalo, regular ou não, de tempo. Em casos de programas que raramente são interrompidos, como em servidores, os *profilers* são mais adequados uma vez que anexam-se ao programa em execução. Entretanto, a interpretação das respostas destas ferramentas nem sempre é tão acurada, podendo haver grandes divergências de resultados de *profiler* para *profiler*, conforme é explicado em mais detalhes por Mytkowicz et al. (2010).

Para testar a performance de uma implementação geralmente fala-se de *benchmarking*, que, no caso da computação, refere-se ao exame de um algoritmo ou programa. Oaks (2014) divide o *benchmarking* de programas em três: *microbenchmarking*, *macrobenchmarking* e *mesobenchmarking*.

No *microbenchmarking* são coletadas as informações de uma implementação pequena e totalmente isolada, geralmente testando-se diferentes tipos de algoritmos ou configurações. Oaks (2014) comenta que, no caso da linguagem Java, existem poucas oportunidades que o *microbenchmarking* pode ser proveitoso. Isto ocorre devido ao compilador JIT (*Just In Time*), que define os otimizadores utilizados conforme o uso do código, e, por isso, não é possível prever se a otimização utilizada durante o *microbenchmarking* será a mesma ao executar um programa complexo.

O *macrobenchmarking* é a medida de performance da própria aplicação sendo executada. Segundo Oaks (2014), é a abordagem mais adequada e, à medida que um programa cresce, torna-se cada vez mais importante e difícil de obter. Ele explica que, quando se trata de um *software* escrito em Java, muitas otimizações da JVM (*Java Virtual Machine*) assumem que todos os recursos da máquina estão disponíveis e, por isto, a eficiência pode ser bastante diferente quando várias aplicações estão sendo executadas em um mesmo computador. Um exemplo é o GC (*garbage collector*) que utiliza 100% de todos os processadores de um computador quando um ciclo de limpeza de memória é realizado. Estes aspectos somente reforçam mais ainda o porquê do *microbenchmark* não ser suficiente para a avaliação de performance de

um programa.

Por último, o *mesobenchmarking* é uma modalidade que fica entre o *microbenchmark* e o *macrobenchmark*, ou seja, os módulos testados não são tão simples e pequenos mas também não é a totalidade do aplicativo. Este tipo de teste produz resultados com menos imprevistos que o *microbenchmarking* ao mesmo tempo que é mais fácil de aplicar que o *macrobenchmarking*.

2.2.1 Lei de Amdahl

Uma forma de estimar o potencial da paralelização em um programa, bastante utilizado ao longo dos anos, é a Lei de Amdahl (Amdahl, 1967). Os resultados de *speedup* são aferidos através da equação a seguir:

$$T_n = T_s \cdot \left(S + \frac{P}{N} \right) \tag{2.3}$$

Sendo S o percentual de código que permanece sendo executada em sequencial e P o percentual restante de código que foi paralelizado. Dessa forma, $S + P$ equivale a 100%, e é possível calcular o *speedup* teórico desta formula a partir de:

$$S(N) = \frac{T_s}{T_N} = \frac{T_1}{T_s \left(S + \frac{P}{N} \right)} = \frac{1}{S + \frac{P}{N}} \tag{2.4}$$

A consequência da lei pode ser vista através da Figura 2.6 evidenciada no trabalho de Shi (1996). Conforme o autor mostrou, no exemplo de um computador com 1024 processadores, o percentual de código sequencial reduz drasticamente o *speedup* máximo possível. Assim, para $S = 0$, o *speedup* seria ideal, $S(N) = N$, e, para programas com a parcela sequencial S muito grandes, o valor do *speedup* seria limitado pelo valor $S(N) = 1/S$. Assim, segundo Shi (1996), a Lei de Amdahl foi utilizada como um forte argumento contra os computadores massivamente paralelos, principalmente quando consideramos a afirmação de Amdahl (1967) de que há

uma considerável fração computacional sequencial nos programas que parece não ser propícia à paralelização.

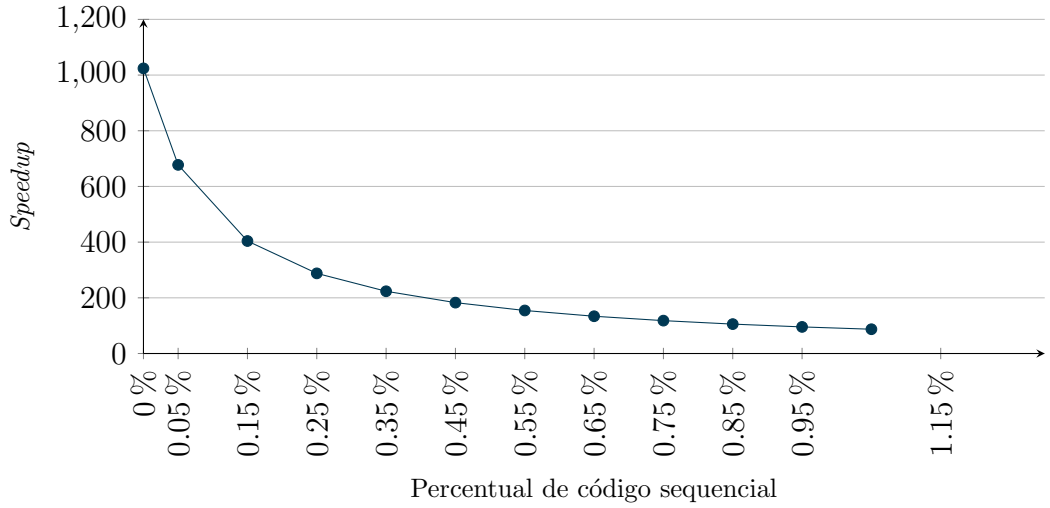


Figura 2.6: *Speedup* teórico da lei de Amdahl para um computador com 1024 processadores variando a percentagem de código sequencial

Mais tarde, o trabalho de Gustafson (1988) mostrou que foram obtidos valores de *speedup* muito superiores aos estimados pela lei de Amdahl. Ao rever a lei, ele propôs o chamado *Scaled Speedup*, que é dado pela seguinte fórmula:

$$Scaled\ Speedup = N + (1 - N) \cdot S \tag{2.5}$$

Contudo, Shi (1996) esclareceu que a Lei de Gustafsson é equivalente à Lei de Amdahl, havendo um engano de Gustafson (1988) ao aplicar a formulação deste. Portanto, apesar do engano, os experimentos de Gustafson (1988), que alcançaram valores propícios de *speedup*, permitiram constatar que não há pessimismos na Lei de Amdahl. Mas a verdade é que, para uma boa parcela de aplicações, a parcela sequencial é bastante pequena, sendo bastate possível alcançar o *speedup* ideal (Krishnaprasad, 2001).

Capítulo 3

Computação Paralela em Elementos Finitos

A aplicação da computação paralela em elementos finitos já é bastante antiga, tendo, provavelmente, seu desenvolvimento partindo de publicações como Noor e Hartley (1978) e Noor e Lambiotte (1979). Estes trabalhos realizaram testes no supercomputador CDC STAR-100, verificando a eficiência do processamento vetorial tanto na solução do sistema de equações quanto na montagem da matriz de rigidez. Já em Storaasli et al. (1982) foi realizada uma das primeiras implementações MIMD para elementos finitos com o *Finite Element Machine*, computador criado especialmente pelo Centro de Pesquisas de Langley da NASA para o cálculo de elementos finitos, conforme relata o artigo de Jordan (1978). Nesta época, houve uma grande concentração de pesquisas de computação paralela na área de dinâmica e aviação, devido a grande demanda computacional destes dois campos de estudo. Contudo, conforme Law (1986), as ideias de processamento e algoritmos paralelos já eram escopo de estudos lá na década de 60, antes mesmo da criação do primeiro computador paralelo.

Adeli et al. (1993) realizaram um resumo das etapas básicas necessárias em uma análise estrutural por elementos finitos.

1. Formação das matrizes de rigidez e vetores de forças dos elementos

2. Montagem do sistema de equações globais

3. Solução das equações

4. Cálculo das tensões e deformações dos elementos

Eles avaliam que estas são possíveis de paralelizar e, no caso da análise estrutural linear, destacam que a solução do sistema de equações pode ser a fase mais crítica em relação ao tempo. No caso de análises não lineares, eles apontam que os estágios de formação das matrizes de rigidez e vetores de forças dos elementos, montagem do sistema global e cálculo de tensões e deformações passam a ser também significativas no tempo de execução.

Já Noor (1997) relacionou as fases de uma análise estática em elementos finitos com os tipos de paralelização possíveis, mostrado na Tabela 3.1. Assim como Adeli et al. (1993), ele considera que todas as etapas são paralelizáveis, mas destacou a possibilidade de vetorização da matriz de rigidez dos elementos, solução de equações e o pós-processamento. Também, Noor (1997) incluiu a fase de entrada do problema, que pode ser igualmente paralelizada.

Tabela 3.1: Diferentes fases de uma análise estática estrutural em elementos finitos

Fase	Possibilidade de Paralelização
Entrada do problema	Pode ser paralelizado
Avaliação das características dos elementos	Fácil de ser paralelizado e pode ser vetorizado
Montagem	Pode ser paralelizado com cuidado e difícil de ser vetorizado
Condições de Contorno	Facilmente paralelizado
Soluções das equações	Importante de ser paralelizado e vetorizado
Pós-processamento	Pode ser paralelizado e vetorizado

(Adaptado de Noor, 1997)

Quanto à solução de um problema em elementos finitos e sua possível paralelização, existem diferentes procedimentos possíveis de se usar. Um exemplo são metodologias de Decomposição de Domínios (DD), que transformam a formulação de um problema em elementos finitos, de forma a torná-lo inerentemente paralelo. Estes procedimentos provavelmente foram desenvolvidos com uma finalidade original de diminuir o volume de dados de uma análise sem a perda de precisão. Consequentemente, o surgimento da computação paralela trouxe uma maior atenção nestes métodos devido a sua facilidade de aplicação. A seguir são discutidos alguns exemplos destes métodos.

3.1 Decomposição de Domínios

Toselli e Widlund (2005) explicam que a Decomposição de Domínios refere-se à divisão das equações diferenciais parciais (EPDs), ou suas aproximações, em problemas menores e acoplados, de forma a construir subdomínios que são pedaços do problema original. Esta decomposição pode dar-se no âmbito da continuidade do problema, onde são utilizados diferentes modelos físicos em diferentes regiões, no âmbito da discretização do problema, cujas metodologias de aproximação podem diferir-se conforme a região, ou no âmbito da solução dos sistemas algébricos oriundos da aproximação das equações diferenciais parciais. Muitas são as variações de DD, entretando estas são baseadas em duas principais linhas, o Método de Shwarz e o Método Complementar de Schur, vistos a seguir.

3.1.1 O Método de Schwarz

Conforme Toselli e Widlund (2005), a origem dos atuais métodos de decomposição de domínios é atribuída ao trabalho de Schwarz sobre dois subdomínios sobrepostos. Até então, existiam soluções fechadas para a resolução da equação de Laplace $\Delta u = 0$ em domínios círculares e retangulares, mas não de outras formas

arbitrárias. Schwarz, então, propôs uma solução numérica iterativa para uma geometria composta. O problema foi construído utilizando dois subdominios sobrepostos, Ω_1 e Ω_2, que formam o domínio $\Omega := \Omega_1 \cup \Omega_2$, com destaque nas interfaces Γ_1 e Γ_2, conforme mostrado na Figura 3.1a. Em uma aplicação de elementos finitos, a solução do problema é aproximada pelo sistema de equações lineares (Toselli e Widlund, 2005):

$$\begin{cases} Au = f, & \text{para } u \in \Omega \\ u = g, & \text{para } u \in \partial\Omega \end{cases} \tag{3.1}$$

Onde:

A é a matriz de rigidez;

u é o vetor de deslocamentos nodais;

f é o vetor de forças nodais;

g são as condições de contorno de deslocamentos.

Schwarz mostrou que a solução para o domínio Ω pode ser calculada de forma iterativa, alternando a solução dos domínios Ω_1 e Ω_2. Para isto, dado uma suposição inicial u^0 em Ω, a iteração u^{n+1} é calculada resolvendo os dois passos a seguir (Toselli e Widlund, 2005) e elucidada na Figura 3.1b.

$$\begin{cases} A_1 u_1^{n+1/2} = f_1 & \text{em } u \in \Omega_1 \\ u_1^{n+1/2} = g, & \text{para } u_1 \in \partial\Omega_1 \setminus \Gamma_1 \\ u_1^{n+1/2} = u_2^{n-1} \mid \Gamma_1 & \text{para } u_1, u_2 \in \Gamma_1 \end{cases} \quad \begin{cases} A_2 u_2^{n+1} = f_2, & \text{em } u_2 \in \Omega_2 \\ u_2^{n+1} = g, & \text{para } u_1 \in \partial\Omega_2 \setminus \Gamma_2 \\ u_2^{n+1} = u_1^{n+1/2} \mid \Gamma_2 & \text{para } u_1, u_2 \in \Gamma_2 \end{cases} \tag{3.2}$$

Percebe-se que a metodologia original não pode ser paralelizada, pois é preciso calcular sequencialmente cada subdomínio. Contudo, pode-se facilmente gerar um Método Paralelo de Schwarz simplesmente utilizando o valor inicial u^0 para calcular todos os subdominios simultaneamente, sob a penalidade de afetar a convergência.

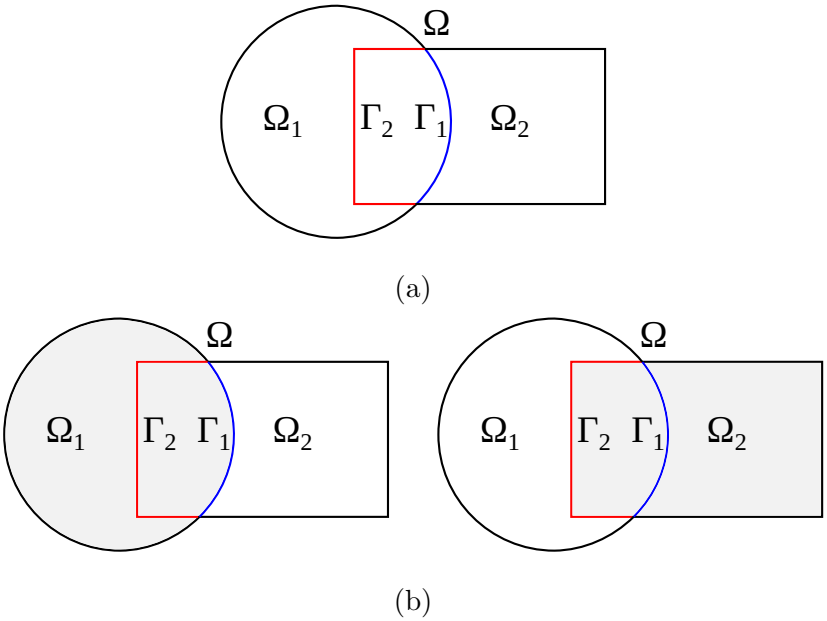


Figura 3.1: Decomposição de domínios com sobreposição – Método Alternante de Schwartz. (Adaptado de Toselli e Widlund, 2005)

3.1.2 O Método Complementar de Schur

A partir do método de Schwarz, foram derivadas as metodologias de Decomposição de Domínios sem sobreposição (*nonoverlapping*), conhecidos também por método Complementar de Schur. Na engenharia, esta meotodologia é chamada de subestruturação (*substructuring*) e foi apresentada por Przemieniecki (1963), engenheiro da Boeing que desenvolveu este recurso com a finalidade de diminuir a complexidade da análise de grandes estruturas aeroespaciais. Para um domínio Ω dividido em dois subdomínios Ω_1 e Ω_2 não sobrepostos, como da Figura 3.2, tem-se que (Toselli e Widlund, 2005):

$$\Omega = \Omega_1 \cup \Omega_2, \qquad \Omega_1 \cap \Omega_2 = \emptyset, \qquad \Gamma = \partial\Omega_1 \cap \partial\Omega_2 \tag{3.3}$$

Pela solução discreta em 3.1, os subdomínios podem ser separados da seguinte forma (Toselli e Widlund, 2005):

$$f^{(i)} = \begin{pmatrix} f_I^{(i)} \\ f_\Gamma^{(i)} \end{pmatrix}, \quad u^{(i)} = \begin{pmatrix} u_I^{(i)} \\ u_\Gamma^{(i)} \end{pmatrix}, \quad A^{(i)} = \begin{pmatrix} A_{II}^{(i)} & A_{I\Gamma}^{(i)} \\ A_{\Gamma I}^{(i)} & A_{\Gamma\Gamma}^{(i)} \end{pmatrix} \tag{3.4}$$

Onde o subscrito **I** representa os graus de liberdade internos ao subdomínio e **Γ** representa os graus de liberdades compartilhados na interface entre subdomínios. Os sobrescritos **i** representam a numeração dos subdomínios.

Uma representação gráfica desta divisão da matriz A em parcelas I (numerados do 1 ao 4) e Γ (numerados do 5 ao 10) é mostrada na Figura 3.3. Assim, é criada uma divisão entre os graus de liberdade internos aos domínios Ω_1 e Ω_2 e os graus de liberdade localizados em Γ. O problema apresentado na Figura 3.2 pode ser escrito como (Toselli e Widlund, 2005):

$$\begin{pmatrix} A_{II}^{(1)} & 0 & A_{I\Gamma}^{(1)} \\ 0 & A_{II}^{(2)} & A_{I\Gamma}^{(2)} \\ A_{\Gamma I}^{(1)} & A_{\Gamma I}^{(2)} & A_{\Gamma\Gamma}^{(1)} + A_{\Gamma\Gamma}^{(2)} \end{pmatrix} \cdot \begin{pmatrix} u_I^{(1)} \\ u_I^{(2)} \\ u_\Gamma^{(1)} + u_\Gamma^{(2)} \end{pmatrix} = \begin{pmatrix} f_I^{(1)} \\ f_I^{(2)} \\ f_\Gamma^{(1)} + f_\Gamma^{(2)} \end{pmatrix} \tag{3.5}$$

Logo, para resolver u_Γ é possível usar o artifício de decompor a matriz A em uma parcela triangular inferior e uma triangular superior, $A = LR$ (Toselli e Widlund, 2005).

$$A = LR = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ {A_{II}^{(1)}}^{-1} A_{\Gamma I}^{(1)} & {A_{II}^{(2)}}^{-1} A_{\Gamma I}^{(2)} & I \end{pmatrix} \cdot \begin{pmatrix} A_{II}^{(1)} & 0 & A_{I\Gamma}^{(1)} \\ 0 & A_{II}^{(2)} & A_{I\Gamma}^{(2)} \\ 0 & 0 & \boldsymbol{S} \end{pmatrix} \tag{3.6}$$

A matriz **S** é denominada complemento de Schur e é utilizada para calcular u_Γ a partir do seguinte sistema:

$$S^{(i)} \cdot u_\Gamma^{(i)} = g_\Gamma^{(i)} \tag{3.7}$$

ou,

$$\sum_{i=1}^{n} (S^{(i)} \cdot u_\Gamma^{(i)}) = \sum_{i=1}^{n} g_\Gamma^{(i)} \tag{3.8}$$

Onde:

$$S^{(i)} = A_{\Gamma\Gamma}^{(i)} - A_{\Gamma I}^{(i)} A_{II}^{(i)^{-1}} A_{\Gamma I}^{(i)}, \quad i = 1, 2 \tag{3.9}$$

$$g_{\Gamma}^{(i)} = f_{\Gamma}^{(i)} - A_{\Gamma I}^{(i)} A_{II}^{(i)^{-1}} f_{I}^{(i)} \tag{3.10}$$

Após resolver a Equação 3.8, os graus de liberdade internos u_I podem ser obtidos resolvendo a equação a seguir (Toselli e Widlund, 2005):

$$u_{I}^{(i)} = A_{II}^{(i)^{-1}} \left(f_{I}^{(i)} - A_{I\Gamma}^{(i)} u_{\Gamma}^{(i)} \right) \tag{3.11}$$

Portanto, para calcular os graus de liberdade u_Γ precisa-se a inversa de todas matrizes A_{II}, que podem ser computadas paralelamente. Os graus de liberdade u_I, que possuem uma relação direta com os valores de u_Γ, podem ser calculados, também, em paralelo. Infelizmente, é claro que a obtenção da inversa de matrizes é eventualmente uma operação eficientemente impeditiva. Todavia, existem alternativas numéricas e iterativas para evitar procedimentos diretos. Mais detalhes de algumas destas metodologias são descritas em Toselli e Widlund (2005).

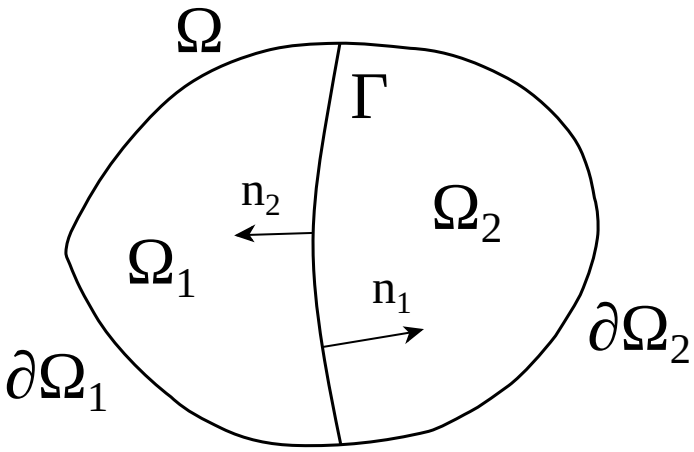


Figura 3.2: Decomposição de domínios sem sobreposição (Adaptado de Toselli e Widlund, 2005)

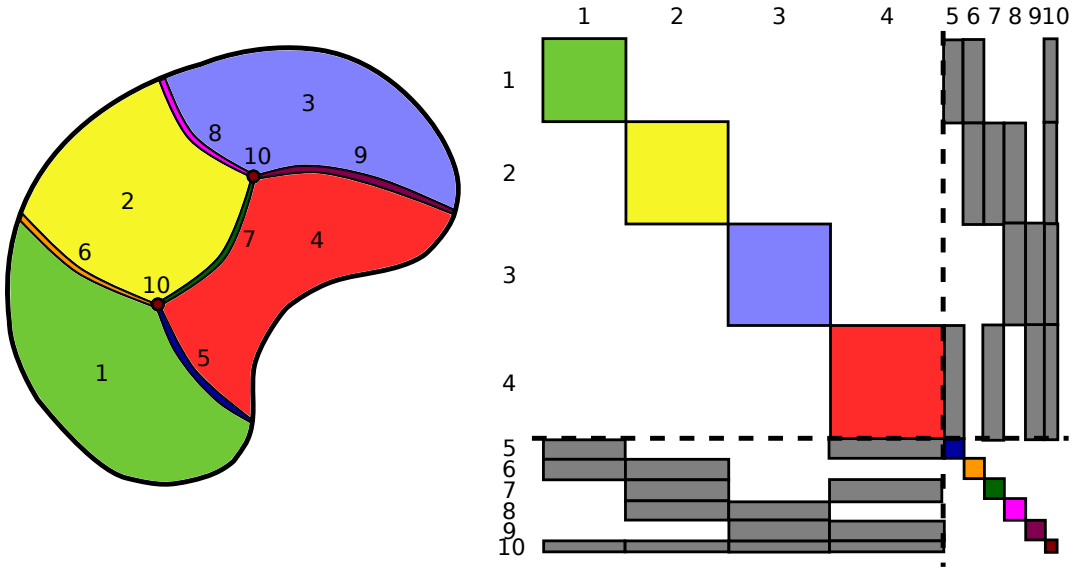


Figura 3.3: Ilustração da montagem da matriz de rigidez de uma decomposição de domínio dividia entre os graus de liberdade internos ao domínio e compartilhados entre dois ou mais domínios (Adaptado de Barth et al., 1998)

3.1.3 O Método Global-Local

O método Global-Local segue o mesmo princípio da Decomposição de Domínios, já que trata-se de um tipo de composição de malhas, e pode até ser classificado como tal. No entanto, sua metodologia de análise é diferente em comparação com Schwartz e Schur.

Considerando um domínio Ω contendo um subdomínio, com sobreposição, $\Omega_1 \subset \Omega$ de interface definida por Γ_1, ilustrado na Figura 3.4a. Calculando, pela aproximação da Equação 3.1, os valores de u do domínio Ω, denominado problema global, um novo problema pode ser formulado no domínio Ω_1, chamado local, ao serem aplicadas as respostas de u como condições de contorno na interface Γ_1, definido na equação abaixo.

$$\begin{cases} A_1 u_1 = f_1, & \text{para } u \in \Omega_1 \\ u_1 = u \mid \Gamma_1 & \text{para } u, u_1 \in \Gamma_1 \end{cases} \tag{3.12}$$

Quando a aproximação no domínio local, Ω_1, for mais refinada, naturalmente

suas respostas terão um erro numérico menor em relação ao domínio global Ω. Desse modo, pode-se melhorar a solução do domínio global utilizando as respostas de u_1 no seu cálculo. Se mais subdomínios são criados, é possível paralelizar a solução de todos domínios locais, já que há total independência entre estas tarefas.

No sistema INSANE já existem implementações da metodologia Global-Local em conjunto com o Método dos Elementos Finitos Generalizados, realizadas por Alves (2012), Malekan (2017) e dos Santos (2018), para acurar os resultados em problemas com concentração de tensões em trincas. Nestes trabalhos, além de empregarem formulações de enriquecimento na construção da aproximação, foram adotadas malhas mais refinadas no domínio local, como demostrado na Figura 3.4b.

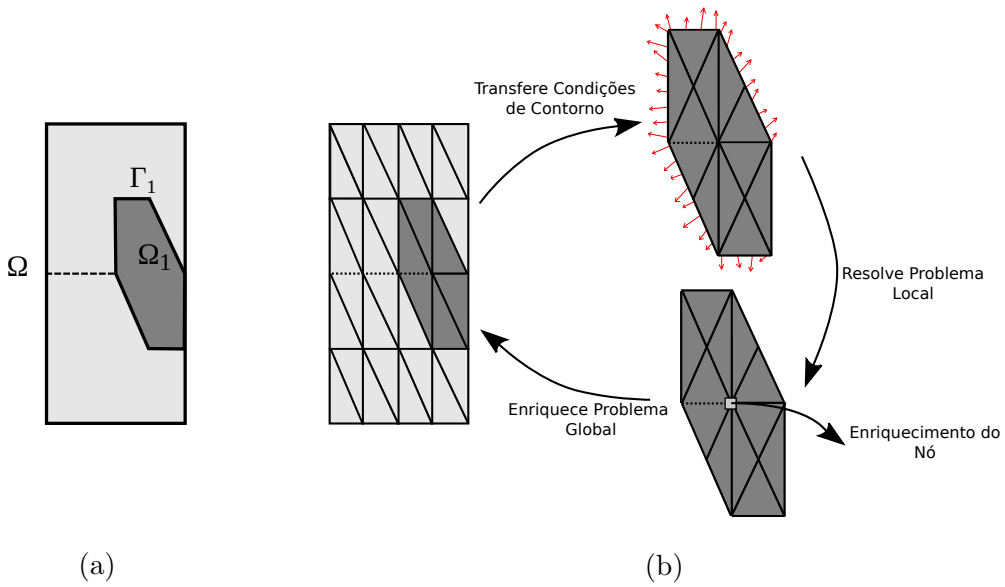


Figura 3.4: Esquema Global-Local de uma placa com trinca (Autor: Alves, 2012)

3.2 Sobre Solucionadores Paralelos

Naturalmente, quando não se deseja utilizar uma análise por Decomposição de Domínios, destacadas na seção anterior, pode-se paralelizar diretamente os solucionadores de equações. Ao longo destes anos de pesquisa em computação paralela

aplicada ao método de elementos finitos, houve uma grande concentração de esforços na investigação de métodos paralelos de solução de sistemas de equações. Isso ocorre devido à dois fatores: o alto custo de tempo que esta etapa representa e a dificuldade de criar algoritmos paralelos eficientes e escaláveis.

Os *solvers* paralelos podem ser utilizados tanto em memória compartilhada quanto em memória distribuída, contudo estes dois têm principios bastante diferentes, principalmente para alcançar uma boa eficiência. No caso da memória compartilhada, os solucionadores não precisam necessariamente de uma divisão da malha, já que o sistema de equações é compartilhado entre os processadores. Geralmente o que acontece é que estes paralelizam apenas as operações para a solução do sistema, e não o algoritmo em si.

Já os *solvers* distribuídos devem necessariamente dividir o sistema de equações, sendo, para isto, essencial a partição da malha. Diferentemente da Decomposição de Domínios, não existem problemas independentes que são resolvidos por solucionadores sequenciais funcionando paralelamente em cada pedaço da malha. Por isso, quando se usam solucionadores paralelos, o sistema de equações deve ser resolvido como um todo. Portanto, não é preciso o acoplamento das equações, pois estas ainda representam o problema original. Assim, algoritmos para a resolução de sistemas de equações originalmente sequenciais são alterados para funcionar em paralelo, resolvendo cada uma das partes deste sistema distribído concomitantemente e com a menor comunicação possível.

Várias foram as tentativas de paralelizar tanto *solvers* diretos quanto iterativos, no entanto, conforme King e Sonnad (1987), as metodologias diretas são difíceis de tornar eficientemente paralelas devido a sua forte característica serial, exibindo uma rápida saturação do *speedup* com o aumento de processadores. Exemplos da aplicação destes tipos de *solvers* podem ser encontrados em Farhat e Wilson (1987), Chien e Sun (1989), Chiang e Fulton (1990), que realizaram, em ordem, decomposição $\mathrm{LDL^T}$, eliminação de Gauss e decomposição de Cholesky paralelos.

Por outro lado, os métodos iterativos, apesar da fácil paralelização, são mais suscetíveis a problemas de convergência quando usados em matrizes mal-condicionadas. Porém, a criação dos pré-condicionadores trouxe maior confiança e rapidez nos resultados destas metodologias, aumentando sua popularidade. Destacam-se os métodos baseados nos subespaços de Krylov, como Gradientes Conjugados, Resíduo Mínimo Generalizado (GMRES) e Gradientes Biconjugados.

Felizmente, hoje já existem bibliotecas de *solvers* para sistemas distribuídos disponibilizadas de forma a serem facilmente integrados qualquer programa que necessite. Exemplos destas bibliotecas são Chameleon (Agullo et al., 2010), ScaLAPACK (Choi et al., 1992), MUMPS (Amestoy et al., 2000), PaStiX (Hénon et al., 2002), SuperLU (Li, 2005), DUNE (Blatt e Bastian, 2007), HYPRE (Falgout e Yang, 2002), pARMS (Li et al., 2003) e PETSc (Balay et al., 2019*b*) (Balay et al., 2019*a*) (Balay et al., 1997).

3.3 Sobre a Montagem Paralela do Sistema de Equações

A etapa de montagem dos sistemas de equações, poucas décadas atrás, era considerada bastante custosa, certamente devido à ausência de compiladores otimizados e pela tecnologia computacional até então existente. Por ser influenciada pelo tipo de análise e computador paralelo utilizados, esta fase continua tendo sua devida parcela de importância. Por exemplo, no caso de computadores com memória compartilhada, pode-se utilizar apenas uma única matriz de rigidez, suscitando problemas de sincronização nos processadores e perda de eficiência. Já em computadores de memória distribuída, a matriz deve obrigatoriamente ser dividida entre processadores. Nesses casos, é possível que se produza um excesso de comunicação.

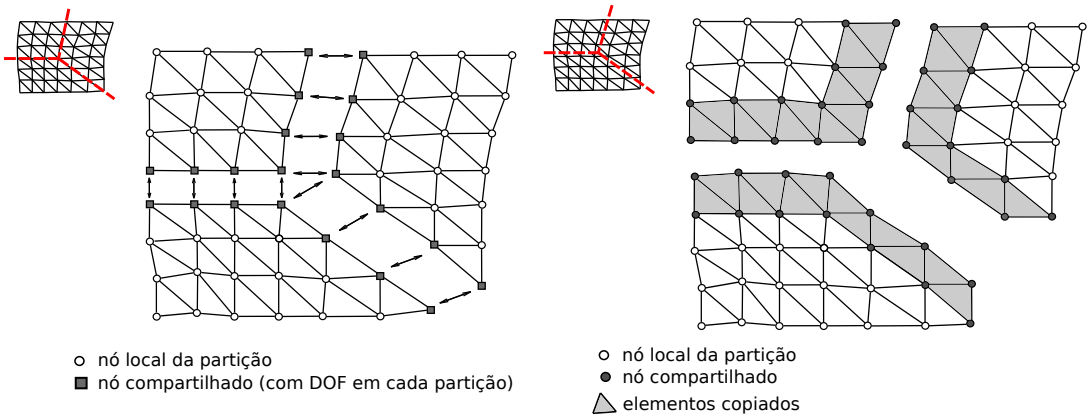
Em análises que não possuam uma divisão inerente do problema, como visto nas técnicas de Decomposição de Domínios, a solução mais simples de paralelização

da montagem do sistema de equações é a distribuição da lista de elementos entre processadores, como fizeram Chien e Sun (1989). Os autores realizaram testes para a montagem do sistema de equações em um computador de memória compartilhada. Eles propuseram duas soluções para o problema de sincronização da matriz global, uma através de semáforos e outra utilizando numeração diferenciada dos elementos. Na primeira solução, os semáforos foram criados para cada componente da matriz. Dessa forma, quando um processador acessasse uma posição da matriz para gravar sua parcela de rigidez calculada, outro processador era impossibilidade de requisitar o mesmo endereço de memória. Na segunda solução, eles realizaram uma numeração de forma que não coincidisse o acesso de uma mesma posição de memória por vários processos. Como resultado, mostrou-se que, nesta implementação, a sincronização dos processos usando travas de acesso à memória foi menos eficiente. Usando 12 processadores, a eficiência da primeira solução foi de apenas 68,25%, e da segunda foi de 91,76%. Portanto, a solução usando semáforos tende a ter uma maior degradação da eficiência com o aumento de processadores devido *locking* dos semáforos.

Como destacado por Farhat e Wilson (1987), a maioria dos trabalhos acabaram por optar por estratégias mais sofisticadas de particionamento de malhas. Neste trabalho, o autor propôs uma metodologia automática de geração de subdomínios. Para a criação destes, o algoritmo proposto utilizava uma lista de conectividade entre elementos e outra lista de conectividade de nós. Dessa forma, a repartição ocorria por um laço de repetição que renumerava os elementos conforme sua conectividade e o número de processadores. Contudo, hoje, predominam as heurísticas de particionamento de grafos com minimização de comunicação entre subdomínios e equilíbrio de carga de processamento entre computadores, que estão melhor detalhadas na seção 3.3.1.

Seguindo as metodologias de subdomínios, Krysl e Bittnar (2001) avaliam a aplicação de duas estratégias para divisão da malha, que eles nomeiam de *node-cut* e *element-cut*. Na primeira, a malha é partida nas arestas e faces dos elementos,

duplicando-se somente os nós entre os processadores, Figura 3.5a. No segundo caso, as arestas e faces dos elementos são partidos e, portanto, os elementos são duplicados entre os processadores, Figura 3.5b. As motivações que os levaram a essa pesquisa foram as diferentes vantagens e desvantagens dessas duas técnicas. A duplicação de nós é menos custosa tanto em termos de memória quanto, como explicam os autores, no custo computacional para o cálculo das forças internas de elementos e atualização do estado do material em análises não lineares. No entanto, a duplicação dos elementos permite que a etapa de montagem da matriz global seja feita sem nenhuma comunicação entre processadores. No exemplo do trabalho, a implementação de um problema de dinâmica em elementos finitos mostrou uma maior eficiência no uso do *node-cut*. Portanto, neste caso, o tempo de comunicação entre processadores foi inferior ao tempo despendido para a atualização das não linearidades do sistema.



(a) (b)

Figura 3.5: Tipos de cortes na malha (a) Estratégia *node-cut*, (b) estratégia *element-cut* (Adaptado de Krysl e Bittnar, 2001)

Outra questão importante da montagem do sistema de equações é o tamanho das matrizes formadas. No caso de análises estruturais por elementos finitos, a matriz de rigidez global contém predominantemente zeros, por isso sendo chamada de matriz esparsa. Consequentemente, o consumo de memória para a representação desta matriz acaba sendo majoritariamente em valores nulos. Conforme Law (1986), os

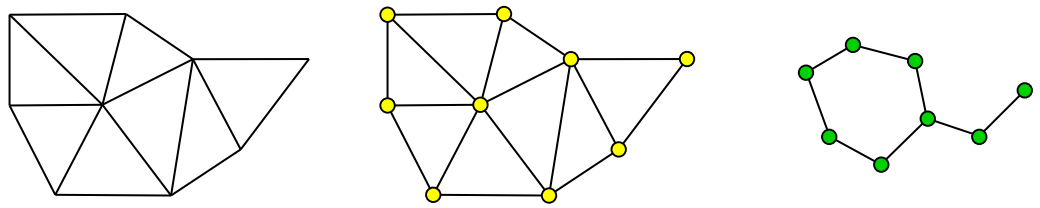
programas de elementos finitos idealizados durante as décadas de 60 e 70 foram elaboradas para computadores sequenciais com grandes limitações em tamanho de memória. Este fato acabou impulsionando a criação de técnicas que exijam menos espaço de memória. Um recurso desenvolvido devido a isto foi a criação de estruturas de dados para matrizes esparsas, que é amplamente utilizado na atualidade e discutido brevemente na seção 3.3.2.

Outro recurso importante para poupar consumo de memória é a montagem parcial da matriz de rigidez. Grandes economias de memória são alcançadas ao armazenar apenas a matriz de rigidez dos elementos e não a matriz de rigidez da estrutura. Alguns trabalhos chamam esta técnica de *element-by-element*, ou EBE, entretanto este termo também é utilizado para referir-se ao *loop* durante a montagem da matriz de rigidez global, realizado de elemento em elemento. Em Bording (1981), foi formulada uma solução paralela do Método Frontal (*Frontal Method*), uma espécie de eliminação de Gauss apresentada por Irons (1970), utilizando a montagem reduzida *element-by-element*. Bording (1981) afirma que, além da economia de memória, as operações para resolver o sistema não necessitam de comunicação entre processadores.

Em outro exemplo de trabalho, King e Sonnad (1987) utilizaram a técnica EBE juntamente com o Gradientes Conjugados Precondicionado. Como resultado, eles mostraram um *speed-up* de 3,88 com 4 processadores e uma eficiência de 97% em uma malha de 576 elementos. Com a tendência de trabalhos fazendo uso do paralelismo de dados das GPU's, os procedimentos EBE voltaram a ser estudados. Como é o caso do trabalho de Martínez-Frutos et al. (2015), cujos autores testaram uma implementação EBE de Gradientes Conjugados em dois tipos de placas gráficas. Os resultados foram comparados com um computador sequencial, e um *speedup* máximo de 14 foi alcançado.

3.3.1 Particionamento de Malhas

Hoje, o particionamento de malhas já é fortemente fundamentado nos métodos de divisão de grafos, com a construção destes a partir dos nós ou elementos da malha, como mostrado na Figura 3.6. Estes métodos facilitam a computação paralela pois buscam uma divisão equilibrada de nós/elementos entre processadores e a mínima comunicação entre subdomínios, ou seja, a menor fronteira para um certo número de subdomínios. Schloegel et al. (2003) classificaram as técnicas de divisão de grafos em geométricas, combinatórias, espectrais, multilevel e combinadas.



(a) (b) (c)

Figura 3.6: Concepção de um grafo a partir de uma malha: (a) malha original, (b) grafo formado através dos nós da malha, (c) grafo formado através dos elementos da malha (Autores: Schloegel et al., 2003)

3.3.1.1 Técnicas Geométricas

Estas metodologias fazem uso das coordenadas espaciais dos nós da malha, sem ponderar a conectividade, assim não são consideradas propriamente uma metodologia de particionamento de grafos (Schloegel et al., 2003). Como não minimizam-se as comunicações, há uma busca pela redução das fronteiras entre os subdomínios. São exemplos:

1. Bisseção Coordenada Recursiva (*Recursive Coordinate Bisection*)
2. Bisseção Inercial Recursiva (*Recursive Inertial Bisection*)
3. *Space-filling Curves*

3.3.1.2 Técnicas Combinatórias

Diferentemente das divisões geométricas, as metodologias combinatórias trabalham diretamente o particionamento de grafos através do agrupamento de vértices altamente conectados (Schloegel et al., 2003). Esta divisão dá-se independentemente da proximidade física e, por isso, não são necessárias coordenadas, somente conectividades da malha. Como exemplo, seguem:

1. *Levelized Nested Dissection*
2. Método de Kernighan-Lin
3. Método de Fiduccia-Mattheyses

3.3.1.3 Técnicas Espectrais

Os métodos espectrais são calculados através da obtenção de autovalores da matriz Laplaciana de um grafo. Por isso, tendem a ter um custo computacional e de tempo maior que outros métodos, mas apresentam resultados mais otimizados. São:

1. Bisseção Espectral
2. *K-way*
3. Método de Lanczos

3.3.1.4 Técnicas Multinível

As metodologias multinível são uma combinação de alguma técnica de partição, como as mostradas anteriormente, com o paradigma de três fases: condensamento, particionamento e refinamento multinível (Schloegel et al., 2003). Na primeira fase, diminui-se o grafo original, colapsando suas arestas de forma a produzir um novo grafo menor. Repete-se esta etapa até que exista um grafo com um número de

vértices iguais ao número de partições pretendidas. Então, particiona-se o novo grafo utilizando a estratégia escolhida. Por fim, reconstrói-se o grafo original refinando o particionamento à medida que as arestas vão sendo reincorporadas.

3.3.1.5 Técnicas Combinadas

Conforme Schloegel et al. (2003), todos os procedimentos de partição de grafos têm vantagens e desvantagens. Portanto, ao combinar de forma inteligente os diferentes procedimentos é possível maximizar as vantagens sem ocasionar todas as desvantagens.

3.3.1.6 Bibliotecas de Particionamento de Grafos

Como o particionamento de grafos é um problema bastante amplo e utilizado em diversas áreas que lidam com dados conectados que precisam ser divididos, existem diferentes bibliotecas especializadas nesta atividade. Algumas delas são: METIS (Karypis e Kumar, 1999), ParMETIS (Karypis e Kumar, 1998), SCOTCH (Pellegrini e Roman, 1996), PT-SCOTCH (Chevalier e Pellegrini, 2008), Zoltan (Devine et al., 2009) e Kahip (Sanders e Schulz, 2013).

3.3.2 Matrizes Esparsas

Existem algumas formas de expressar uma matriz esparsa sem a representação dos valores nulos. O procedimento mais simples é o formato triplo, que usa três vetores para representar as posições e valores de elementos não nulos, ou seja, um vetor com o número da linha, um vetor com o número da coluna, e um vetor com o valor desta posição na matriz, que são apresentados aqui por $\boldsymbol{rowPtr}$, $\boldsymbol{colPtr}$ e $\boldsymbol{x}$. Segundo Davis (2006), este formato, apesar de fácil representação e entendimento, dificulta a aplicação da maioria dos algoritmos de matrizes esparsas.

Também existem os formatos que priorizam as colunas ou linhas de uma matriz, como é o caso do formato CSC (*compressed sparse column*). Nele também existem

três vetores, mas representam-se os índices de colunas condensados, de forma a economizar mais memória e facilitar o acesso de colunas inteiras. Para uma matriz esparsa com um número de não zeros igual a $\boldsymbol{nnz}$, com $\boldsymbol{ncol}$ número de colunas e $\boldsymbol{nrow}$ número de linhas, o vetor $\boldsymbol{colPtr}$ passa a indicar apenas quantos valores não nulos existem em cada coluna. Portanto, para uma dada coluna $\boldsymbol{j}$, $\mathbf{0} \leq \boldsymbol{j} \leq (\boldsymbol{ncol} - \mathbf{1})$, calculam-se quantos elementos não nulos há nesta coluna a partir da subtração $\boldsymbol{colPtr}[\boldsymbol{j}+\mathbf{1}] - \boldsymbol{colPtr}[\boldsymbol{j}]$. Então, para saber quais posições de linhas da coluna $\boldsymbol{j}$ possuem valores não nulos, verifica-se o vetor $\boldsymbol{rowPtr}$ de $\boldsymbol{rowPtr}[\boldsymbol{colPtr}[\boldsymbol{j}]]$ até $\boldsymbol{rowPtr}[\boldsymbol{colPtr}[\boldsymbol{j}+\mathbf{1}]-\mathbf{1}]$.

Para conseguir o valor do elemento contido na linha $\boldsymbol{i}$ desta coluna $\boldsymbol{j}$ deve-se buscar o valor $\boldsymbol{i}$ neste intervalo do vetor $\boldsymbol{rowPtr}$ que, convenientemente, é escrito ordenamente de forma crescente. Assim, se a linha i for encontrada na posição $\boldsymbol{k}$, pode-se resgatar o elemento não nulo no vetor $\boldsymbol{x}[\boldsymbol{colPtr}[\boldsymbol{j}]+\boldsymbol{k}]$. Percebe-se a forte predisposição desta representação para trabalhar os elementos de uma coluna devido a facilidade com que estes são resgatados. Entretanto, esta destreza não ocorre para os elementos das linhas, sendo necessário toda vez realizar uma pesquisa no vetor de linhas.

O mesmo raciocínio explicado anteriormente pode ser aplicada para o formato CSR (*compressed sparse row*), bastando inverter a lógica dos vetores de linhas e colunas. A Figura 3.7 mostra uma matriz esparsa e os vetores resultantes para os formatos CSC e CSR.

Quando a matriz deve ser dividida entre vários processadores, passa-se a necessitar de duas numerações para as colunas e linhas. Isso porque deseja-se guardar apenas uma porção da matriz, e esta submatriz possui seu próprio tamanho de linhas e colunas. Desta forma, quando alguma operação deve ser realizada localmente na submatriz, converte-se a numeração da matriz global para a submatriz localmente armazenada.

$$A = \begin{pmatrix} 2 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 6 & 0 & 0 \end{pmatrix}$$

$$\text{CSC} \begin{cases} \text{colPtr}[] = \{0, 1, 2, 5, 6, 8\} \\ \text{rowPtr}[] = \{0, 3, 0, 1, 3, 1, 0, 2\} \\ \text{x}[] = \{2, 2, 3, 5, 6, 7, 4, 1\} \end{cases}$$

$$\text{CSR} \begin{cases} \text{rowPtr}[] = \{0, 3, 5, 6, 8\} \\ \text{colPtr}[] = \{0, 2, 4, 2, 3, 4, 1, 2\} \\ \text{x}[] = \{2, 3, 4, 5, 7, 1, 2, 6\} \end{cases}$$

Figura 3.7: Exemplo de uma matriz esparsa e suas representações em *Compressed Sparse Column* e *Compressed Sparse Row*

Capítulo 4

INSANE

O programa INSANE é um software de código livre e aberto para análises estruturais, criado e ampliado pelos professores e alunos do Departamento de Engenharia de Estruturas na Escola de Engenharia da Universidade Federal de Minas Gerais. Sua implementação é feita principalmente com a linguagem de programação Java, apesar de haver algumas funcionalidades nativas, utilizando o modelo de programação orientado a objetos. Uma visão geral e simplificada do funcionamento do núcleo numérico do INSANE pode ser representada através das quatro classes e inferfaces mostradas no diagrama da Figura 4.1.

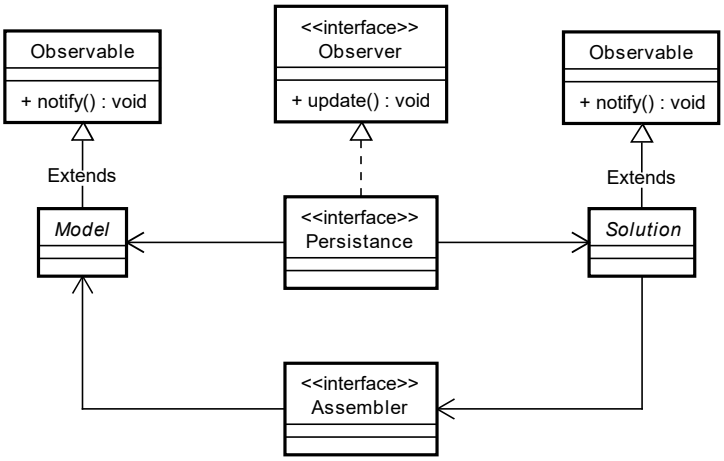


Figura 4.1: Organização do núcleo numérico do sistema INSANE (Autor: Fonseca, 2006)

A interface ***Persistence*** tem as funções de pré-processamento e pós-processamento das informações dadas pelo usuário e, portanto, faz a leitura dos dados de entrada e redige os resultados da análise . Para isto, o padrão *Observer* é implementado, de forma que esta interface é notificada quando os resultados estão prontos para, então, gerar o arquivo de saída formatado em XML (*Extensible Markup Language*).

A classe abstrata ***Model*** é a abstração designada para representar o modelo da análise, armazenando dados como pontos nodais, elementos, materias, etc. Suas classes filhas representam diferentes tipos de modelos, como, por exemplo, elementos finitos, elementos finitos generalizados e elementos de contorno. Deste modo, é possível generalizar a utilização da classe *Model* ao mesmo tempo que os supermétodos são reaproveitados.

A interface ***Assembler***, que já faz parte da etapa de processamento, tem como função solicitar propriedades do modelo de forma a construir a análise numérica, ou seja, criar matrizes e vetores. Para isto, esta interface possui a classe *Model* como sua variável de instância.

Por último, a classe abstrata ***Solution***, que também é objeto ativo da fase de processamento, executa as rotinas para solucionar a análise estrutural. Para tal, ela possui um objeto *Assembler* como sua variável para, assim, requisitar diretamente matrizes e vetores necessários para o problema que estiver solucionando.

Dado que este trabalho propõe-se a paralelizar a montagem e solução de problemas estáticos lineares do método dos elementos finitos, dedicou-se as seções a seguir para explicar funcionalidades das classes que são diretamente envolvidas nestas atividades, e que deverão ser adaptadas para a proposta deste trabalho.

4.1 *Assembler* no sistema INSANE

A construção de uma análise estrutural estática linear pode ser resumida na equação abaixo.

$$\underline{\underline{C}} \cdot \underline{x} = \underline{f} \tag{4.1}$$

Sendo $\underline{\underline{C}}$ a matriz de rigidez do modelo, $\underline{x}$ o vetor de deslocamentos nodais e $\underline{f}$ vetor de forças nos nós.

De forma a facilitar a aplicação computacional deste problema, separam-se os deslocamentos desconhecidos, identificados pelo subscrito u, dos deslocamentos prescritos, identificados pelo subscrito p. Assim, a matriz de rigidez e o vetor de forças devem ser, também, separados nestas parcelas para manter a relação linear, como mostrado a seguir.

$$\begin{pmatrix} \underline{\underline{C_{uu}}} & \underline{\underline{C_{up}}} \\ \underline{\underline{C_{pu}}} & \underline{\underline{C_{pp}}} \end{pmatrix} \cdot \begin{pmatrix} \underline{x_u} \\ \underline{x_p} \end{pmatrix} = \begin{pmatrix} \underline{N_p} \\ \underline{N_u} \end{pmatrix} + \begin{pmatrix} \underline{E_p} \\ \underline{E_u} \end{pmatrix} - \begin{pmatrix} \underline{F_p} \\ \underline{F_u} \end{pmatrix} \tag{4.2}$$

Neste sistema de equações, o vetor de forças é dividido em três parcelas diferentes, $\underline{N}$, $\underline{E}$ e $\underline{F}$, que são, respectivamente, o vetor de forças aplicadas aos nós, o vetor de forças equivalentes nodais às forças de superfície/corpo e o vetor de forças nodais equivalentes aos esforços internos.

A Figura 4.2 mostra o diagrama UML (*Unified Modeling Language*) da classe *Assembler* com algumas classes que a implementam. Também, mostram-se as chamadas dos principais métodos utilizados na etapa de montagem das matrizes e vetores do *software*: *getNp()*, *getNu()*, *getEp()*, *getEu()*, *getFp()*, *getCuu()*, *getCup()*, *getCpu()* e *getCpp()*.

A definição e numeração dos graus de liberdade desconhecidos e prescritos também é um trabalho dado às classes que implementam a interface *Assembler*, que ocorre através da chamada do método *numberEquations()*, executado quando chamada a rotina *init()*. Ao acionar *numberEquations()* é que são conhecidas as variáveis *sizeOfX*, número total de graus de liberdade, *sizeOfXu*, número de graus de liberdade desconhecidos, e *sizeOfXp*, o número de graus de liberdade prescritos. Estas têm como finalidade informar o tamanho das diferentes parcelas da matriz

de rigidez e vetores de força da Equação 4.2. Por exemplo, a matriz C_{pu} possui o número de linhas igual a *sizeOfXp* e o número de colunas igual a *sizeOfXu*.

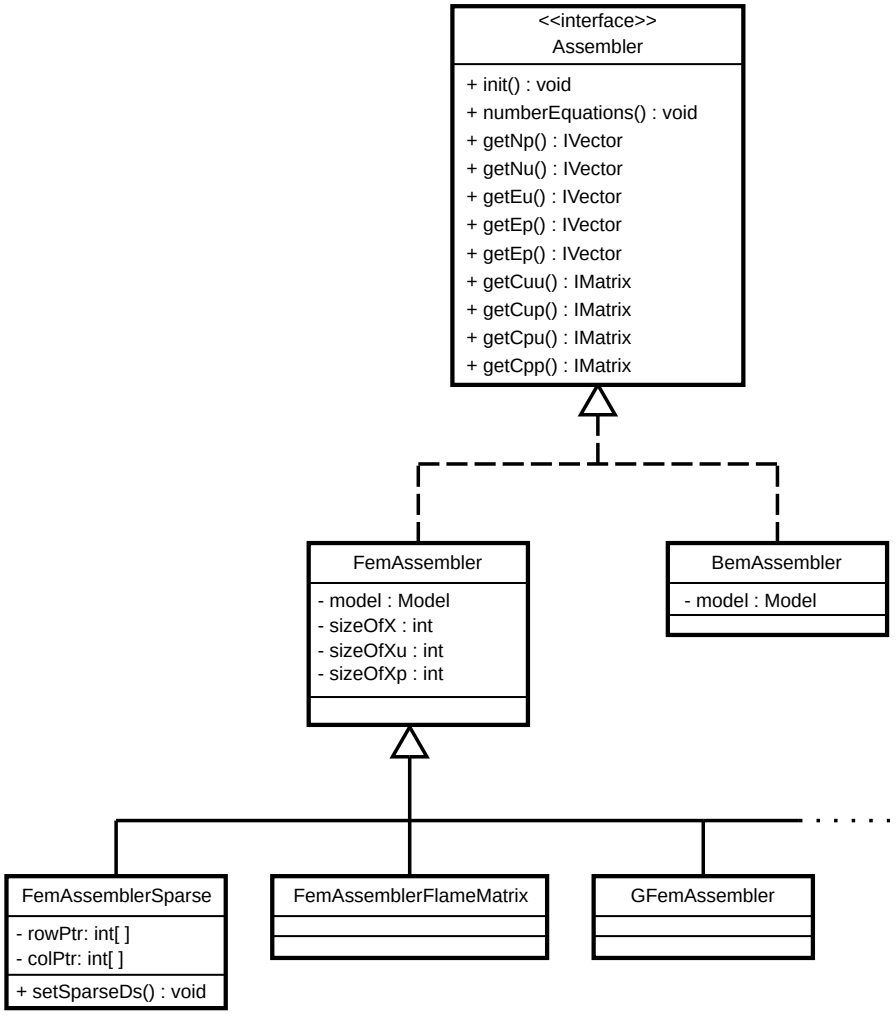


Figura 4.2: Diagrama UML da interface *Assembler* algumas das classes que a implementam

No caso de matrizes esparsas, o *software* já contém algumas implementações, como, por exemplo, a classe *FemAssemblerSparse*. Esta utiliza o formato *compressed sparse column* apenas para a montagem da parcela C_{uu} da matriz de rigidez, usando matrizes densas nas demais parcelas. Como esse tipo de matriz demanda uma estrutura de dados mais complexa, já não se é suficiente apenas a informação de número de linhas e colunas, como é a função das variáveis *sizeOfXu* e *sizeOfXp*.

Portanto, a criação desta matriz passa a ter uma correlação direta com as conectividades entre os elementos da malha, ou seja, com o modelo. Por esse motivo, são necessárias as variáveis *rowPtr*, referente ao vetor de linhas da matriz esparsa, e *colPtr*, referente ao vetor de colunas da matriz esparsa, como pode ser visto na Figura 4.2. Assim, a determinação dessas variáveis ocorre com o método *setSparseDs()*, que é chamado para avaliar os valores não nulos e suas posições na matriz esparsa.

Como não existe um encapsulamento adequado para a criação de matrizes, principalmente no caso de matrizes esparsas, faz-se necessária a criação de várias classes *Assembler* para acomodar diferentes tipos de matrizes, como é o caso, por exemplo, da classe *FemAssemblerFlameMatrix*, também mostrada na Figura 4.2.

4.2 *Solution* no sistema INSANE

Da classe *Solution*, destacam-se os métodos *setAssembler()* e *execute()*, mostrados na Figura 4.3. O primeiro tem destaque aqui pois é neste método que desencadeia-se o método *init()* do *Assembler*, chamando outras rotinas importantes para inicializar diversas variáveis necessárias para o andamento da análise, como, por exemplo, a numeração dos graus de liberdade. Já o método *execute()* tem a função de iniciar o processo de solução do problema.

No caso de uma implementação baseada em Decomposição de Domínios, como Schwartz ou Schur, esta é a classe que deve conter o processo de solução dos subdomínios, coordenando todas as trocas de informações. Contudo, se a paralelização ocorre no solucionador, não existe uma classe encarregada de realizar esta tarefa. Até então, todas as rotinas de solucionadores são guardadas na classe que representa matrizes, chamada *IMatrix*. A definição do *solver* no *software* é feita através do valor da variável *solverType*, determinada a partir do arquivo de entrada. A seleção do solucionador é feita, então, por uma estrutura de *if-else* em cada uma das classes

Solution, que chama o devido método da classe *IMatrix*.

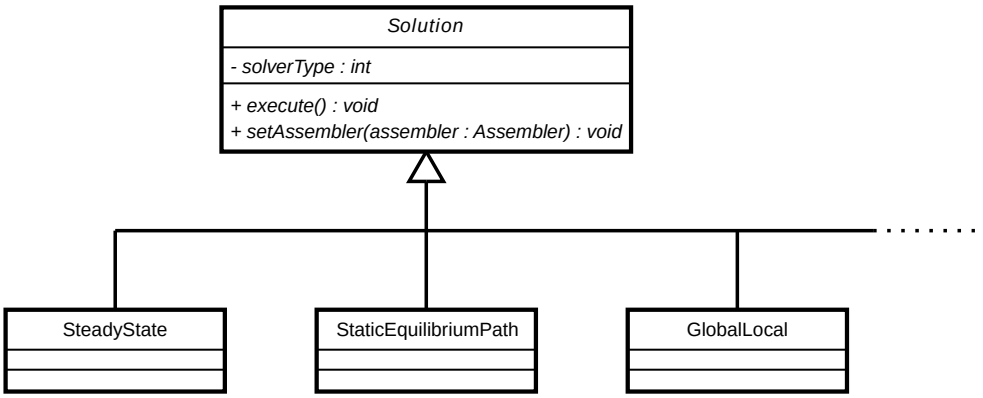


Figura 4.3: Diagrama UML da superclasse *Solution* e algumas de suas classes herdeiras

4.3 Bibliotecas Nativas

Já existem algumas bibliotecas nativas, produzidas por outros grupos de desenvolvedores, adotadas no programa através do padrão JNI (*Java Native Interface*). Assim como qualquer código nativo, seu uso tem implicações na questão de portabilidade do *software*. Contudo, estas são bastante necessárias para viabilizar análises que necessitam de alto desempenho, principalmente quanto aos *solvers*. Nesta seção, são discorridas as funções das bibliotecas usadas no sistema durante o processo de solução.

4.3.1 OpenBLAS

O padrão BLAS (*Basic Linear Algebra Subprograms*) é uma proposta para a criação de bibliotecas de alta performance com funções básicas de kernel para operações em vetores e matrizes (Blackford et al., 2002), facilitando a incorporação destas rotinas excepcionalmente eficientes em programas de computação científica. Geralmente, fabricantes de processadores disponibilizam sua própria implementação do padrão BLAS, dada a ligação direta entre a arquitetura do *chip* e o desempenho dos

programas. No entanto existem diversas implementações livres como OpenBLAS (Zhang et al., 2012), BLIS (Van-Zee e Van de Geijin, 2015), ATLAS (Whaley e Dongarra, 1998) e XBLAS (Li et al., 2008).

A versão OpenBLAS é uma implementação de código aberto que deu continuação no projeto GotoBLAS (Goto e Van de Geijn, 2008), que teve de ser interrompido. O projeto inicial tinha suas rotinas escritas em Assembly, pacientemente, à mão pelo criador do projeto, mostrando ganhos muito superiores aos de bibliotecas automatizadas como a ALTAS (Markoff, 2005). Estes incríveis ganhos em desempenho dão-se pelo uso eficiente dos diferentes níveis de memória *cache* e dos registradores para a movimentação dos dados. Hoje, o projeto conta com diversos códigos escritos em C e Assembly, além de ter incorporado, também, rotinas da biblioteca ATLAS (Zhang et al., 2012). É claro que, para que a biblioteca tenha eficiência no uso dessas funções de tão de baixo nível, é necessário que ela seja compilada tanto para o tipo de sistema operacional quanto para o tipo de arquitetura do computador em uso.

Apesar da OpenBLAS não ser diretamente chamada nos métodos de álgebra linear implementados no INSANE, ela é um requisito para performance de diversas outras bibliotecas utilizadas no sistema.

4.3.2 LAPACK

A LAPACK (*Linear Algebra Package*) é uma biblioteca que disponibiliza implementações de rotinas de solução de sistemas de equações e autovalores (Dongarra e Luszczek, 2011). Escrita em Fortran 77 e criada para substituir as bibliotecas EISPACK (Smith et al., 1976) e LINPACK (Dangorra et al., 1979), ela utiliza as funções do padrão BLAS para prover a performance de suas rotinas (Dongarra e Luszczek, 2011). Apesar de ser originalmente uma implementação de álgebra linear numérica, a LAPACK acabou tornando-se um padrão seguido inclusive por fabricantes de processadores. Além da LAPACK, surgiram bibliotecas com o objetivo de trazer as mesmas funções aplicadas em computadores paralelos de memória distribuída, como

exemplo ScaLAPACK (Choi et al., 1992) e PLAPACK (Alpatov et al., 1997).

Assim como a OpenBLAS, não existe uma implementação direta dos métodos da LAPACK no sistema INSANE, contudo, ela é necessária para o uso de outras bibliotecas.

4.3.3 LibFLAME

Esta biblioteca é derivada do projeto FLAME (*Formal Linear Algebra Methods Environment*), realizado colaborativamente entre a Universidade do Texas, em Austin, e a Universidade Jaume I, em Castellon, (Van Zee et al., 2009). Sua aplicação é principalmente direcionada a operações de álgebra linear com elementos (matrizes e vetores) densos, isto é, não esparsos. Fazendo uso das bibliotecas BLAS e, opcionalmente, LAPACK, do *multithreading* (através do OpenMP ou Pthreads) e de algoritmos desenvolvidos pelo grupo, esta biblioteca compromete-se com o alto desempenho e código livre para a comunidade científica.

Segundo Van Zee et al. (2009), é possível alcançar performances que superam a biblioteca LAPACK, isto devido a detecção de tarefas dependentes e, com isso, gerenciar as *threads* que são independentes. Ainda conforme Van Zee et al. (2009), a libFLAME difere das implementações BLAS e LAPACK pois são fornecidos algoritmos diversificados para cada operação, de forma que pode-se escolher o mais adequado conforme situação do usuário. Também, a biblioteca pode ser utilizada como *framework* para produzir novas implementações de álgebra linear.

Sua utilização no projeto é importante no Método dos Elementos de Contorno, pois este gera matrizes densas.

4.3.4 SuiteSparse: LIBUMFPACK

SuiteSparse é um pacote de algoritmos desenvolvidos para operações em matrizes esparsas e criada pelo professor Timothy A. Davis. Dentre os vários métodos

de solução de sistema de equações disponibilizados, adicionou-se ao repertório a biblioteca UMFPACK (*Unsymmetric MultiFrontal Package*) (Davis, 2004*a*) (Davis, 2004*b*). Segundo Davis (2004*a*), a UMFPACK é um *solver* direto de sistemas de equações lineares esparso e assimétrico escrito em C. Para isto, o algoritmo reordena colunas e linhas de forma a diminuir o *fill-in* da matriz, isto é, reduzir o preenchimento de elementos originalmente nulos na matriz e evitando o aumento de memória consumida. Após, multiplicam-se os *pivots* da matriz para diminuir o erro numérico da solução. Por fim, é feita a fatoração LU da matriz a partir do método multifrontal descrito em Davis (2004*b*).

Capítulo 5

Implementações

Reservou-se este capítulo para apresentar a construção da solução proposta e, principalmente, a linha de motivos e considerações que levaram o trabalho à sua forma final. Para isto, o trabalho foi iniciado com a implementação de uma metodologia inerentemente paralela, como é o caso da análise Global-Local, que, como citado anteriormente, já está incorporada no INSANE. Esta determinação foi realizada com o intuito de investigar imediatamente as ferramentas de computação paralela disponíveis, discutidas na seção 2.1.3. Em seguida, partiu-se para uma implementação paralela da análise estática linear do método dos elementos finitos, utilizando as informações adquiridas pelo teste inicial do Global-Local paralelo. Esta implementação acabou tornando-se o elemento principal deste trabalho.

5.1 Implementação Paralela da Estratégia de Solução Global-Local

De princípio, considerando os padrões de programação possíveis de utilizar em uma aplicação paralela, decidiu-se pelo uso de uma biblioteca MPI em Java. Esta determinação foi tomada pensando, principalmente, na portabilidade do programa,

que não seria afetada com a incorporação das funcionalidades desta implementação. Também queria-se testar a configuração *multicore* para o uso do *multithreading* e, desta forma, proporcionar a paralelização tanto para multicomputadores quanto multiprocessadores. Dentre as três opções apresentadas na seção 2.1.3, optou-se pela biblioteca MPJ-Express devido à disponibilidade de bons materiais de referência. Apesar do PCJ ser uma biblioteca mais atual e com atividades recentes de seus desenvolvedores, não foi possível utilizá-la, pois, no momento da elaboração do trabalho, esta biblioteca estava indisponível.

A principal dificuldade encontrada para a integração do padrão MPI no sistema INSANE, que é um *software* fortemente idealizado para execução sequencial, foi a inicialização do ambiente paralelo. Como explicado anteriormente na seção 2.1.3.1, o MPI é um conjunto de programas que devem ser iniciados, preferencialmente, pelo comando *mpirun* ou *mpiexec*. Portanto, para o *software* possibilitar o uso do MPI torna-se necessária a integração destes comandos com a inicialização do sistema INSANE, e, assim, vinculando, obrigatoriamente, o programa ao MPI. Como não era desejado a criação de um novo programa, isto é, queria-se manter as implementações sequencial e paralela juntas, esta imposição trouxe dois cenários possíveis para o trabalho.

(i) Modificar completamente o ambiente existente e amplamente consolidado do INSANE para incluir chamadas MPI;

(ii) Adicionar o MPI em uma aplicação a ser executada à parte apenas na solução do problema.

Como a primeira opção pareceu uma possibilidade complexa e precipitada para um assunto ainda não explorado pelo grupo de pesquisa, decidiu-se pela criação de um programa independente para a execução paralela da solução. Sendo assim, o programa principal, isto é, o INSANE, deve ler todas as informações do problema a ser resolvido e, de alguma forma, transmití-las para no novo programa fazer o

processamento. Todavia, esta comunicação entre estes dois programas mostrou-se de uma complexidade inusitada. Mesmo que as duas aplicações residam na mesma memória de um computador, a linguagem Java não disponibiliza funcionalidades de memória compartilhada para programas distintos. Dessa forma, as únicas vias possíveis para realizar isso são através do RMI (*Remote Method Invoaction*) ou do TCP/IP, duas funcionalidades específicas para computadores distribuídos e, por isso, sendo consideravelmente mais lentas em comparação ao compartilhamento de memória. Como o pacote RMI do Java é uma API que disponibliza uma boa base de funções para este tipo de comunicações, encarregando-se também da serialização dos objetos, optou-se por seu uso.

Determinados os mecanismos básicos para proceder com a proposta, o programa paralelo foi incorporado na própria classe de solução, a nova classe denominada *GlobalLocalParallel*, herdeira da classe *Solution*, mostrada na Figura 5.1. Desse modo, além do supermétodo *execute()*, o método *main*, que define o ponto de início da execução do programa paralelo, é incluso juntamente com outros métodos para a comunicação entre processos, como mostrado na Figura 5.1.

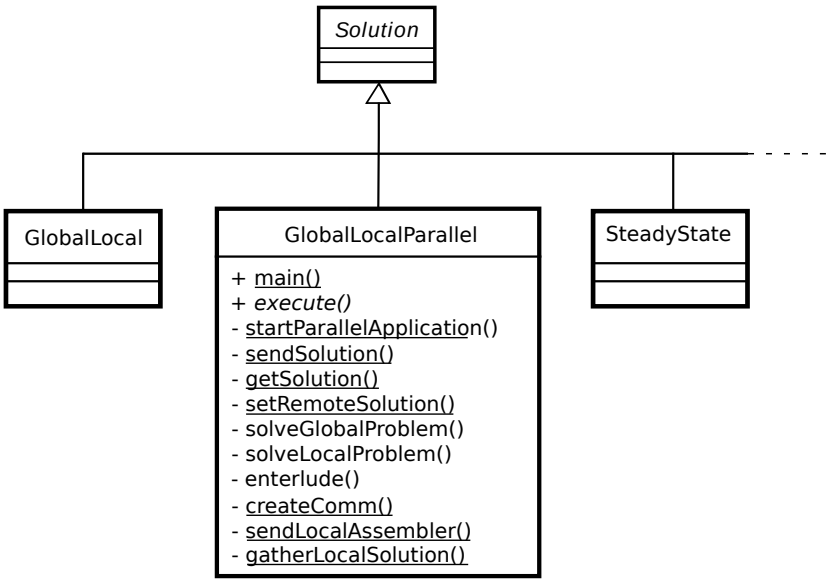


Figura 5.1: Diagrama UML da classe *Solution* com detalhes dos novos métodos da classe *GlobalLocalParallel*

As funcionalidades da nova implementação podem ser melhor expostas a partir do fluxograma da Figura 5.2. A nova classe começa a atuar através do método abstrato *execute()*, herdado da superclasse *Solution*. Ao entrar neste método, inicia-se a rotina *sendSolution()*, que prepara o ambiente RMI para envio de dados do problema contidos na própria classe *GlobalLocalParallel*, isto é, objetos como *FemAssembler*, *FemModel*, lista de elementos, nós, materiais, etc. Em seguida, inicializa-se a aplicação paralela através do comando *startParallelApplication()*. Neste instante, os dois programas estão em andamento, entretanto, somente a aplicação paralela é que está efetivamente em execução.

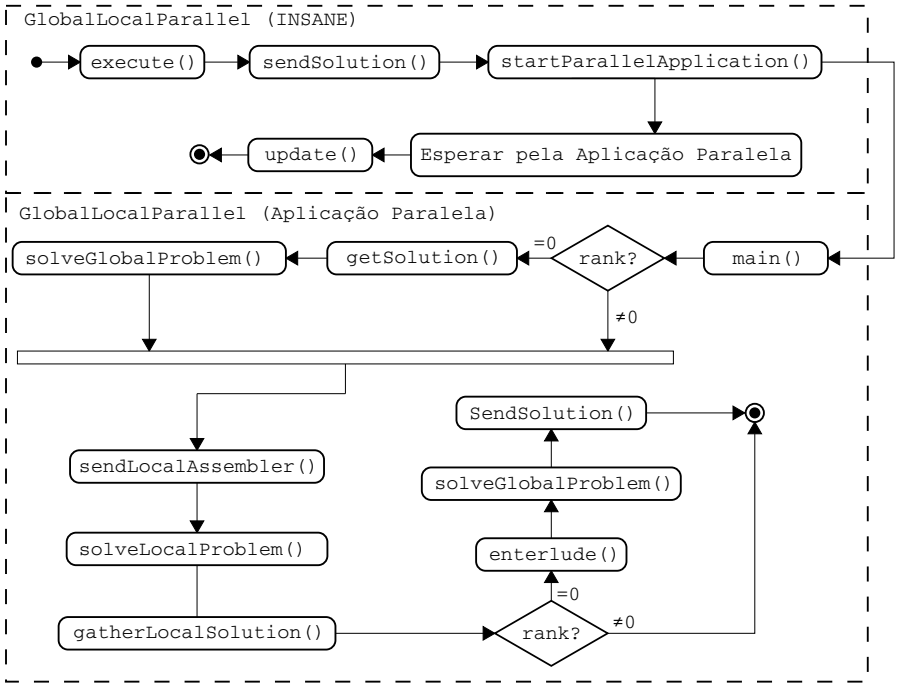


Figura 5.2: Fluxograma dos método executados pela implementação paralela do método Global-Local

Após a inicialização dos programas paralelos com o MPI para Java, apenas um destes deve receber o objeto *GlobalLocalParallel*, de forma a evitar cópias desnecessárias em outros processos. Para isto, seleciona-se apenas o processo de *rank* igual a zero, que aciona o método *getSolution()* e, em seguida, resolve o problema global com *solveGlobalProblem()*.

Com a solução global definida, enviam-se os objetos *Assembler* dos problemas locais para os demais *ranks*, com o método *sendLocalAssembler()*, cada um contendo seu próprio objeto *Model* já com as devidas condições de contorno oriundas da resposta global. Assim, todos os processos resolvem um problema local com *solveLocalProblem()*. Em seguida os resultados são enviados para o processo de número 0, que é responsável por finalizar a análise enquanto os demais processos são finalizados.

O *rank* 0 acaba a análise executando o método *enterlude()*, que tem a finalidade de repassar os resultados dos domínios locais para o domínio global, e, então, solucionar este, novamente, com *solveGlobalProblem()*. Por último, o *rank* 0 retorna a resposta para o programa principal, devolvendo todos os objetos atualizados, através do método *setRemoteSolution()*

Para testar a implementação, utilizou-se o problema estudado em Alves (2012) de uma cunha sujeita a uma força concentrada em seu vértice, apresentada na Figura 5.3a. A peça é modelada como estado plano de deformações utilizando-se elementos triangulares de três nós. A malha global foi modelada com um total de 25 elementos. Já as malhas locais, foram utilizados 16 elementos para a malha local 1 e 48 elementos para as malhas locais 2 e 3, como ilustrado nas Figuras 5.3a e 5.3b.

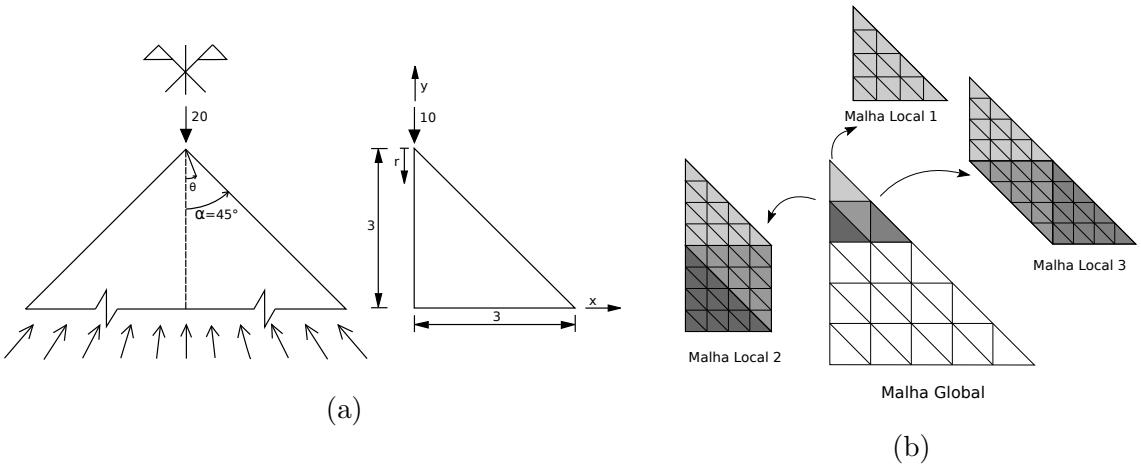


Figura 5.3: Exemplo de uma cunha utilizado para testar a implementação Global-Local Paralela (Autor: Alves, 2012)

A simplicidade do modelo testado foi intencional, com o único intuito de validar a implementação paralela da estratégia. Visto que o projeto está em andamento, ainda não existem ferramentas para gerar um problema Global-Local de forma automática, portanto, dificultando a criação de um modelo muito refinado que justifique a utilização da implementação paralela. Desse modo, não foi possível concluir sobre a eficiência da proposta, uma vez que o custo de comunicação para a transferência dos modelos locais é bastante alto. No entanto, foi possível identificar que a serialização dos objetos, realizada pela interface *Serializable* da linguagem Java, é bastante ineficiente. Assim, para estabelecer um *framework* paralelo vantajoso é preciso elaborar uma rotina própria de serialização dos objetos do programa.

Também, concluiu-se que não há razão em aplicar o MPI em computadores de memória compartilhada, pois há muito desperdício de tempo na serialização de objetos. Dessa maneira, não existe uma justificativa para utilizar uma biblioteca Java puro. Portanto, decidiu-se concentrar o trabalho numa implementação para o *cluster* que faça uso de bibliotecas de solucionadores paralelos.

5.2 Implementação Paralela de Elementos Finitos

Decidido que o paralelismo deveria, neste trabalho, ser focado no uso do *cluster*, optou-se-se então pela utilização do MPI nativo e, consequentemente, de uma biblioteca de *solvers* paralelos, mostradas na seção 3.2. Também, percebeu-se que a estratégia de iniciar o ambiente paralelo a partir de uma aplicação à parte continua sendo válida, pois evitam-se as complexidades de alterar grande parte do código original. Assim, decidiu-se expandir esta ideia, de forma que fosse possível generalizar o uso de outras classes *Solution* paralelas. Para tanto, criou-se a classe *ParallelSolutionApplication*, que contém o ponto de inicialização do programa paralelo, e, assim, desvinculou-se o programa principal paralelo das próprias classes *Solution* paralelas.

Como a comunicação interprocessos ainda se faz necessária, esta deve ocorrer

agora entre processo de *rank* 0 do programa *ParallelSolutionApplication* e o sistema INSANE. Para centralizar estas transferências, decidiu-se pela criação de um gerenciador de dados, chamado *SolutionDataManager*, que também é responsável pela serialização e desserialização dos subdomínios da malha e dos resultados. Também, percebidas algumas rotinas base para a classe *Solution* e a interface *Assembler* no algoritmo paralelo, decidiu-se generalizá-las com a criação da classe abstrata *ParallelSolution* e da interface *ParallelAssembler*. A seguir, os novos métodos e bibliotecas acrescentados ao sistema bem como o funcionamento geral da proposta são descritos.

5.2.1 METIS

A biblioteca METIS (Karypis e Kumar, 1998) é um dos vários projetos desenvolvidos e mantidos pelo professor George Karypis, do Departamento de Ciência da Computação na Universidade de Minessota. Ela oferece rotinas de particionamento de grafos e malhas não-estruturados através da metodologia multinível com bissecção recursiva ou *k-way*. Escrito em C, o programa compromete-se com eficiência e velocidade, além de disponibilizar configurações para adaptar-se às necessidades do usuário. Por exemplo, é possível selecionar diferentes algoritmos para cada etapa da técnica multinível, diferentes pesos para elementos do grafo, número máximo de iterações, impor condição de continuidade nos domínios particionados, solicitar minimização da comunicação ou escolher a precisão da divisão do grafo. Disponível desde 1997, ela é, provavelmente, uma das aplicações de particionamento de grafos mais utilizadas, sendo, inclusive, empregada na redução do *fill-in* de matrizes do pacote SuiteSparse.

Para a sua integração com o JNI no particionamento das malhas, optou-se pelo *element-cut*. Ou seja, decidiu-se por partir as arestas de todos os elementos localizados nas fronteiras dos subdomínios, e duplicando-os nos subdomínios vizinhos, como demonstrado na Figura 5.4a. Esta decisão deu-se devido ao receio de que a comunicação entre computadores onerasse a solução proposta, já que o trabalho

apenas abordou análises lineares. Além disso, utilizou-se as opções de continuidade, mínima comunicação dos domínios e o método *k-way* de divisão de grafos. Optou-se por não alterar a precisão da divisão da malha, deixando o valor padrão de 3%.

5.2.2 PETSc

Dentre as várias bibliotecas de solucionadores paralelos citados na seção 3.2, decidiu-se pelo uso da PETSc (*Portable, Extensible Toolkit for Scientific Computation*). Segundo os desenvolvedores, a biblioteca possui um conjunto de rotinas e estruturas de dados que proporcionam ferramentas-base para aplicações paralelas em larga escala pelo padrão MPI de comunicação (Balay et al., 2019*a*). Além de funcionalidades essenciais com matrizes e vetores paralelos, também estão inclusos diversos tipos de solucionadores lineares, não lineares e *solvers* de equações diferenciais ordinárias e equações algébricas diferenciais. Também, é possível integrá-la em linguagens como FORTRAN, C, C++ e Python, além de CUDA e OpenCL, para GPUs. Ademais, existe uma extensa documentação com exemplos, facilitando seu entendimento e uso.

Para a implementação do JNI desta biblioteca, foi necessário decidir o tipo de matriz utilizada, pois são disponibilizados diversos formatos (esparsas, densas, sequenciais, paralelas e em blocos) de forma a adequar a necessidade do usuário. Escolheu-se trabalhar com a matriz paralela esparsa de formato CSR (*Compressed Sparse Row*), pois este é o formato padrão da PETSc. Nesta configuração, cada processador guarda um conjunto de linhas da matriz de rigidez global.

Para não haver comunicação desnecessária, estabeleceu-se que as linhas pertencentes a cada processador corresponderiam aos graus de liberdade atribuídos ao próprio subdomínio, como é representado, de forma geral, na Figura 5.4. Nesta imagem, a malha particionada, Figura 5.4a, tem seus graus de liberdade numerados em sequência para cada domínio, Figura 5.4b. Assim, cada subdomínio irá montar apenas a parte da matriz que se refere aos seus graus de liberdade, representado na

Figura 5.4c pelas cores correspondentes a cada domínio. Com isto, as parcelas de rigidez são calculadas no mesmo processador que devem ser armazenadas.

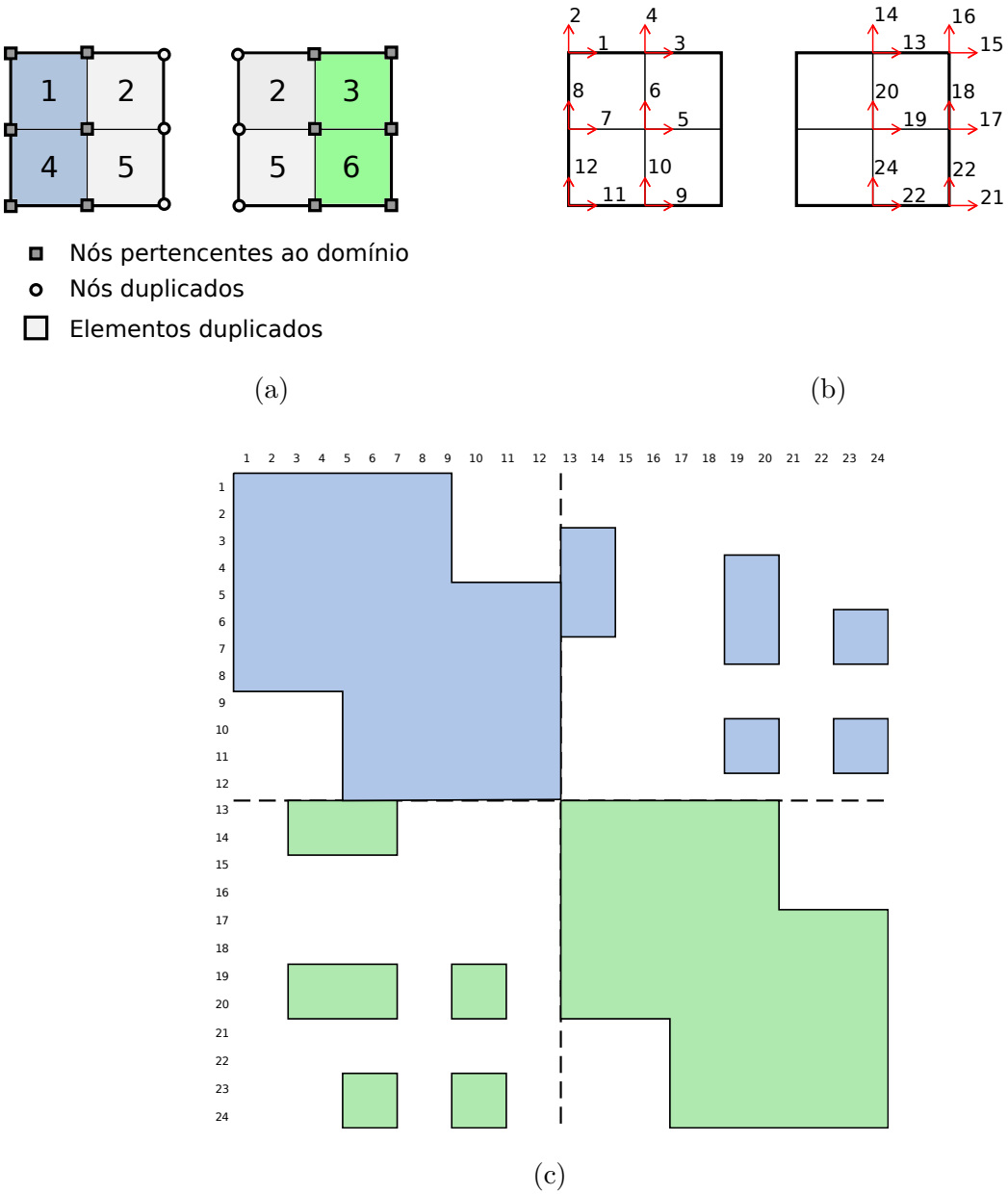


Figura 5.4: Ilustração de (a) uma partição de uma malha em dois domínios com *element-cut*, (b) a numeração global dos graus de liberdade realizada por domínios e (c) o armazenamento da matriz paralela esparsa em linhas para cada processador (identificados pelas cores correspondentes aos domínios)

Como a matriz paralela deve armazenar linhas inteiras, algumas colunas dos

graus de liberdade que não fazem parte do domínio estão presentes devido ao corte gerado na malha. Estas parcelas da matriz são calculadas sem qualquer custo de comunicação devido a duplicação dos elementos. No exemplo da Figura 5.4c mostra-se a divisão que ocorre naturalmente com as colunas que são de graus de liberdade de outro domínio.

Conforme é aconselhado pelos desenvolvedores da PETSc, por questões de eficiência, realizou-se o pré-alocamento de memória para a matriz. Para isto, mapearam-se os graus de liberdade que possuem valores não nulos e suas respectivas posições na matriz, como já era feito em outras implementações de matrizes esparsas do programa INSANE, mas agora conforme padrão necessário para a PETSc.

Para o solucionador, decidiu-se trabalhar com as implementações baseadas em Gradientes Conjugados, com uma tolerância de 10^{-10}. Também, escolheu-se utilizar apenas o pré-condicionador de Jacobi, apesar de ser uma etapa de estudo importante para uma rápida convergência do Gradientes Conjugados.

5.2.3 Classe *SolutionDataManager*

Para facilitar a transmissão dos dados entre o programa paralelo e o INSANE, criou-se o gerenciador de dados *SolutionDataManager*, mostrado na Figura 5.5. Durante a execução, ao invés de enviar o problema original em sua totalidade, já são encaminhados todos os subdomínios com todos os dados da análise. Assim, após a partição da malha, que é desempenhada pela interface *ParallelAssembler*, os resultados são guardados nas variáveis *domainElementList* e *domainNodeList*, mostradas no diagrama da Figura 5.5. Após a definição destas, inicia-se a serialização dos domínios através do *serializeDomains()*. Cada domínio gera um vetor de *bytes*, que são armazenados em uma lista na variável *data*.

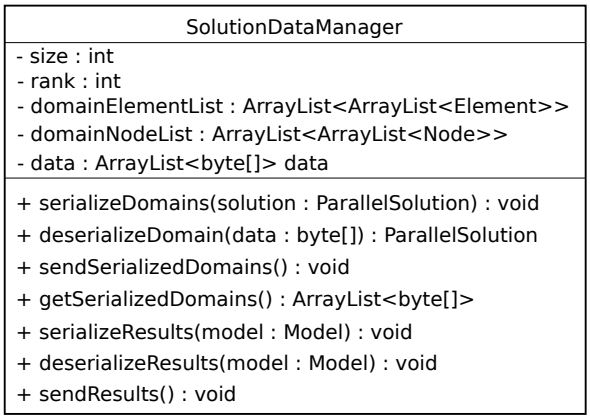


Figura 5.5: Diagrama UML da nova classe *SolutionDataManager*

Ao acionar o processo de solução, os dados da serialização são preparados para envio através do método *sendSerializedDomains()*. Então, o *rank* 0 da aplicação paralela solicita a lista de vetores de *bytes* de todos domínios através de *getSerializedDomains()*. Em seguida, cada vetor de *bytes* é enviado ao seu respectivo *rank* para ser desserializado através do *deserializeDomain()*.

Após a solução do problema em paralelo pela classe *ParallelSolution*, as respostas de deslocamentos e de reações das restrições são serializadas em cada processo com *serializeResults()*. Estas, então, são enviadas para o *rank* 0, que deve transmitir ao programa principal por *sendResults()*. Enfim, no INSANE os resultados são desserializados através da chamada de *deserializeResults()* e, em seguida, enviados para a impressão do arquivo, com o método *update()*.

5.2.4 Interface *ParallelAssembler*

De modo a separar o *Assembler* sequencial da versão paralela, criou-se a interface *ParallelAssembler*. Esta extende a primeira, de forma a garantir que as novas classes também tenham os métodos básicos de construção dos sistemas de equações, como pode ser visto na Figura 5.6.

Percebeu-se que a partição dos domínios poderia ser realizada na classe *Solution*,

na interface *Assembler* ou na classe *Model*. Escolheu-se efetuar esta etapa diretamente no *ParallelAssembler*, pois além de esta possuir acesso direto ao modelo, deu-se preferência por não criar uma nova classe *Model* paralela. Portanto, foram definidos três métodos na nova interface, *partitionDomains()*, *numberDomainsEquations()* e *initParallelAssembler()*.

Em *partitionDomains()* ocorre a chamada JNI do METIS para a divisão da malha, sendo o resultado de cada domínio gravado na lista de nós e elementos do *SolutionDataManager*. O método *numberDomainsEquations()* é chamado logo após e tem como finalidade realizar a numeração global dos graus de liberdade por domínio. Esta etapa é necessária para definir a posição das parcelas de rigidez na matriz global paralela, sendo realizada em um domínio de cada vez, de forma a criar uma numeração contínua em cada processador, conforme explicado na seção 5.2.2. Ambas as rotinas são realizadas para preparar a estrutura de dados para a computação paralela e, por isso, são utilizadas antes de se iniciar o ambiente MPI.

O último método incluso, *initParallelAssembler()*, tem uma função análoga ao *init()* do *Assembler* sequencial, executando rotinas que devem preceder o processamento no programa paralelo. Neste método são numeradas as equações para a formação das submatrizes de cada processo, calculadas as quantidades de número não nulos e alocação de memória da matriz de rigidez esparsa, inicializados carregamentos e variáveis para o começo da análise.

Também na Figura 5.6, mostra-se a nova classe implementada, *FemAssemblerParallel*. Nela, escolheu-se por continuar utilizando o formato esparso apenas na parcela C_{uu} , isto é, são matrizes densas as parcelas C_{up}, C_{pu} e C_{pp}. Desse modo, as variáveis *rowPtr* e *colPtr* têm a função de mapear as posições não nulas da matriz C_{uu} a partir do método *setSparseDs()*, de forma igual à classe *FemAssemblerSparse*.

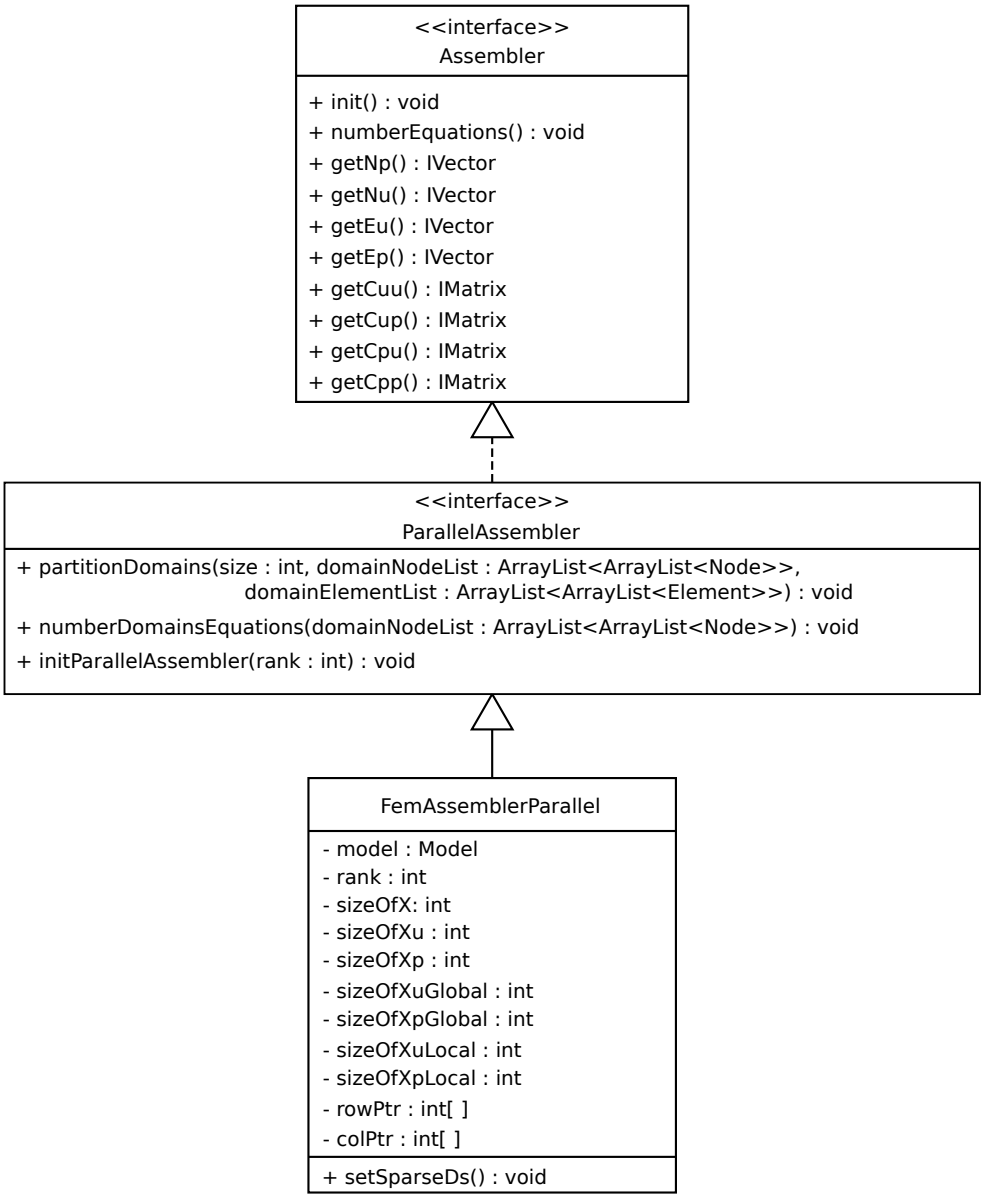


Figura 5.6: A relação de herança da nova interface *ParallelAssembler* com *Assembler* e a nova classe *FemAssemblerParallel* que a implementa

Para as matrizes densas, foi necessário adicionar outras variáveis para representar o número de graus de liberdade total do problema, o número de graus de liberdade pertencentes ao domínio e o número total de graus de liberdade do domínio, isto é, incluindo os graus de liberdade copiados. Portanto, as variáveis com *Global* no nome, referem-se ao total de graus de liberdade do problema, podendo ser desconhecidos ($sizeOfXuGlobal$) ou prescritos ($sizeOfXpGlobal$). Já as variáveis

com o sufixo *Local* são reservadas para identificar o total de graus de liberdade que pertencem ao domínio. Por fim, as variáveis $sizeOfX$, $sizeOfXu$ e $sizeOfXp$ são, respectivamente, o número total de graus de liberdade, o número total de graus de liberdade desconhecidos, e o número total de graus de liberdade prescritos, todos referentes à soma dos graus de liberdade pertencentes ao domínio e os copiados.

No exemplo da Figura 5.4 de uma malha dividida em dois domínios, temos que $sizeOfXuGlobal = 24$, $sizeOfXpGlobal = 0$. Para ambos os domínios temos: $sizeOfXuLocal = 12$ e $sizeOfXu = 18$. Dessa maneira, quando, por exemplo, a matriz C_{up} for montada, ela deverá ter o número de linhas igual a $sizeOfXuLocal$ e número de colunas igual a $sizeOfXp$.

5.2.5 Classe *ParallelSolution*

Generalizaram-se os métodos necessários para a análise paralela a partir da criação da classe *ParallelSolution*, que extende diretamente *Solution*. Optou-se por deixar a instância da classe *SolutionDataManager* nesta classe pois a criação do programa paralelo e o consequente envio dos domínios serializados deve ocorrer dentro do método *execute()*, que é o supermétodo responsável por iniciar o processo de solução em todas as classes que estendem *Solution*. Também, foram adicionados três métodos à nova classe, mostrados na Figura 5.7.

Em *startParallelApplication()* inicia-se o executável da classe *ParallelSolutionApplication*, utilizando o *mpirun* da biblioteca OpenMPI e o número de processos que devem ser criados, guardados na variável *size* do *SolutionDataManager*. O método *setParallelAssembler* teve de ser adicionado porque em *setAssembler* são realizadas várias etapas de inicialização do *Assembler* e do *Model*, que precisaram ser alterados no ambiente paralelo. Por fim, em *executeParallel()* é que inicia-se a solução da análise paralela, realizando a montagem de vetores e matrizes, solução do sistema de equações e cálculo de reações em paralelo.

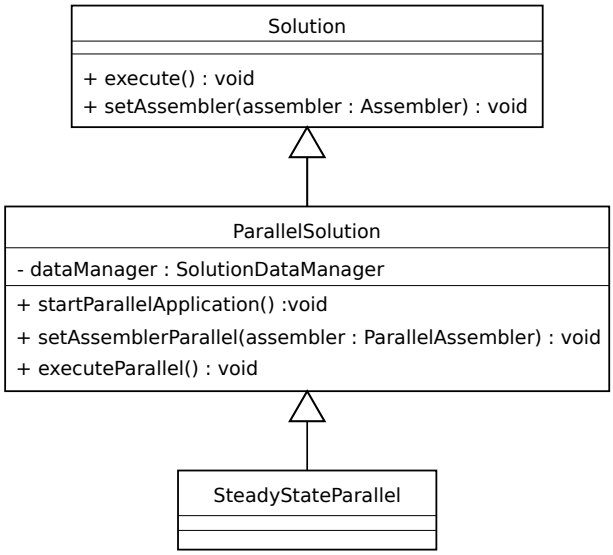


Figura 5.7: Diagrama UML da relação de herança entre a nova classe *ParallelSolution* com *Solution* e a nova implementação *SteadyStateParallel* que a implementa

5.2.6 Classe *ParallelSolutionApplication*

Como explicado anteriormente, a adição desta classe foi realizada principalmente para criar um único ponto de partida para o ambiente paralelo. Para a comunicação interprogramas (INSANE e programa paralelo) e chamadas MPI do *rank* 0 com os demais *ranks*, criaram-se os três métodos estáticos *getParallelSolution()*, *scatterData()* e *gatherResults()*, mostrados na Figura 5.8 e que são explicados a seguir.

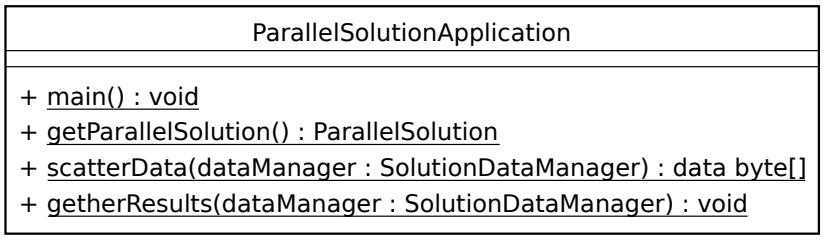


Figura 5.8: Diagrama UML da nova classe *ParallelSolutionApplication*

Para facilitar o entendimento tanto das funcionalidades desta classe como da

sequência de execuções que são realizadas para toda análise paralela, criou-se o fluxograma da Figura 5.9. Assim como na paralelizaçao do método Global-local, a implementação inicia-se pelo método *execute()*, no INSANE, realizada, agora, por uma classe qualquer que estenda *ParallelSolution*. Dentro deste método, prepara-se o ambiente RMI para o envio da solução através da chamada do *sendSerializedDomains()*, pertencente ao *SolutionDataManager*. Da mesma forma que foi realizado na implementação anterior, a aplicação paralela é criada pelo método *startParallelApplication()*, que inicia o *main()* da *ParallelSolutionApplication*. A partir do método ***getParallelSolution()*** é que ocorre a transferência dos domínios serializados, advindos do INSANE, para o *rank* 0 e, em seguida, distribuídos para os *ranks* remanescentes com ***scatterData()***. Então, ainda em *getParallelSolution()*, desserializam-se os dados, com *desserializeDomains()* do *SolutionDataManager*, criando os objetos pertinentes à análise, como *ParallelSolution*, *ParallelAssembler* e *Model*.

Estando todos os objetos prontos, inicializa-se a resolução paralela do problema com *executeParallel()* da classe *ParallelSolution*. Ao finalizar esta etapa, reúnem-se os resultados apenas no *rank* 0, em ***gatherResults()***, para enviar novamente ao sistema INSANE, através do RMI com *sendResults()* do *SolutionDataManager*. Então, por fim, os resultados são desserializados com *deserializeResults()* do *SolutionDataManager* e o arquivo de saída é gerado através do método *update()*.

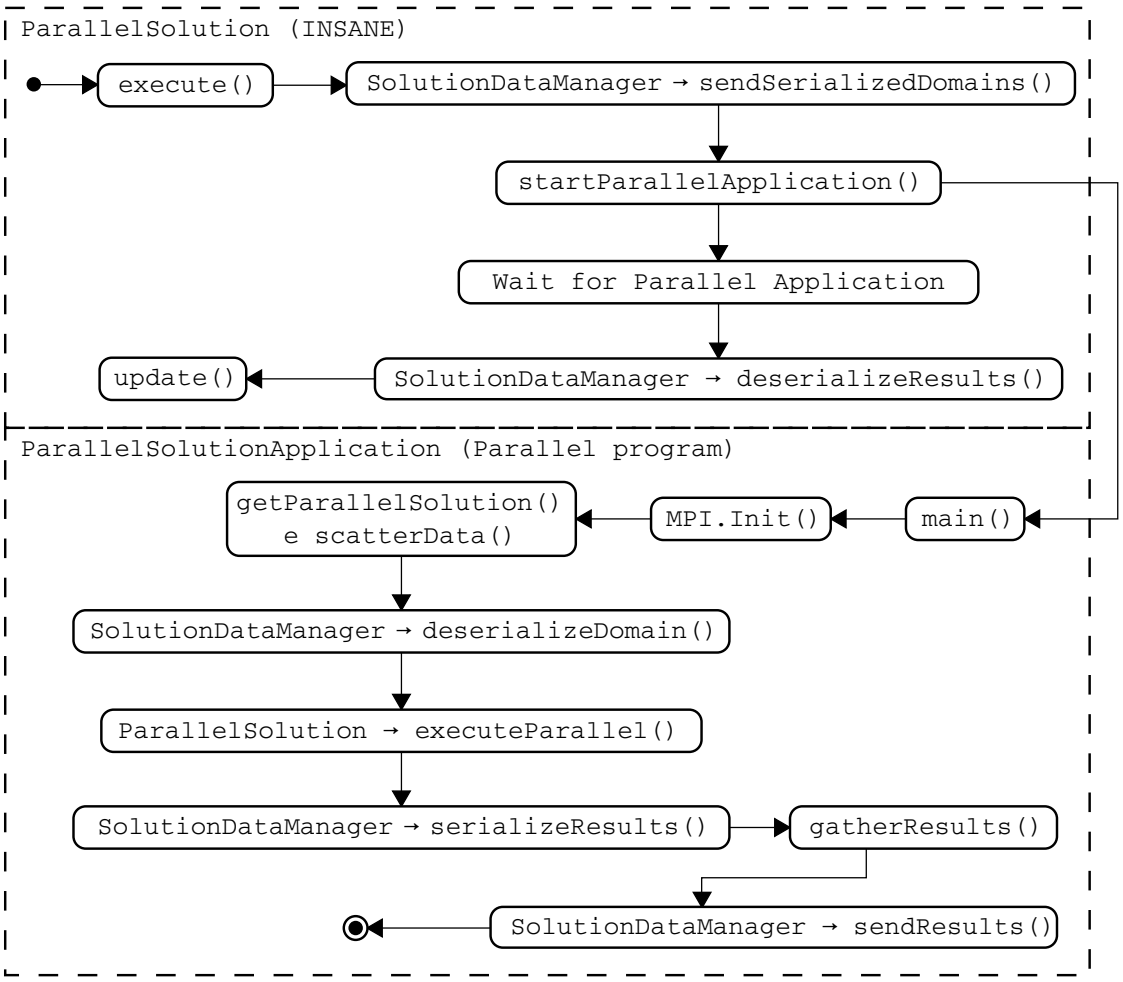


Figura 5.9: Fluxograma dos métodos chamados no método *execute()* da classe *ParallelSolution* e o *main()* da classe *ParallelSolutionApplication*

5.2.7 Refatoração da Solução de Equações

Como mencionado anteriormente, até então, existia um problema grave de repetição de código na escolha do solucionador das equações lineares. Esta era realizada por uma estrutura de *if-else* em cada uma das classes *Solution*, apesar de já existir a classe *LinearEquationSystems* para guardar a estrutura de dados referente a um sistema de equações. Em vista disto, foi realizada uma refatoração desta fase importante das análises, de modo a trazer um melhor encapsulamento do código. Para isto foi criada a classe abstrata *Solver*, que foi adicionada ao pacote *util/linearalgebra*. Todas as implementações de solucionadores foram, então, estabelecidas em

classes próprias, estendendo essa e nomeadas conforme o nome do algoritmo. Como mostrado na Figura 5.10, um objeto *Solver* deve possuir uma matriz, a, um vetor de incógnitas, x, e um vetor de constantes, b, para formar um sistema de equações lineares e acionar o solucionador com *solveX()*.

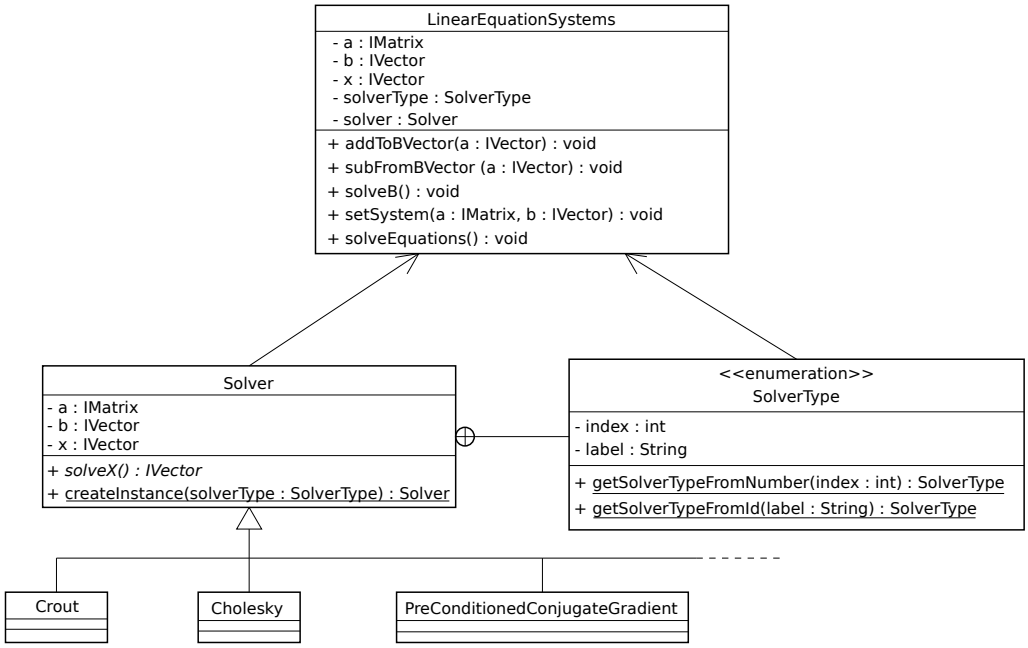


Figura 5.10: Diagrama UML da nova classe *Solver* e a enumeração *SolverType*

Além disso, para facilitar o uso e instância destes objetos, criou-se uma enumeração com nome *SolverType* que deve substituir a variável inteira *solverType* da classe *Solution*. Assim, nesta enumeração são armazenados o nome do *solver*, para ser visualizado na interface gráfica, e seu índice, que é o número de identificação do solucinador no arquivo de leitura.

As duas novas classes, então, foram definidas como atributos da classe *LinearEquationSystems*, que teve os métodos *setSystem()* e *solveEquations()* adicionados de forma a desencadear o processo de resolução da classe *Solver*. Com esta nova funcionalidade, simplificou-se a adição de novas ferramentas de solução, como, por exemplo, a biblioteca PETSc.

Para armazenar todos os códigos necessários para a criação dos *bindings* de bibliotecas nativas e desvincular qualquer classe pertencente ao INSANE desta tarefa,

criou-se um novo repositório especializado apenas nas implementações JNI. Assim, cada novo projeto contido neste repositório é referente a uma diferente biblioteca. Estes projetos contém o código Java que realiza a chamada JNI, que tornam-se uma biblioteca Java a ser importada, o código C que faz a chamada para os métodos nativos, a biblioteca original, compilada para linux e windows, e a biblioteca dinâmica compilada com a biblioteca original e o código C criado.

Aproveitando as modificações, foram inclusas funcionalidades da biblioteca ParallelColt (Wendykier e Nagy, 2010), que é a versão *multithreaded* da biblioteca Colt, desenvolvida para computação científica de alta performance em Java puro. Apesar do *multithreading* estar presente apenas nos *solvers* iterativos, baseados no subespaço de Krylov, também são disponibilizados solucionadores diretos. Além destes, são disponibilizados precondicionadores, com matrizes esparsas e densas, operações de precisão simples e dupla além de uma infraestrutura para operações com números complexos.

Capítulo 6

Simulações Numéricas e Resultados Obtidos

Neste capítulo são apresentadas simulações numéricas para avaliar a proposta de implementação computacional desenvolvida neste trabalho e, assim, investigar os resultados e as possíveis interpretações destes. Para a verificação da eficiência desta implementação, utilizou-se o *cluster* do grupo de pesquisa, que consiste em três máquinas com as especificações descritas na Tabela 6.1 e que fazem uso do sistema operacional CentOS 6.5.

Tabela 6.1: Especificações do *cluster* utilizado nos testes

Nome	*HeadNode*	*Node 0*	*Node 1*
Sistema	IBM System x3550 M4	IBM System x3650 M3	IBM System x3650 M3
Processadores	2x Xeon E5-2620	2x Xeon E5645	2x Xeon E5645
Cores/Processador	6	6	6
Memória	2x 8GB DDR3	2x 8GB DDR3	2x 8GB DDR3

6.1 Considerações Sobre os Solucionadores

Para o entendimento da eficiência da implementação distribuída, é preciso compará-la com o código sequencial em sua forma mais eficiente. Assim, primeiramente, fez-se um exame dos solucionadores implementados no *software* para definir qual destes empregar durante a obtenção do parâmetro base T_1 da Equação 2.1. Para isto utilizou-se um modelo estático linear de uma viga em flexão, ilustrada na Figura 6.1. Esta foi discretizada na malha mostrada na Figura 6.2, com 13664 elementos e 6933 nós, com um total de 13863 graus de liberdade desconhecidos. De forma a capturar os tempos dos diferentes *solvers*, foram testados os solucionadores de matrizes esparsas, com resultados na Tabela 6.2, bem como os de matrizes densas, com resultados na Tabela 6.3, apesar da implementação do trabalho ser apenas com matrizes esparsas.

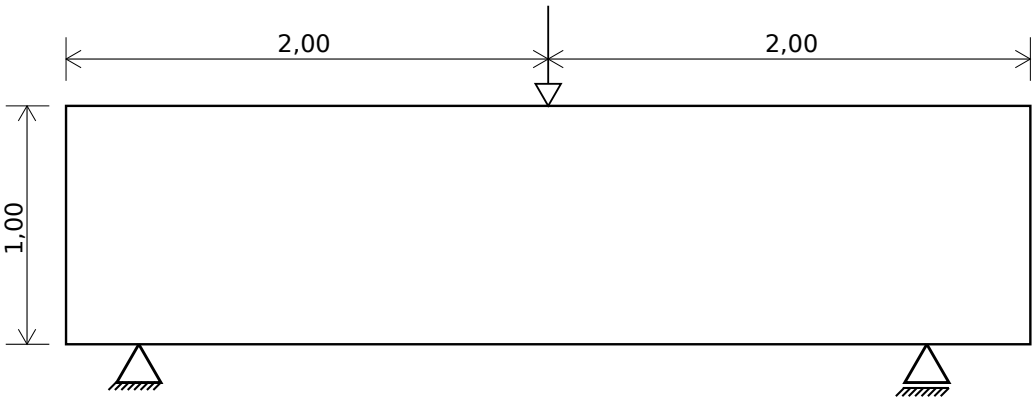


Figura 6.1: Viga em flexão utilizada para o teste dos solucionadores

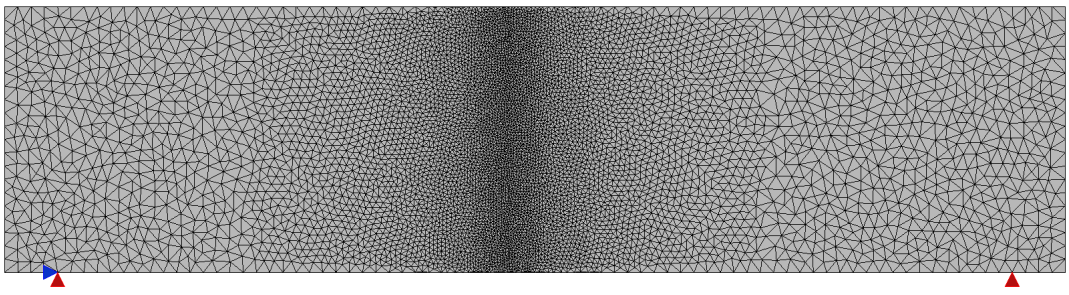


Figura 6.2: Discretização do exemplo de viga para o teste com os solucionadores

Tabela 6.2: Tempos dos solucionadores esparsos implementados no sistema INSANE

Solucionadores		Tempo (s)
UMFPACK		0,13
Skyline		1,01
ParallelColt	Lu	2,30
	Cholesky	0,77
	QR	3,99
	Gradientes Conjugados	20,91

Tabela 6.3: Tempos dos solucionadores densos implementados no sistema INSANE

Solucionadores		Tempo (s)
Crout		2015,00
Cholesky		702,54
Flame		104,85
Gradientes Conjugados com Jacobi		798,38
ParallelColt	Lu	922,97
	Cholesky	61,03
	QR	133,43
	Gradientes Conjugados	468,14

Como comentado anteriormente na seção 4.3.4, a rotina UMFPACK faz parte da biblioteca SuiteSparse criada para operações matriciais de alta performance. Assim, pelos resultados, percebe-se com clareza a rapidez deste solucionador nativo para problemas esparsos, chamado no INSANE através do *framework* JNI. Por isso, utilizou-se esta implementação como base de comparação com os resultados paralelos.

6.2 Considerações Sobre o Cálculo de *Speedup*

O cálculo do *speedup* é uma etapa bastante importante, pois ele que aponta a eficiência da implementação paralela. Por isto, dedicou-se esta seção para esclarecer a metodologia utilizada neste ponto da avaliação do código.

Identificado o melhor solucionador, partiu-se para o *profiling* de uma análise grande o suficiente para viabilizar a computação paralela. Para isto, selecionou-se um problema em estado plano de de tensões de uma chapa retangular com um furo em seu centro, conforme mostrado na Figura 6.3. Foram criadas 3 malhas de elementos triangulares, variando-se o número total de elementos, conforme mostrado na Tabela 6.4. Alguns exemplos das divisões de domínios realizadas na malha pela biblioteca METIS durante a execução do problema de forma distribuída são expostos na Figura 6.4.

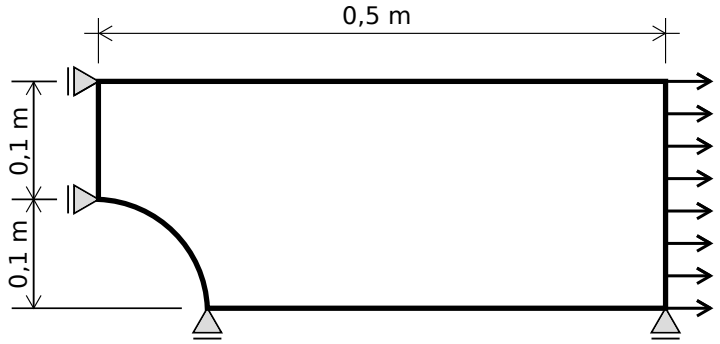
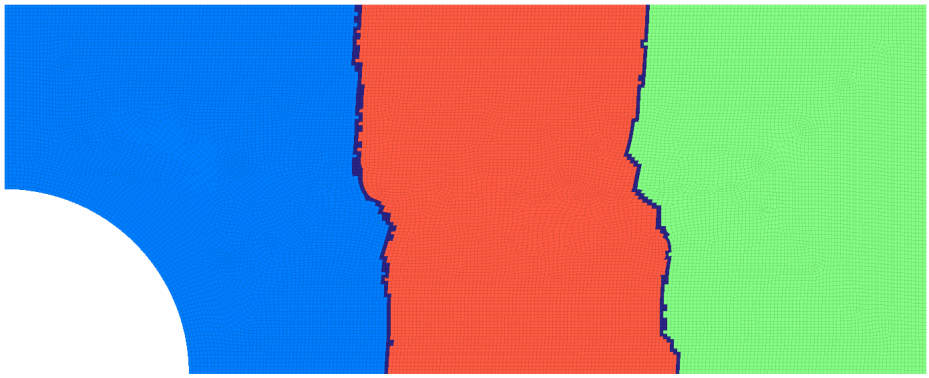


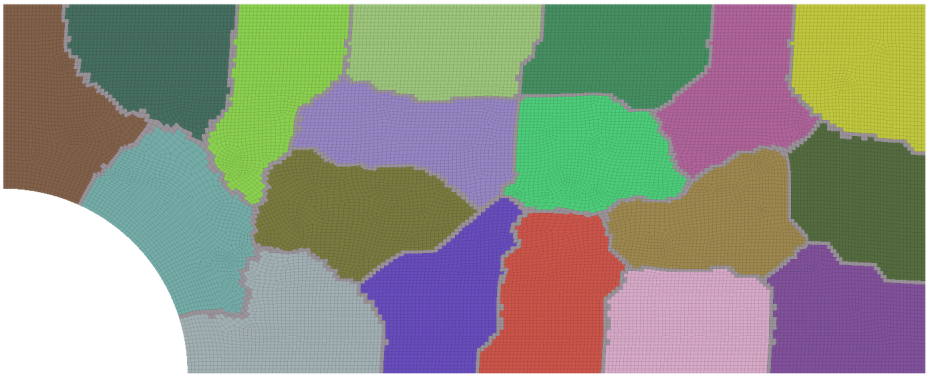
Figura 6.3: Modelo utilizado de exemplo de uma placa furada com um carregamento distribuído

Tabela 6.4: Informações das três malhas utilizadas nos testes

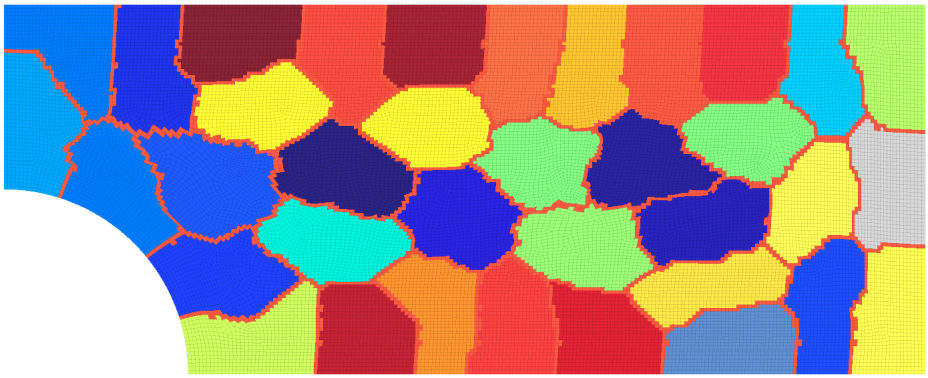
Malha	1	2	3
Número de elementos	24.985	45.148	97.642
Número de nós	25.318	45.595	98.307
Graus de liberdade desconhecidos	50.384	90.854	196.113
Graus de liberdade prescritos	252	336	501



(a)



(b)



(c)

Figura 6.4: Divisões da malha do problema com (a) 3 domínios, (b) 18 domínios e (c) 36 domínios. Os elementos que são copiados em dois ou mais domínios estão representados em uma única cor.

Para o estabelecimento de um tempo médio representativo do processo de solução, foram executados um total de 5 vezes o programa INSANE, em sua totalidade, para cada malha, e guardados os tempos de execução dos métodos de interesse. Considerou-se que, para o entendimento do comportamento geral do *software*, este total de amostras é suficiente. Claro que a variação dos resultados tornou-se mais acentuada com o aumento da malha e o número de *cores* utilizados. Entretanto, na maioria dos casos, estes valores não mostraram-se significativos quando comparados ao tempo total. Nos poucos casos que esta variação foi considerável, decidiu-se, mesmo assim, não aumentar o número de amostras. Entendeu-se que, considerando o contexto geral que desejava-se obter, os resultados do trabalho não foram prejudicados. Também, acredita-se que não haveria um ganho significativo com o aumento na precisão da média de tempo de execução do programa. Ainda, todos os valores de tempo que são apresentados em gráficos foram disponibilizados no Apêndice A em forma de tabelas.

Quanto à metodologia de aquisição dos tempos, considerou-se suficiente, ainda que mais trabalhosa, a obtenção de tempos através da direta solicitação ao sistema. Esta decisão foi tomada, principalmente, devido a dificuldade de uso com *profilers*. Na parcela Java do programa, a ferramenta *Async Profiler*, um *profiler* de código aberto e livre, mostrou-se bastante útil e simples de usar, contudo, os resultados mostrados pelo programa, às vezes, não correspondiam aos tempos coletados diretamente pelo sistema. Concluiu-se que a incoerência ocorria quando um tempo considerável era utilizado para o alocamento de memória. Desta forma, o *profiler* contabiliza este tempo de alocação como sendo pertencente ao método do sistema e não como um tempo do método que o originou. Apesar de parecer um aspecto ruim, isto acaba tornando fácil a detecção de ineficiências geradas pelo gerenciamento de memória oriundas de um método.

Já com relação ao código escrito em C, tentou-se, também, fazer o uso de dois *profilers* específicos do MPI, mpiP (Vetter e Chambreau, 2014) e mpe2 (Chan et al.,

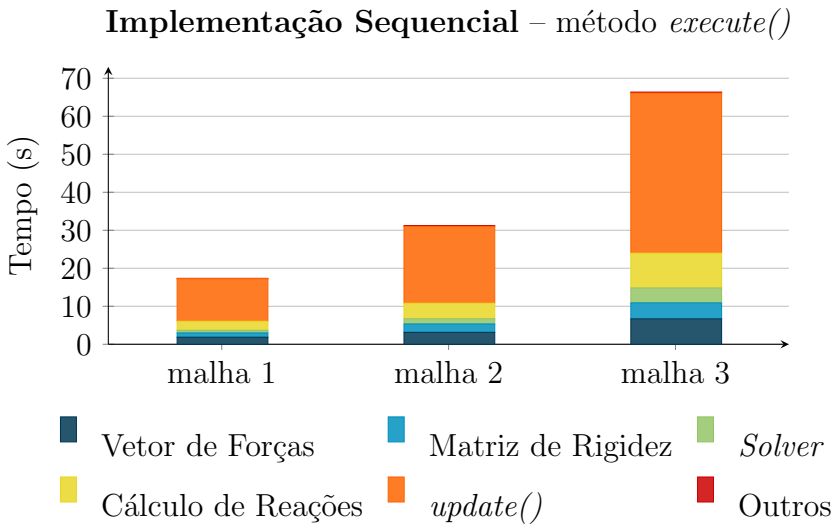
2008). O primeiro não tem a função de perfilar a totalidade do programa C, mas apenas das chamadas MPI, verificando a função utilizada, onde ela ocorre, o tamanho das mensagens enviadas e o tempo total despendido na função. Apesar de exibir resultados, algumas das chamadas do MPI não apresentaram identificação, tornando impossível rastrear sua origem, e, por isto, não foi possível a sua utilização. Já o segundo programa não foi possível operar integrado com o JNI.

Os tempos despendidos na divisão, numeração e serialização dos domínios são mostrados na Tabela 6.5. Decidiu-se por apresentar estes valores à parte, pois estes não fazem parte efetiva do processo de solução.

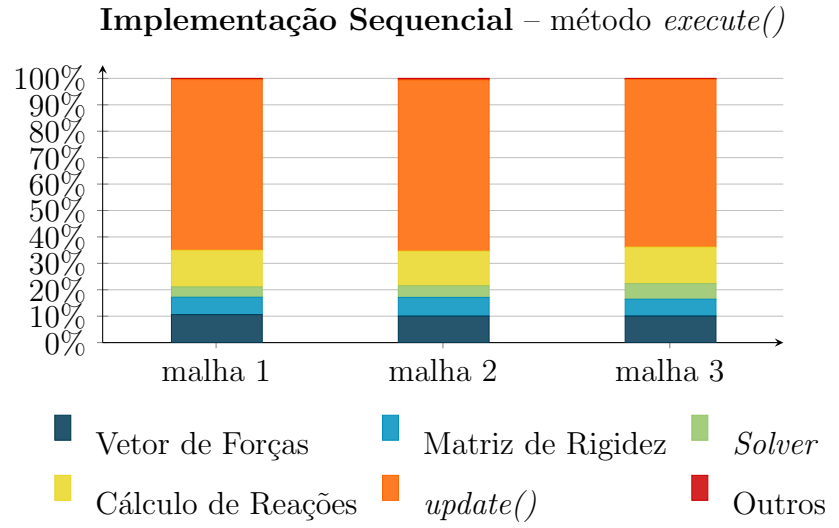
Tabela 6.5: Tempos de particionamento e serialização dos testes

	partitionDomains()	*numberDomains()*	*serializeDomains()*	Tempo Total
Malha 1	0,31 s	0,09 s	0,14 s	0,54 s
Malha 2	0,42 s	0,10 s	0,21 s	0,73 s
Malha 3	0,86 s	0,21 s	0,97 s	2,04 s

O resultado do *profiling* da implementação sequencial no método *execute()* da classe *SteadyState* pode ser visto nas Figuras 6.5a e 6.5b. É evidente, apenas pela visualização do gráfico, que existe um domínio de tempo despendido na etapa de escrita dos resultados em arquivo, através do método *update()*. Isto já era um problema familiar ao grupo de pesquisa, sabendo-se principalmente do seu inconveniente em análises grandes e/ou não lineares. Contudo, a dimensão da ineficiência foi destacada com estes exemplos.



(a)

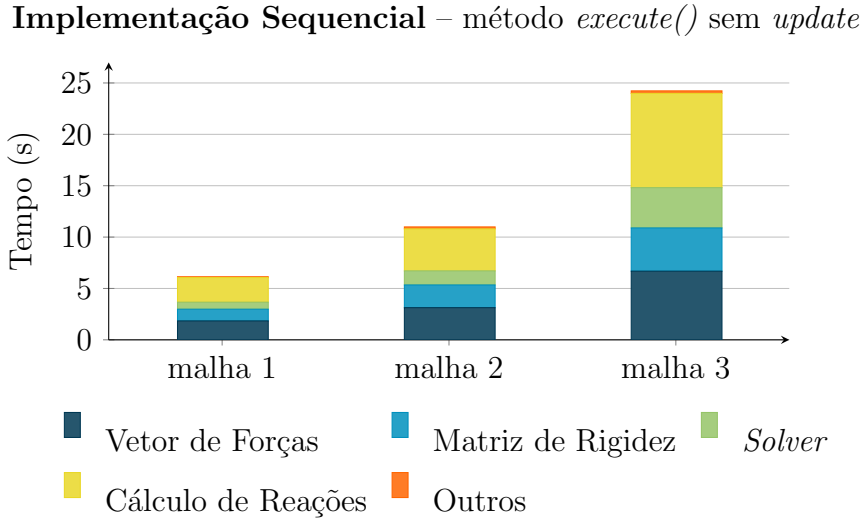


(b)

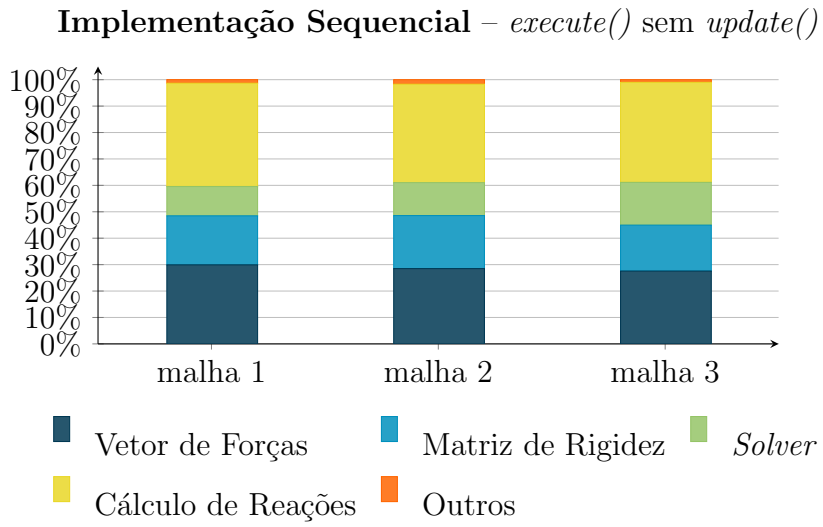
Figura 6.5: Detalhes de tempos despendidos no método *execute()* de cada um dos exemplos (a) em valores totais e (b) em valores percentuais

Como este trabalho não focou no aspecto da paralelização na geração de arquivos de entrada e saída e nem em sua otimização, foram destacados nas Figuras 6.6a e 6.6b apenas os métodos que fazem parte efetiva do processo de solução da análise estrutural estática. Nestes gráficos, é possível constatar que as etapas de montagem do vetor de forças e cálculo de reações predominam, e não a montagem da matriz

de rigidez e solução das equações, como era evidenciado pela revisão bibliográfica, indicando que estes algoritmos não estão otimizados.



(a)



(b)

Figura 6.6: Detalhes de tempos despendidos apenas na resolução de cada um dos exemplos (a) em valores totais e (b) em valores percentuais

Um ponto que pode estar influenciando os resultados é o alocamento de memória. No caso da matriz esparsa, o espaço de memória necessário para sua construção já é definido anteriormente à análise, não havendo alocação durante a etapa de montagem. Isto já não acontece com o vetor de forças e o cálculo de reações, pois

a determinação do tamanho de memória das matrizes e dos vetores necessários são realizados juntos com a análise, contribuindo para o seu tempo total. Também, estas etapas lidam com matrizes cheias, que precisam de muito mais memória e, consequentemente, mais tempo para o alocamento e para as operações matriciais.

Ainda, descobriu-se que, aproximadamente, metade do tempo para a montagem da matriz de rigidez global é utilizado na alocação de memória das matrizes de rigidez dos elementos. No código atual, uma nova matriz é criada cada vez que a rigidez de um elemento é montada, que torna-se inútil logo em seguida. Toda essa memória utilizada acaba sendo acumulada para, posteriormente, o *garbage collector* limpar, promovendo outro gasto considerável de tempo. Contudo, apesar de todos esses aspectos ressaltados, espera-se que a paralelização reduza, de forma geral, os tempos de todas estas etapas.

Quanto à biblioteca PETSc, constatou-se, através de verificações, uma sensível melhora nos resultados de tempos ao utilizar a opções de otimização do código C ($COPTFLAGS$ =*-O3*) e de desativação do debug (*with-debugging* = 0). Outra questão importante foi a escolha de um dos muitos solucionadores disponíveis. Foram testadas três implementações do Gradientes Conjugados: KSPCG (*Conjugate Gradient*), KSPPIPECG (*Pipelined Conjugate Gradient*) e KSPGROPPCG (*Pipelined Conjugate Gradient Gropp*). Por fim, a última opção apresentou resultados melhores e, por isto, os testes incluíram apenas o KSPGROPPCG. Apesar disto, os outros tipos de *solvers* estão inclusos no sistema, além do gradientes biconjugados estabilizado (KSPBCGS), que permite a utilização de matrizes não simétricas.

Por fim, durante a realização dos testes, percebeu-se uma grande perda de eficiência computacional quando a maioria dos processos eram concentrados em apenas um computador. Após examinar o fenômeno, concluiu-se que esta ocorrência dava-se quando três *cores*, ou mais, de um mesmo computador eram utilizados. Essa decorrência é evidenciada pelos resultados na Tabela 6.6, que compara os tempos resultantes da execução do problema com as três malhas em ambiente *multicore* e

distribuído. Essa discrepância de tempos decorre devido à grande competição dos *cores* pelos recursos da memória e dos barramentos, conforme discutido brevemente na seção 2.1.2.3. O fato de haver instaladas apenas duas memórias em cada computador, apesar de existirem 4 *sockets*, contribui para a grande perda de eficiência das máquinas. Este efeito, como é demonstrado nas seguintes seções, gerou uma rápida deterioração do ganho com a computação paralela. Isso porque, mesmo com a diminuição do número de nós/elements/graus de liberdade com o aumento de *cores*, o efeito do engarrafamento nos recursos do computador torna-se cada vez mais pronunciado com mais processadores. Assim, o tempo de execução dos testes acabaram sendo distorcidos, uma vez que o tempo medido representa, basicamente, o tempo de interferência entre processadores. Além disso, esta decorrência mostrou-se bastante significativa quando o modelo utilizado é muito grande e demanda muita atividade computacional.

Tabela 6.6: Tempos para a execução do método *executeParallel()* com três *cores* no mesmo computador (*multicore*) e com três computadores (multicomputador)

	Malha 1	Malha 2	Malha 3
Multicore	7,59 s	17,52 s	98,80 s
Multicomputador	6,95 s	15,03 s	67,61 s

Portanto, percebeu-se que, no caso deste *cluster*, existem dois efeitos que podem reger a eficiência de um programa paralelo: a comunicação entre computadores (internodal) e o uso da memória/barramento. Deste modo, os exemplos com 2 *cores* têm resultados melhores quando realizados em apenas um computador, ou seja, sem o uso da LAN. Com mais de 2 processos o custo da comunicação em rede é inferior ao custo da disputa de memória entre os processadores de uma mesma máquina. Devido a importância deste efeito, os testes com 3, 6, 12, 18, 24, 30 e 36 *cores* utilizam as três máquinas, enquanto que os testes com 2 *cores* foram realizados em apenas um

computador. É claro que, à medida que vão sendo utilizados mais processadores em todas as máquinas, novamente e inevitavelmente, vão surgindo os efeitos de congestionamento da memória. Uma provável solução para esta ineficiência seria a instalação de mais memória RAM, de forma que todos os *slots* sejam usados.

Reunindo essas informações, decidiu-se sintetizar os resultados evidenciando diferentes partes da implementação para realizar uma avaliação justa. Isso porque, ao ponderar apenas os tempos totais resultantes da implementação paralela, sem quantificar a atuação individual de cada método, chega-se a conclusões incompletas sobre o potencial do programa. Portanto, além dos tempos totais do método *execute()* da classe *SteadyStateParallel* juntamente com o cálculo do *speedup*, são detalhados os tempos de cada método chamado na aplicação paralela, e, com isto, são calculados, também, os *speedups*. Também, achou-se necessária a comparação de tempos das etapas unicamente relativas ao processamento da solução (montagem de vetores e matrizes e solução). Assim avaliou-se o possível *speedup* caso não fosse utilizado um programa paralelo à parte, que faz necessária a comunicação interprogramas e serialização do modelo.

6.3 Considerações Sobre a Leitura de Arquivos

À medida que foi-se utilizando malhas maiores, percebeu-se uma grande dificuldade na leitura dos arquivos de entrada a partir da classe *Persistence*. Para viabilizar a quantidade de testes que deveriam ser executados, examinou-se as possíveis fontes de ineficiência do código e dois problemas foram encontrados.

O primeiro problema identificado era uma repetição de código desnecessária, que realizava várias vezes a leitura do mesmo arquivo de entrada. Corrigiu-se este problema criando uma variável que guarda o conteúdo do arquivo de entrada, podendo ser acessada pelos métodos da classe *Persistence*.

O segundo problema, e mais importante, ocorria na tarefa de localização dos nós

incidentes nos elementos. Os nós, que são identificados por um número tanto no arquivo quanto no objeto Java, são guardados no modelo através de uma lista. Para gerar a referência dos nós de incidência em cada elemento, deve-se fazer uma pesquisa nesta lista, sendo a chave de busca o número do nó. Contudo, o tempo consumido nesta busca para cada incidência de cada elemento torna-se impraticável em uma lista contendo uma quantidade considerável de nós. Assim, para acelerar esta busca, aproveitou-se que o gerador de malha numera os nós de forma que coincide com sua posição na lista. Uma nova *tag*, *Indexed*, foi adicionada ao XML, tendo a função de indicar à classe *Persistence* que os nós estão indexados com relação à lista no modelo. Desta forma, a busca pela referência dos nós torna-se extremamente rápida e simples, ocorrendo diretamente pelo índice da lista.

Na Tabela 6.7 estão listados os tempos médios obtidos para a leitura de cada malha com nenhuma melhoria, com a melhoria no XML, ou seja, sem a repetição da leitura do arquivo, com a melhoria na leitura das incidências de nós nos elementos e, por último, com as duas otimizações. Ao observar os resultados, percebe-se que com as duas alterações o tempo de geração do modelo passa por uma considerável melhora, principalmente no caso da malha 3.

Tabela 6.7: Média e desvio padrão dos tempos na leitura de arquivos

Melhoria		Nenhuma	XML	Incidências	XML+Incidências
Malha 1	Média	23,64 s	19,87 s	8,90 s	3,57 s
	Desvio padrão	0,23 s	0,34 s	0,10 s	0,16 s
Malha 2	Média	86,93 s	70,74 s	13,99 s	5,13 s
	Desvio padrão	7,98 s	3,95 s	0,24 s	0,10 s
Malha 3	Média	975,56 s	956,21 s	29,36 s	10,17 s
	Desvio padrão	107,37 s	81,10 s	0,39 s	0,16 s

6.4 Resultados da Malha 1

O tempo total para o método *execute()* da análise estática linear com a primeira malha, de 1 a 36 *cores*, pode ser visto na Figura 6.7, enquanto que os detalhes de tempo dos métodos originados apenas do algoritmo paralelo são mostradas na Figura 6.8. Assim como nos resultados mostrados do programa sequencial, predomina o método *update()*, que influencia grandemente o valor final do *speedup*. Este alcançou valores bastante próximos de 1, como mostrado na Figura 6.9 e na Tabela 6.8, já que grande parte do processamento trata-se de um código sequencial que não foi paralelizado. Um mínimo ganho com o aumento de processadores foi alcançado utilizando 24 *cores*, totalizando um *speedup* de 1,07.

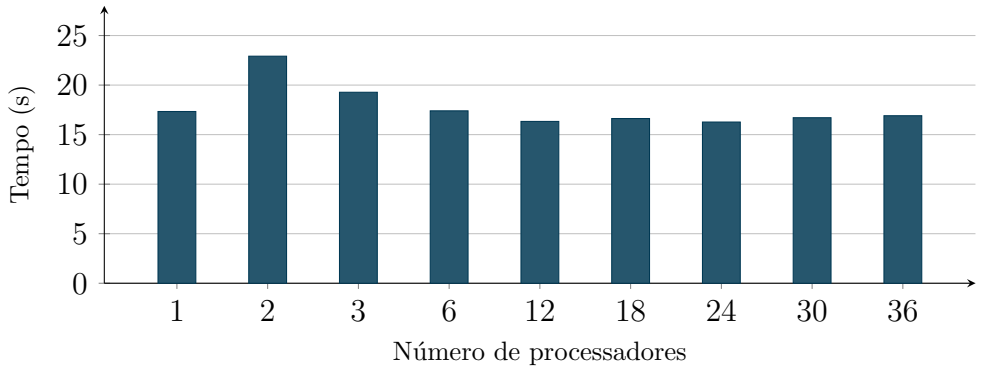


Figura 6.7: Tempos totais do método *execute()* das implementações sequencial e paralela com a malha 1

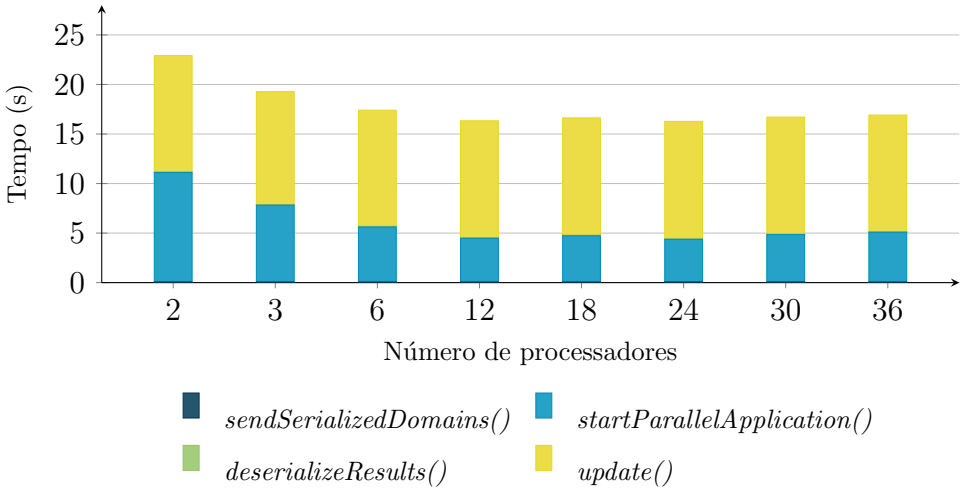


Figura 6.8: Tempos detalhados do método *execute()* na implementação paralela com a malha 1

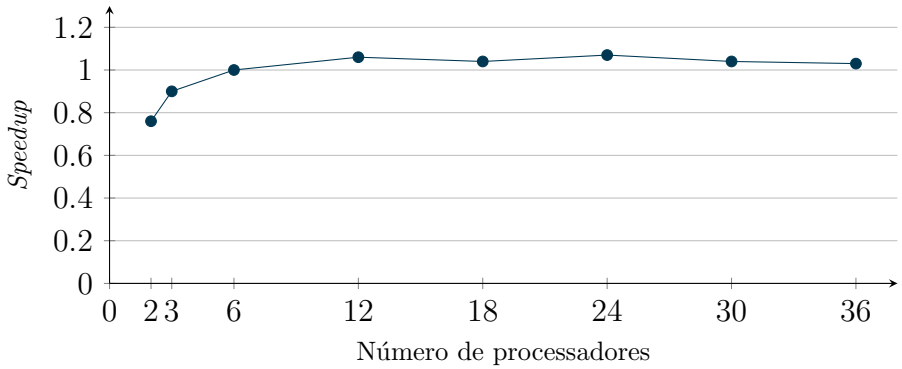


Figura 6.9: Resultados de *speedup* do método *execute()* com a malha 1

Tabela 6.8: Valores de *speedup* do método *execute()* com a malha 1

Cores	2	3	6	12	18	24	30	36
Speedup	0,76	0,90	1,00	1,06	1,04	1,07	1,04	1,03

Devido à interferência do *update()*, decidiu-se que poderia ser mais elucidativo avaliar, também, os tempos sem contar esta etapa. Portanto, detalhou-se a execução da aplicação paralela, *ParallelSolutionApplication*, na Figura 6.10. Para tanto, foram utilizados os valores de tempos do *rank* 0, isso, porque este sempre é o último a finalizar o programa, sendo, portanto, o mais demorado.

Com o uso de poucos *cores*, nota-se que boa parte do processamento ocorre na desserialização do problema, isto é, o método *getParallelSolution()*. Ao avaliar a causas da ineficiência desta etapa, percebeu-se que o problema estava na busca da incidência dos nós nos elementos. Assim como ocorria na leitura do arquivo de entrada, o algoritmo simples de busca por cada nó incidente de cada elemento através de um *loop* na lista de nós, e não pelos índices da lista, é extremamente ineficiente para domínios que ainda são bastante grandes. Portanto, com o acréscimo de processadores, esta etapa passa por uma considerável redução já que a lista de nós decresce devido à diminuição do tamanho dos domínios.

Também, ainda na Figura 6.10, vê-se que o método *sendResults()* praticamente não representa uma perda no processo. Em contrapartida, *gatherResults()* passa a contribuir no aumento do tempo à medida que adicionam-se mais processos, provavelmente devido ao acréscimo de comunicação gerada pelos vários *ranks*. Ainda assim, não pode-se dizer que as chamadas MPI oneraram muito o tempo da análise, preponderando realmente a desserialização dos objetos. Infelizmente, a etapa *executeParallel()* não mostra muitos ganhos com o aumento de *cores*, ficando com um resultado de tempo bem aquém da solução serial.

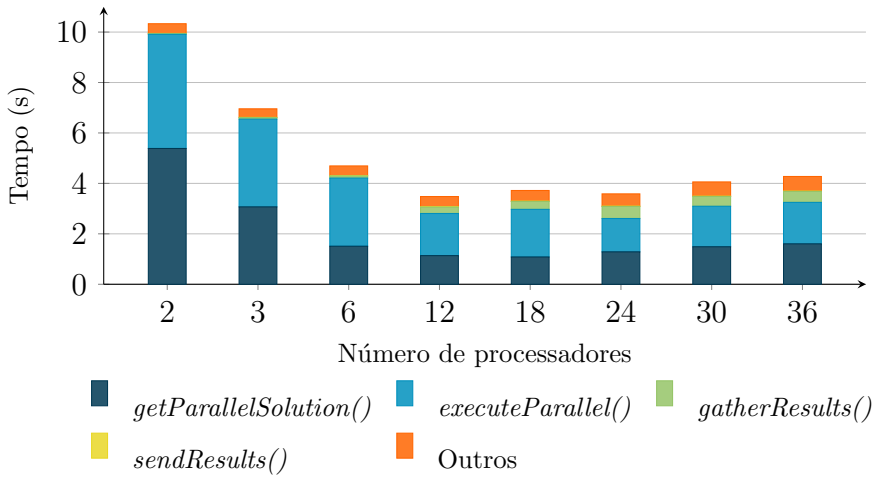


Figura 6.10: Detalhes do tempo de execução do *ParallelSolutionApplication* com a malha 1

O *speedup* obtido ao comparar o tempo de solução do problema sequencial (sem a fase de formação do arquivo de saída) com a execução do programa paralelo é mostrado na Figura 6.11 e na Tabela 6.9. Percebe-se como este índice melhora sem a penalidade gerada pelo método *update()*, alcançando um máximo de 1,76 com 12 cores, mas ainda sendo bastante ineficiente em contraposição ao *speedup* ideal. A partir dos 12 processos constata-se o problema com o barramento da memória, que gera uma estabilização do *speedup* para valores próximos de 1,6, mesmo com o aumento do número de processadores.

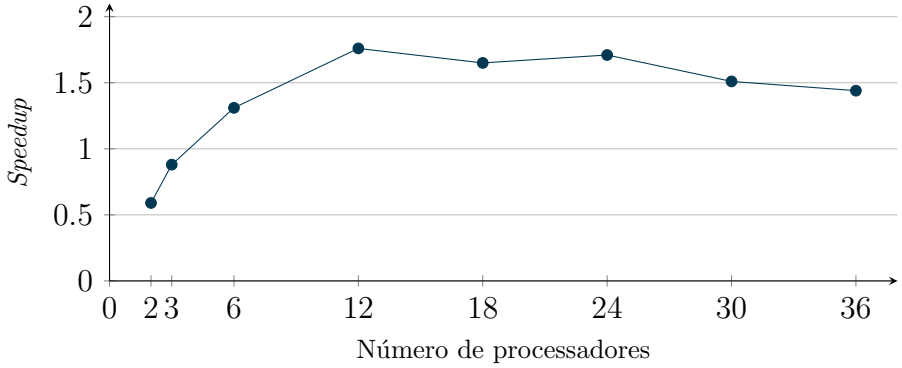


Figura 6.11: *Speedup* da execução do *ParallelSolutionApplication* com a malha 1 de 2 a 36 *cores*

Tabela 6.9: Valores de *speedup* para a execução do *ParallelSolutionApplication* na malha 1 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	0,59	0,88	1,31	1,76	1,65	1,71	1,51	1,44

Pensando em como poderia comportar-se o programa caso fosse completamente paralelizado, ou seja, sem usar o programa do *ParallelSolutionApplication* para executar a solução paralela à parte, separaram-se os tempos de execução apenas da etapa *executeParallel()*. Isso, porque é neste método que ocorrem, efetivamente, a

montagem, resolução do sistema de equações e cálculo de reações. Com isto, tornou-se mais fácil a visualização do ganho real com a paralelização de cada uma destas etapas.

Os gráficos da Figura 6.12 e 6.13 foram obtidos usando de referência o processador com a maior média de tempo para o método *executeParallel()*, podendo, ou não, ser o *rank* 0. Esta escolha ocorreu devido a variação de tempos de *rank* para *rank*, principalmente porque alguns processos tinham que calcular valores de reações e outros não, gerando uma considerável discrepância de tempos.

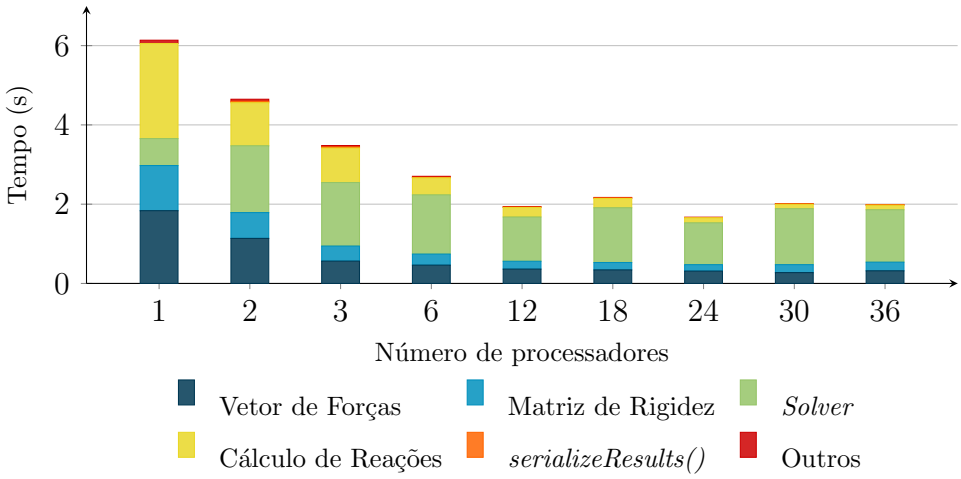


Figura 6.12: Detalhes do tempo de execução do método *executeParallel()* com a malha 1

As etapas de formação do vetor de forças e cálculo das reações, que no programa sequencial são as etapas mais onerosas, apresentaram bastante redução com o aumento no número de processadores, assim como a montagem da matriz de rigidez. Contudo, a resolução do sistema de equações não conseguiu atingir tempos menores com a paralelização e nem apresentou reduções significativas com o uso de mais processadores.

Entendeu-se que, além do problema no barramento da memória citado no início do capítulo, isso ocorreu porque o problema utilizado não possui graus de liberdade

suficientes. Segundo Balay et al. (2019*b*), é aconselhado utilizar a PETSc em problemas com pelo menos 10 mil variáveis desconhecidas por processador, sendo o valor ideal 20 mil ou mais. Portanto, esta malha não seria adequada por possuir apenas 50 mil variáveis.

Outro ponto importante é que a comparação ocorre entre dois solcionadores que funcionam com princípios bastante distintos. Enquanto que a versão distribuída utiliza o Gradientes Conjugados, um método iterativo, a versão sequencial utiliza o método Multifrontal, que é uma técnica direta de resolução de sistemas, sendo esta comparação, talvez, injusta. Mesmo assim, ao não contar todas as etapas ineficientes criadas devido à serialização, distribuição e desserialização do problema e resultados, é possível alcançar um *speedup* de até 3,67 com 24 processos.

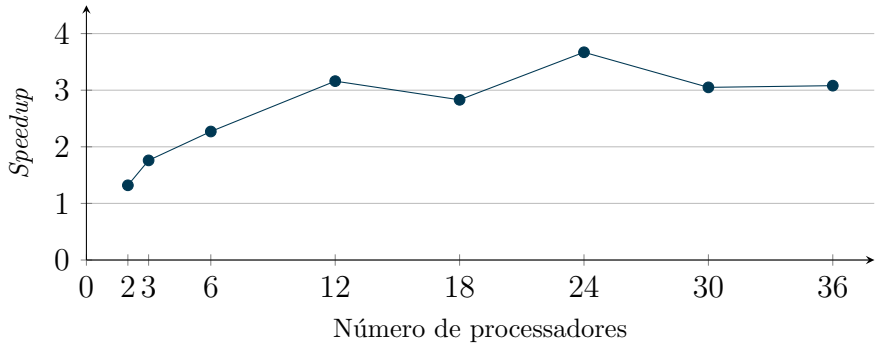


Figura 6.13: *Speedup* da execução do *executeParallel()* com a malha 1 desconsiderando o método *update()*

Tabela 6.10: Valores de *speedup* para a execução do *executeParallel()* com a malha 1 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	1,36	1,76	2,27	3,15	2,83	3,67	3,04	3,08

Ainda, percebendo como o resultado fraco do solucionador encobriu o ganho da paralelização com outras fases da solução, montou-se o *speedup* individual de cada

uma destas. Assim, na Figura 6.14, é possível verificar como cada etapa saiu-se individualmente.

Para tanto, foram selecionados os maiores tempos médios de cada uma destas etapas, independentemente de qual *rank* se tratava. A partir da linha de referência, que mostra os valores do caso de um *speedup* ideal, é possível constatar que, com exceção do *solver*, as outras etapas alcançaram valores de *speedup* maiores, mas ainda longe do ideal. Também, o cálculo do vetor de forças e a montagem da matriz de rigidez parecem possuir um patamar máximo em torno de 5. Como estas não possuem qualquer comunicação, deveria-se esperar pelo menos valores próximos do *speedup* ideal. Como a estagnação do valor ocorreu aos 12 *cores*, isto é, 4 cores para cada computador, conclui-se que, novamente, a grande concorrência pela memória prejudicou a performance. Ou seja, por mais que estas etapas estejam mais rápidas, a dificuldade para acessar a memória predomina no valor total do tempo.

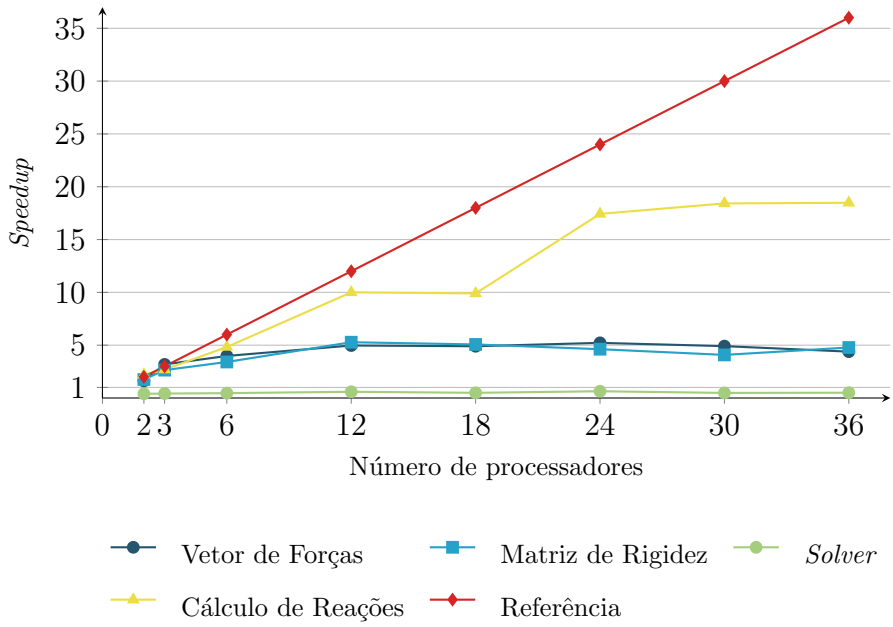


Figura 6.14: Valores de *speedups* individuais de cada etapa da solução para a malha 1

Para finalizar, verificando o cálculo de reações, este alcançou valores muito melhores de *speedup*, sendo, inclusive, bastante perto do ideal em alguns pontos. Existem duas explicações para este ocorrido. Primeiro, esta etapa requer a formação de uma matriz cheia C_{pp} que, diferentemente da parcela C_{uu}, não é esparsa e não tem o alocamento dos recursos de memória realizados previamente, onerando, de certa forma, o método. Assim, quando os graus de liberdade são divididos entre os processadores, o tamanho da matriz C_{pp} diminui e, desta forma, menor é o tempo gasto com a requisição de espaço na memória. Contudo, acredita-se que o motivo que mais deve ter influenciado este resultado é, outra vez, a disputa pelo acesso à memória. Como o próximo método a ser executado é uma chamada MPI síncrona, *gatherResults()*, os *ranks* que não calculam reações ficam inativos à espera pelos outros para realizar a comunicação. Desta forma, a memória deixa de ser acessada por todos os processos, somente sendo requisitada pelos *ranks* que calculam reações.

Tabela 6.11: Valores individuais de *speedup* para cada etapa da solução com malha 1

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	Reações
2	1,62	1,75	0,40	2,21
3	3,18	2,64	0,42	2,69
6	3,98	3,41	0,46	4,83
12	4,97	5,28	0,59	10,00
18	4,90	5,07	0,49	9,90
24	5,22	4,62	0,64	17,43
30	4,91	4,08	0,48	18,42
36	4,39	4,78	0,50	18,48

6.5 Resultados da Malha 2

Os tempos coletados para todos os cores, de 1 a 36, do método *execute()* com a malha 2 são apresentados na Figura 6.15, e maiores detalhes da execução paralela na

Figura 6.16. Assim como no exemplo anterior, o método *update()* prejudica bastante o cálculo do *speedup*, como é detalhado na Figura 6.17 e na Tabela 6.12. Neste caso, os valores continuaram baixos (próximos de 1), alcançando um *speedup* máximo de 1,16 com 18 *cores*.

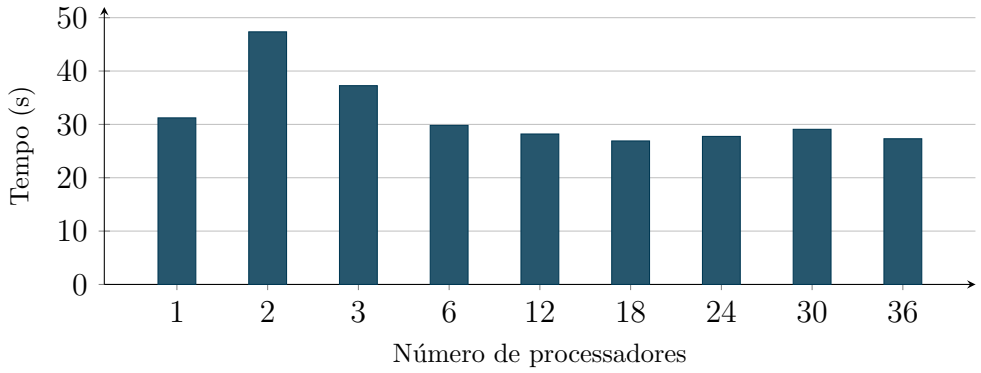


Figura 6.15: Tempos totais do método *execute()* das implementações sequencial e paralela com a malha 2

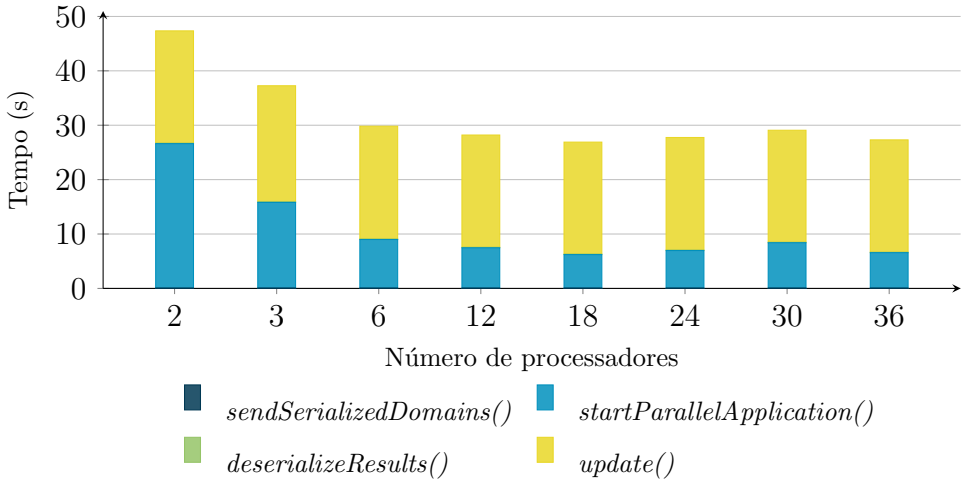


Figura 6.16: Tempos detalhados do método *execute()* na implementação paralela com a malha 2

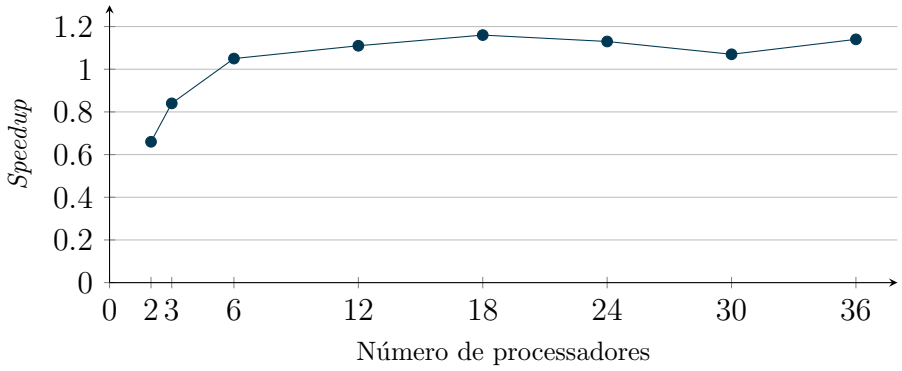


Figura 6.17: Resultados de *speedup* do método *execute()* com a malha 2

Tabela 6.12: Valores de *speedup* do método *execute()* com a malha 2

Cores	2	3	6	12	18	24	30	36
Speedup	0,66	0,84	1,05	1,11	1,16	1,13	1,07	1,14

Também foram determinados os valores de tempos gastos em cada uma das etapas executadas pelo *ParallelSolutionApplication*, Figuras 6.18 e 6.19. Grande parte da execução ocorreu na desserialização do modelo e, da mesma forma, também houve uma redução e estabilização do tempo que esta etapa é executada. Contudo, nesta malha, é a etapa do *executeParallel()* que passa a ser a mais onerosa e não *getParallelSolution()*. Desta vez o *speedup* máximo sofreu um aumento, como é mostrado na Tabela 6.13, tendo o valor de 2,08 com 18 cores.

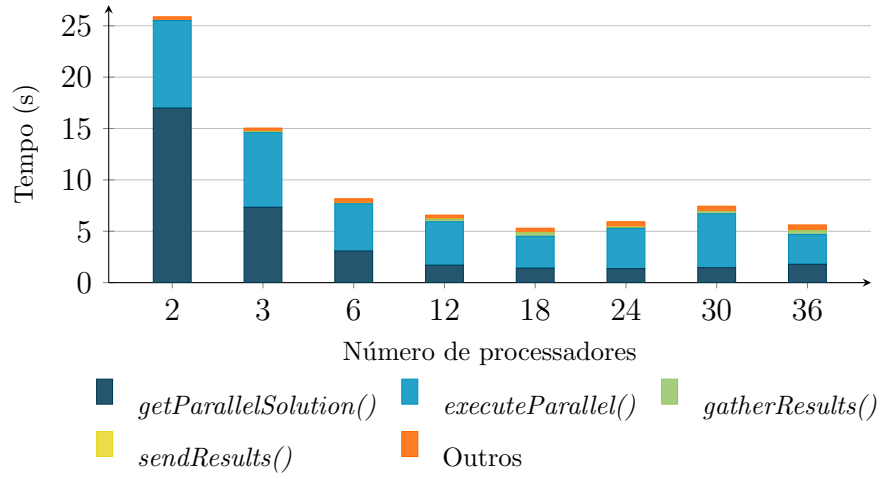


Figura 6.18: Detalhes do tempo de execução do *ParallelSolutionApplication* com a malha 2

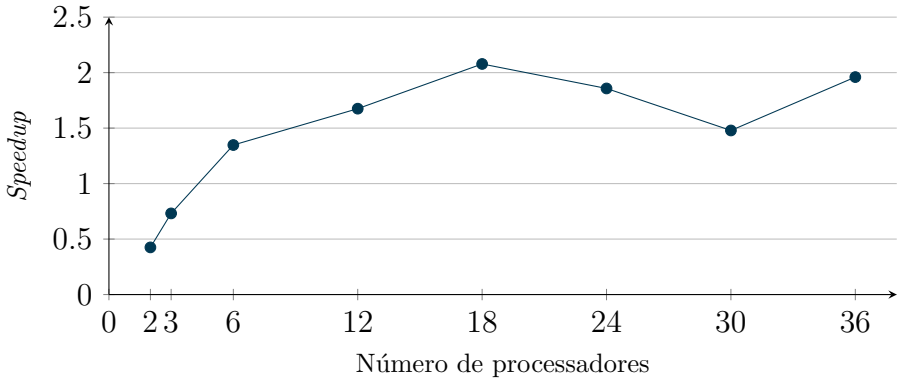


Figura 6.19: *Speedup* da execução do *ParallelSolutionApplication* com a malha 2

Tabela 6.13: Valores de *speedup* para a execução do *ParallelSolutionApplication* na malha 2 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	0,43	0,73	1,35	1,68	2,08	1,86	1,48	1,98

Observando os resultados do *executeParallel()*, na Figura 6.20, percebe-se o mesmo comportamento da malha 1. Há uma grande redução de tempo das etapas de montagem de vetor de forças e matriz de rigidez e cálculo de reações. Igualmente, o

solucionador ainda apresenta poucos ganhos, como mostrado na Figura 6.21, estabilizando seu tempo a partir dos 12 *cores*. Um *speedup* máximo de 3,21 foi alcançado com 36 processadores, mostrado na Tabela 6.14.

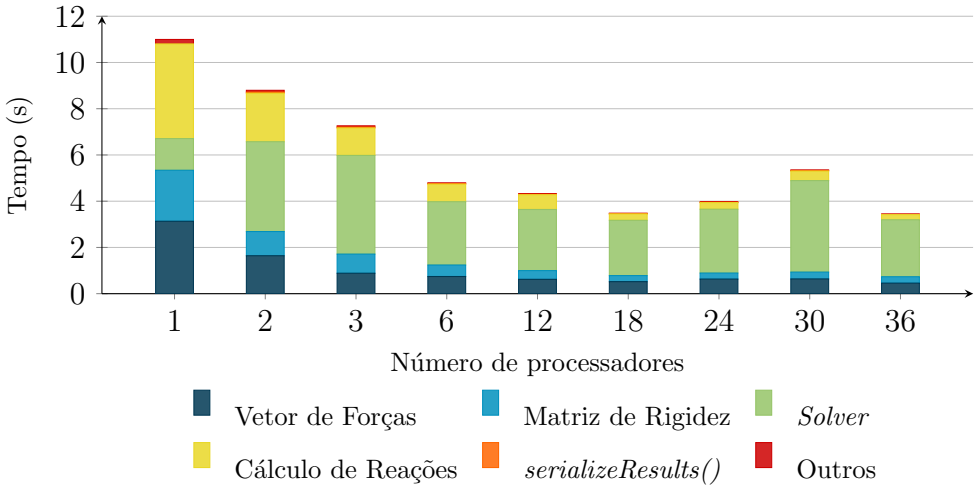


Figura 6.20: Detalhes do tempo de execução do *ParallelSolutionApplication* com a malha 2

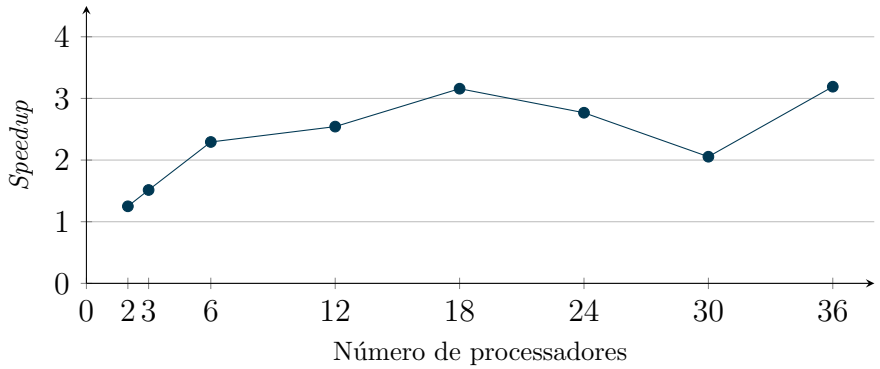


Figura 6.21: *Speedup* da execução do *executeParallel()* com a malha 2 desconsiderando o método *update()*

Tabela 6.14: Valores de *speedup* para a execução do *executeParallel()* com a malha 2 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	1,25	1,51	2,29	2,54	3,16	2,77	2,05	3,21

No *speedup* individual das etapas de montagem e solução do problema, percebe-se o mesmo comportamento da malha anterior, sendo o solucionador e o engarrafamento do barramento da memória os maiores impedimentos para um valor de *speedup* maior. A estabilização do *speedup* para a montagem da matriz e vetor de força ocorreu igualmente com 12 processos com valores próximos de 5 e 6.

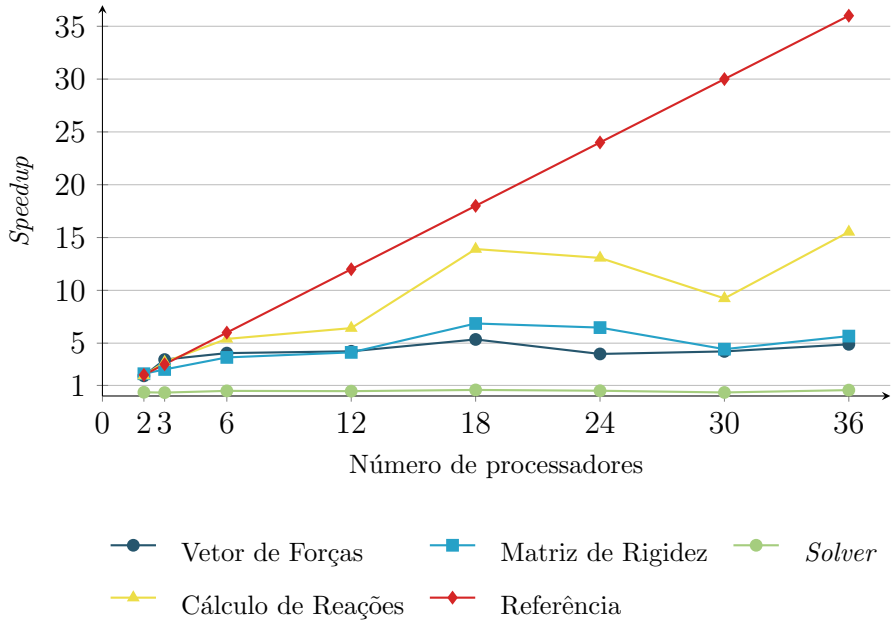


Figura 6.22: Valores individuais de *speedup* para cada etapa da solução com malha 2

Tabela 6.15: Valores de *speedup* individuais de cada etapa da solução para a malha 2

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	Reações
2	1,91	2,11	0,35	1,93
3	3,45	2,51	0,32	3,22
6	4,06	3,65	0,48	5,40
12	4,23	4,13	0,46	6,43
18	5,35	6,86	0,57	13,91
24	3,98	6,48	0,50	13,07
30	4,22	4,43	0,33	9,24
36	4,89	5,67	0,56	15,53

6.6 Resultados da Malha 3

A terceira malha apresentou resultados bastante ruins em comparação às anteriores, como observa-se na Figura 6.23 e 6.24. Entretanto, o que contribuiu tanto para este resultado negativo foi o *Garbage Collector* do Java, atuando durante o *update()*. Enquanto que para um processador a escrita do arquivo teve um tempo médio de 42 segundos, no programa paralelo o tempo médio foi de 65 segundos. Isto ocorreu porque, além do arquivo de entrada ser muito grande, a memória total foi dividida para alocar vários processos, restando uma pequena parcela para o programa principal. Portanto, o processador, ao necessitar memória para gerar o arquivo de saída com o *update()*, ativa o *Garbage Collector*, aumentando mais ainda o tempo do *execute()*. Uma forma bastante simples de resolver esta perturbação nos tempos seria utilizar mais memória, para que não seja necessário o acionamento do GC. Portanto, o *speedup* máximo do método *execute()*, mostrado na Figura 6.25 e na Tabela 6.16, foi de 0,83 com 18 cores.

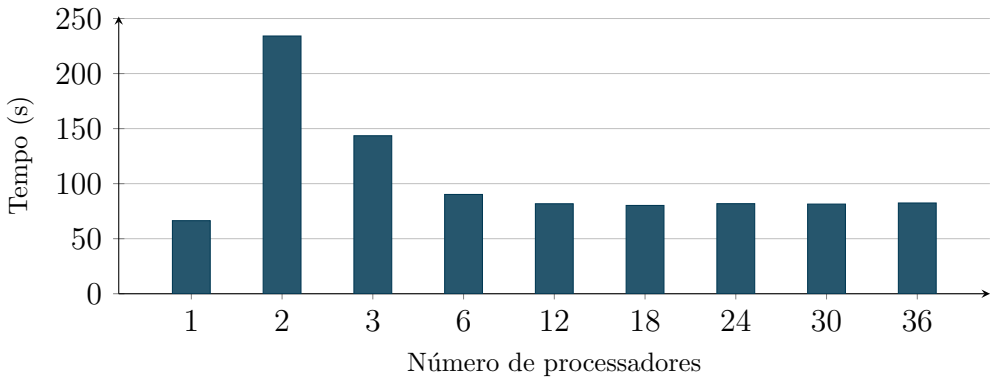


Figura 6.23: Tempos totais do método *execute()* das implementações sequencial e paralela com a malha 3

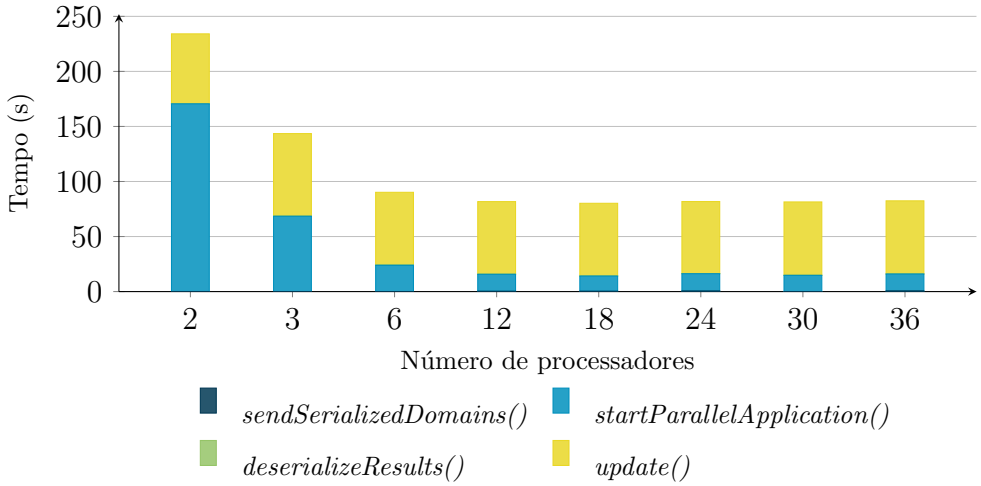


Figura 6.24: Tempos detalhados do método *execute()* na implementação paralela

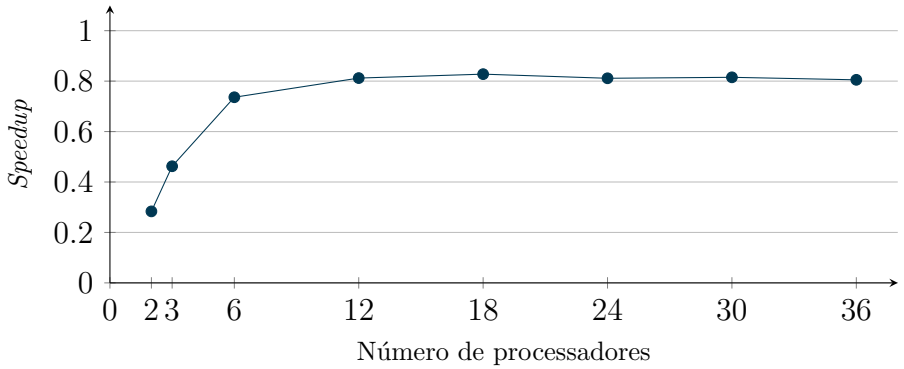


Figura 6.25: Resultados de *speedup* do método *execute()* com a malha 3

Tabela 6.16: Valores de *speedup* do método *execute()* com a malha 3

Cores	2	3	6	12	18	24	30	36
Speedup	0,28	0,46	0,74	0,81	0,83	0,81	0,82	0,81

Ao descontar os tempos referentes à criação do arquivo de saída, a perspectiva melhora bastante, conforme exposto nas Figuras 6.26 e 6.27. Contudo, percebe-se uma queda no *speedup*, mostrado na Figura 6.28 e na Tabela 6.17, chegando a um máximo de 1,85 com 18 processadores, enquanto que na malha 2 o maior valor foi de 2,08. Isso, porque o aumento do modelo piora muito o tempo da desserialização do problema, devido à busca dos nós incidentes nos elementos.

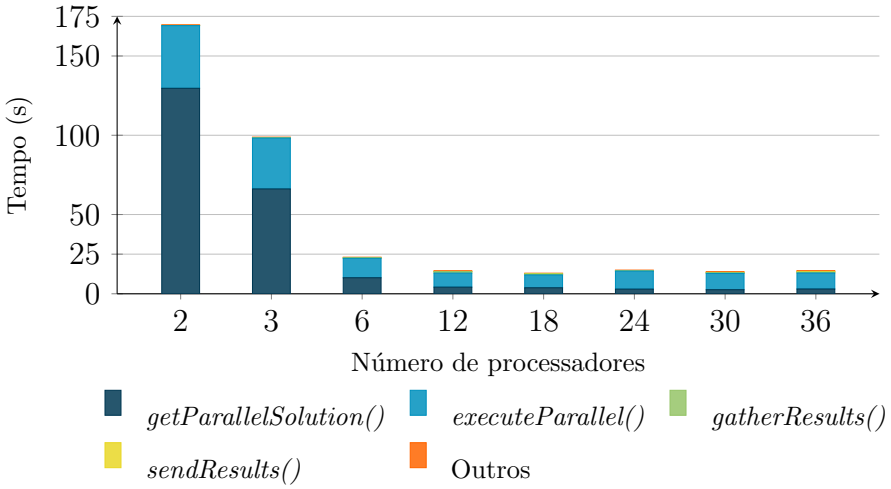


Figura 6.26: Detalhes do tempo de execução do *ParallelSolutionApplication* de 2 a 36 processadores com a malha 3

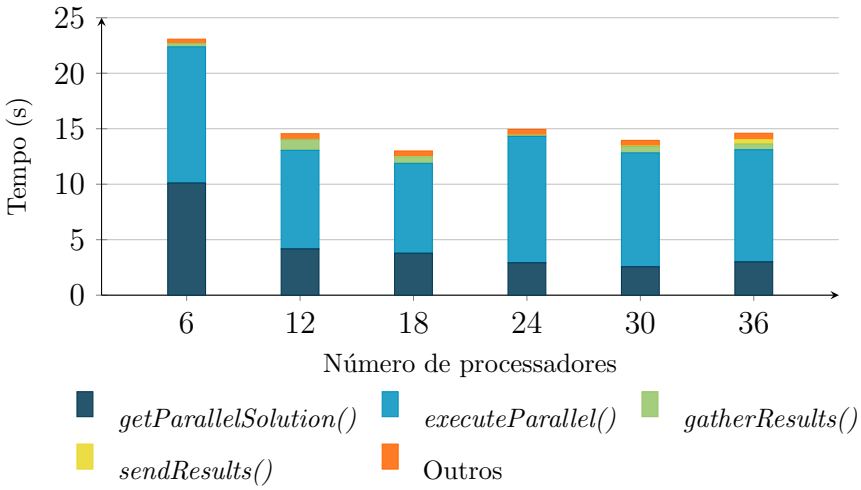


Figura 6.27: Detalhes do tempo de execução do *ParallelSolutionApplication* de 6 a 36 processadores com a malha 3

Tabela 6.17: Valores de *speedup* para a execução do *ParallelSolutionApplication* na malha 3 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	0,14	0,35	1,04	1,65	1,85	1,61	1,72	1,64

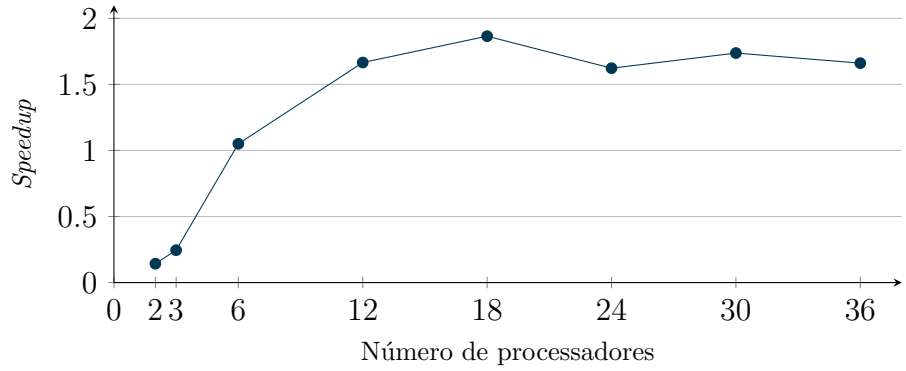


Figura 6.28: *Speedup* da execução do *ParallelSolutionApplication* com a malha 1 de 2 a 36 *cores*

Na Figura 6.29, pode-se averiguar a causa da ineficiência durante a solução do problema, que fica por conta do solucionador. Agora com um problema de 196 mil variáveis, este passou a demorar um tempo considerável. Contudo, houve uma queda considerável de tempo com o aumento dos processadores, mostrando uma boa escalabilidade da biblioteca PETSc, que, infelizmente, estabilizou nos 6 processadores devido aos problemas com o acesso à memória. Alcançou-se um *speedup* máximo de 2,66 para esta etapa, como pode-se ver pela Figura 6.30 e Tabela 6.18, novamente menor em relação ao valor de 3,21 obtido com a malha 2.

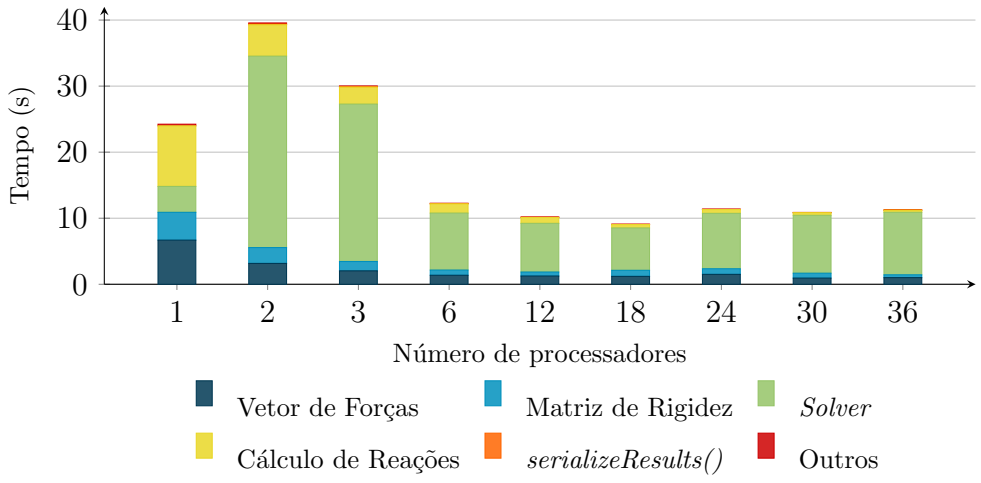


Figura 6.29: Detalhes do tempo de execução do método *executeParallel()* com a malha 3

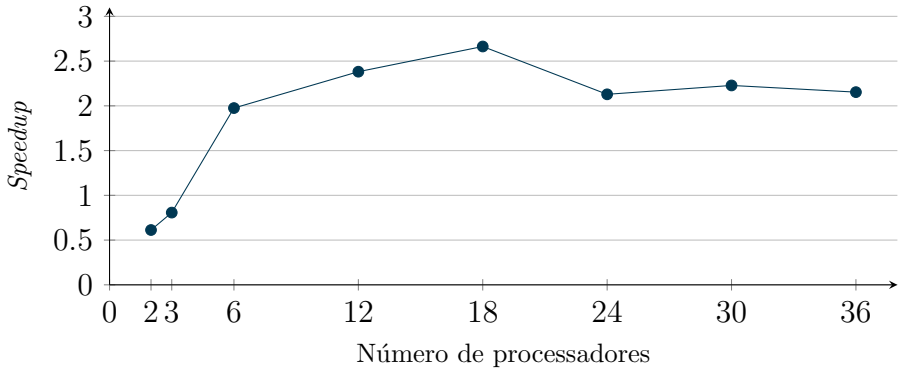


Figura 6.30: *Speedup* da execução do *executeParallel()* com a malha 3 desconsiderando o método *update()*

Tabela 6.18: Valores de *speedup* para a execução do *executeParallel()* com a malha 3 desconsiderando o método *update()*

Cores	2	3	6	12	18	24	30	36
Speedup	0,61	0,81	1,98	2,38	2,66	2,13	2,23	2,15

Avaliando, também, o valor dos *speedups* individuais de cada etapa da solução do problema, mostradas na Figura 6.31 e na Tabela 6.19, constata-se a mesma tendência das malhas anteriores. A montagem de vetor de forças e matriz de rigidez permanecem com um valor limite de *speedup* em torno de 5 a partir dos 12 processos, e o cálculo de reações permanece com valores mais altos devido ao fato da memória ser pouco disputada nesta etapa.

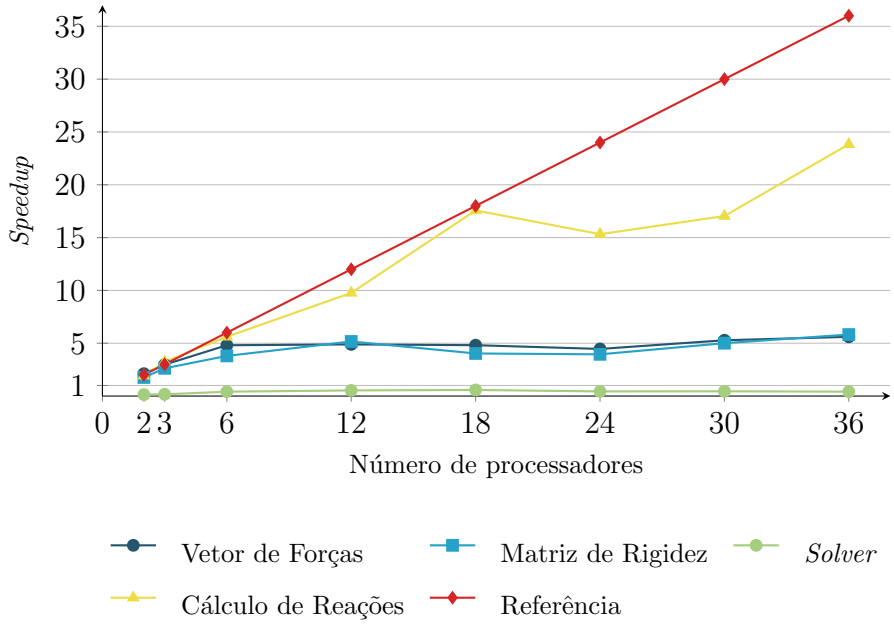


Figura 6.31: Valores individuais de *speedup* para cada etapa da solução com malha 3

Tabela 6.19: Valores de *speedup* individuais de cada etapa da solução para a malha 1

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	Reações
2	2,12	1,76	0,14	1,93
3	2,54	2,29	0,11	2,55
6	4,81	3,80	0,41	5,61
12	4,89	5,17	0,53	9,76
18	4,81	4,03	0,58	17,57
24	4,46	3,95	0,44	15,32
30	5,28	5,00	0,45	17,04
36	5,61	5,82	0,41	23,83

Capítulo 7

Considerações Finais

Neste trabalho apresentou-se uma proposta para a implementação de uma rotina paralela de análise estática linear em elementos finitos no sistema INSANE. A metodologia concebida para esta proposta de pesquisa mostrou-se viável, apresentando alguns resultados positivos. Contudo, foram constatados pontos que a proposição simplificada não mostrou-se ideal, além de algumas dificuldades com os recursos computacionais utilizados na sua avaliação.

A partir do constante aparecimento da ineficiência de divisão dos recursos de memória entre processadores, é possível concluir que a atual configuração do *cluster* utilizado nos testes torna impossível atingir o tempo paralelo ótimo com muitos processos. Para isso, seria necessário adicionar mais memória RAM nos outros *sockets* disponíveis. Ainda assim, é possível que esta solução não seja suficiente visto que existem 12 processadores para apenas 4 *slots* de memória. Porém, apesar desses aspectos, com a finalização deste tabalho, tornou-se possível a utilização do *software* INSANE em multicomputadores e, consequentemente, sua aplicação em análises muito grandes.

É claro que não pode-se responsabilizar a baixa performance do código apenas na configuração do *cluster*, pois mesmo com poucos processadores o *speedup* da implementação ficou bastante longe do ideal. Isto ocorreu principalmente devido à grande carga de trabalho adicionada devido à escolha de criar um programa paralelo em separado do sistema INSANE. Este efeito já era esperado, contudo entendeu-se

que era necessário para que houvesse um primeiro estabelecimento da computação distribuída no programa. Ainda assim, percebeu-se que grande parte da ineficiência ocorre na desserialização dos modelos, com a busca dos nós incidentes em cada elemento. Este problema é consideravelmente fácil de se consertar, portanto, devem existir boas oportunidades de otimizar ainda o programa. No entanto, esta experiência traz atenção ao fato de que a paralelização da leitura e escrita de arquivos é quase tão importante quanto a paralelização dos cálculos.

Ainda, com base nos resultados dos exemplos utilizados para testes, foi possível constatar grandes dificuldades da implementação sequencial para lidar com problemas grandes. Algumas melhorias foram estabelecidas, conquanto, ainda são necessárias mais investigações, principalmente para a criação dos arquivos de resposta. Também, como foram avaliadas as condições do programa ao escalar o tamanho do problema, as soluções utilizadas afetam principalemente os casos examinados pelo trabalho, e podem não ter efeitos significativos em outras configurações. Por isso, ainda são necessárias avaliações da execução de diferentes análises para manter o *software* funcional.

Também, estabeleceram-se padrões para o uso de solucionadores no *software*, que trouxeram melhorias no reuso de código, principalmente quando consideramos a adição de bibliotecas nativas. Com isso, criou-se uma base para manter os códigos criados com o uso do JNI, abrindo mais possibilidades para o uso de bibliotecas nativas em outros âmbitos, como geração de malhas ou interface gráfica, e evitando a perda destas implementações. Contudo, entende-se que ainda é essencial estabelecer um padrão de criação (*factory*) para as matrizes, e, dessa forma, generalizar ainda mais o uso da interface *Assembler* e suas matrizes.

Por último, notou-se que o programa faz um grande uso de memória, como, por exemplo, na leitura/escrita de arquivos ou, como citado anteriormente, com a montagem da matriz de rigidez dos elementos. Além disso, apesar de existir uma certa implementação de matrizes esparsas, estas servem apenas para a montagem

da matriz C_{uu} e o uso das bibliotecas nativas de *solvers*. Ou seja, não existem operações como multiplicação ou adição em matrizes esparsas e, por isso, elas não são utilizadas de modo mais geral. Esse fato mostrou-se um problema pois, ao testar um modelo com 746 mil graus de liberdade, o *software* sequencial somente não foi capaz de finalizar a análise pois a matriz C_{pp} não coube na memória por ser densa. Assim sendo, deixam-se sugestões para a continuação desta linha de pesquisa.

7.1 Sugestões para Trabalhos Futuros

Com base nas constatações ditas anteriormente, aconselha-se, para futuros trabalhos, que seja realizada, principalmente, uma implementação totalmente paralela, isto é, incluindo a leitura/escrita de arquivos. Com isto, surgem novos desafios de como incluir o ambiente paralelo ao *software*, tipos de estruturas de dados que favorecem um modelo em paralelo, como/quando realizar a divisão de domínios e como transferir objetos entre os computadores. Além destes pontos importantes, destacam-se, brevemente, alguns itens que ainda seriam pertinentes para acrescentar ao que já foi realizado.

1. Testar a implementação em outros computadores paralelos
2. Implementar a paralelização de análises não lineares
3. Estudar demais tipos de solucionadores, do PETSc ou outros, avaliando juntamente precondicionadores
4. Incluir metodologias de Decomposição de Domínios, como os métodos de Schwarz e Schur
5. Otimizar a serialização e desserialização dos objetos no sistema INSANE
6. Examinar o uso do *multithreading*
7. Explorar implementações com GPUs

8. Avaliar as causas de ineficiência da leitura e escrita dos arquivos, considerando, também, a utilização de um formato mais leve para problemas grandes

9. Utilizar matrizes esparsas para as outras parcelas da matriz de rigidez, adicionando mais funcionalidades destas

10. Testar e comparar uma implementação utilizando o *framework* Apache Spark

Apêndice A

Tempos Médios da Implementação

No capítulo 6 foram apresentados de forma mais visual, a partir de gráficos, os resultados da média de tempo adquiridos com a execução do programa. Portanto, são mostrados a seguir os valores médios adquiridos com o *profiling* do programa.

A.1 Tempo da Implementação Sequencial

Tabela A.1: Tempos para a execução sequencial dos exemplos utilizando o *solver* UMFPACK

Malha	1	2	3
Vetor de Forças	1,84 s	3,13 s	6,68 s
Matriz de Rigidez	1,14 s	2,21 s	4,21 s
Solver	0,68 s	1,37 s	3,92 s
Cálculo de Reações	2,40 s	4,09 s	9,17 s
update()	11,19 s	20,22 s	42,13 s
Outros	0,08 s	0,19 s	0,25 s
Total com *update()*	17,33 s	31,22 s	66,36 s
Total sem *update()*	6,14 s	11,00 s	24,23 s

A.2 Tempos da Implementação Paralela

Tabela A.2: Tempo total utilizado para o método *execute()* nos três tipos de malhas variando de 1 a 36 *cores*

Cores	Malha 1	Malha 2	Malha 3
1	17,33 s	31,22 s	66,37 s
2	22,92 s	47,35 s	234,11 s
3	19,28 s	37,26 s	143,52 s
6	17,40 s	29,82 s	90,17 s
12	16,33 s	28,20 s	81,73 s
18	16,63 s	26,89 s	80,19 s
24	16,27 s	27,75 s	81,81 s
30	16,71 s	29,08 s	81,43 s
36	16,91 s	27,31 s	82,43 s

Tabela A.3: Detalhes do tempo de execução do método *execute()* com a malha 1 variando de 2 a 36 *cores*

Cores	*sendSerializedDomains()*	*startParallelApplication()*	*deserializeResults()*	*update()*
2	0,06 s	11,05 s	0,06 s	11,74 s
3	0,06 s	7,76 s	0,06 s	11,40 s
6	0,06 s	5,56 s	0,07 s	11,72 s
12	0,06 s	4,41 s	0,07 s	11,79 s
18	0,06 s	4,67 s	0,06 s	11,84 s
24	0,06 s	4,30 s	0,06 s	11,84 s
30	0,06 s	4,78 s	0,06 s	11,80 s
36	0,06 s	5,02 s	0,07 s	11,76 s

Tabela A.4: Detalhes do tempo de execução do método *execute()* com a malha 2 variando de 2 a 36 *cores*

Cores	*sendSerializedDomains()*	*startParallelApplication()*	*deserializeResults()*	*update()*
2	0,05 s	26,56 s	0,08 s	20,65 s
3	0,05 s	15,75 s	0,11 s	21,34 s
6	0,05 s	8,94 s	0,08 s	20,74 s
12	0,05 s	7,41 s	0,08 s	20,65 s
18	0,05 s	6,16 s	0,08 s	20,60 s
24	0,05 s	6,91 s	0,08 s	20,70 s
30	0,05 s	8,35 s	0,08 s	20,60 s
36	0,05 s	6,52 s	0,08 s	20,66 s

Tabela A.5: Detalhes do tempo de execução do método *execute()* com a malha 3 variando de 2 a 36 *cores*

Cores	*sendSerializedDomains()*	*startParallelApplication()*	*deserializeResults()*	*update()*
2	0,05 s	170,46 s	0,16 s	63,44 s
3	0,05 s	68,34 s	0,17 s	64,29 s
6	0,05 s	23,76 s	0,65 s	65,71 s
12	0,30 s	15,38 s	0,17 s	65,89 s
18	0,31 s	13,75 s	0,16 s	65,97 s
24	0,56 s	15,66 s	0,16 s	65,42 s
30	0,05 s	14,66 s	0,17 s	66,54 s
36	0,62 s	15,32 s	0,15 s	66,34 s

Tabela A.6: Detalhes do tempo de execução da classe *ParallelSolutionApplication* com a malha 1 variando de 2 a 36 *cores*

Cores	*getParallelSolution()*	*executeParallel()*	*gatherResults()*	*sendResults()*	Outros	Total
2	5,38 s	4,53 s	0,03 s	0,01 s	0,38 s	10,33 s
3	3,07 s	3,48 s	0,06 s	0,01 s	0,34 s	6,96 s
6	1,51 s	2,70 s	0,08 s	0,01 s	0,39 s	4,69 s
12	1,14 s	1,67 s	0,26 s	0,01 s	0,41 s	3,48 s
18	1,08 s	1,89 s	0,31 s	0,01 s	0,42 s	3,72 s
24	1,29 s	1,32 s	0,48 s	0,01 s	0,49 s	3,58 s
30	1,49 s	1,61 s	0,39 s	0,01 s	0,57 s	4,06 s
36	1,60 s	1,64 s	0,44 s	0,01 s	0,58 s	4,27 s

Tabela A.7: Detalhes do tempo de execução da classe *ParallelSolutionApplication* com a malha 2 variando de 2 a 36 *cores*

Cores	*getParallelSolution()*	*executeParallel()*	*gatherResults()*	*sendResults()*	Outros	Total
2	16,98 s	8,52 s	0,00 s	0,01 s	0,35 s	25,87 s
3	7,33 s	7,26 s	0,10 s	0,01 s	0,39 s	15,04 s
6	3,07 s	4,60 s	0,03 s	0,01 s	0,45 s	8,16 s
12	1,69 s	4,22 s	0,24 s	0,01 s	0,39 s	6,56 s
18	1,40 s	3,11 s	0,34 s	0,01 s	0,43 s	5,29 s
24	1,36 s	3,92 s	0,11 s	0,01 s	0,53 s	5,92 s
30	1,45 s	5,24 s	0,21 s	0,02 s	0,52 s	7,44 s
36	1,78 s	2,91 s	0,36 s	0,01 s	0,56 s	5,61 s

Tabela A.8: Detalhes do tempo de execução da classe *ParallelSolutionApplication* com a malha 3 variando de 2 a 36 *cores*

Cores	*getParallelSolution()*	*executeParallel()*	*gatherResults()*	*sendResults()*	Outros	Total
2	129,60 s	39,56 s	0,05 s	0,02 s	0,42 s	169,66 s
3	66,12 s	39,56 s	0,06 s	0,02 s	0,38 s	98,81 s
6	10,11 s	12,27 s	0,29 s	0,01 s	0,39 s	23,07 s
12	4,18 s	8,88 s	0,97 s	0,01 s	0,51 s	14,55 s
18	3,78 s	8,10 s	0,59 s	0,02 s	0,51 s	12,99 s
24	2,92 s	11,38 s	0,09 s	0,01 s	0,52 s	14,93 s
30	2,56 s	10,26 s	0,59 s	0,02 s	0,51 s	13,95 s
36	3,01 s	10,10 s	0,51 s	0,41 s	0,56 s	14,59 s

Tabela A.9: Detalhes do tempo para a solução do problema com a malha 1 variando de 1 a 36 *cores*

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	Cálculo de Reações	serializeResults()	Outros	Total
1	1,84 s	1,14 s	0,68 s	2,40 s	0,00 s	0,08 s	6,14 s
2	1,12 s	0,65 s	1,69 s	1,07 s	0,04 s	0,05 s	4,65 s
3	0,56 s	0,38 s	1,60 s	0,87 s	0,03 s	0,04 s	3,48 s
6	0,46 s	0,28 s	1,50 s	0,43 s	0,01 s	0,03 s	2,70 s
12	0,37 s	0,19 s	1,12 s	0,24 s	0,01 s	0,02 s	1,95 s
18	0,34 s	0,19 s	1,38 s	0,23 s	0,02 s	0,01 s	2,17 s
24	0,31 s	0,16 s	1,06 s	0,13 s	0,01 s	0,01 s	1,67 s
30	0,22 s	0,20 s	1,41 s	0,11 s	0,01 s	0,01 s	2,02 s
36	0,32 s	0,22 s	1,32 s	0,11 s	0,01 s	0,01 s	1,99 s

Tabela A.10: Detalhes do tempo para a solução do problema com a malha 2 variando de 1 a 36 *cores*

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	*Cálculo de Reações*	serializeResults()	Outros	Total
1	3,13 s	2,21 s	1,37 s	4,09 s	0,00 s	0,19 s	11,00 s
2	1,64 s	1,05 s	3,89 s	2,10 s	0,05 s	0,08 s	8,80 s
3	0,88 s	0,83 s	4,27 s	1,19 s	0,04 s	0,05 s	7,26 s
6	0,74 s	0,50 s	2,74 s	0,76 s	0,03 s	0,03 s	4,79 s
12	0,62 s	0,38 s	2,64 s	0,64 s	0,02 s	0,03 s	4,32 s
18	0,52 s	0,26 s	2,39 s	0,27 s	0,01 s	0,02 s	3,48 s
24	0,63 s	0,26 s	2,76 s	0,29 s	0,02 s	0,02 s	3,97 s
30	0,64 s	0,30 s	3,96 s	0,41 s	0,03 s	0,02 s	5,35 s
36	0,45 s	0,28 s	2,46 s	0,22 s	0,01 s	0,02 s	3,45 s

Tabela A.11: Detalhes do tempo para a solução do problema com a malha 3 variando de 1 a 36 *cores*

Cores	Vetor de Forças	Matriz de Rigidez	*Solver*	*Cálculo de Reações*	serializeResults()	Outros	Total
1	6,68 s	4,21 s	3,92 s	9,17 s	0,00 s	0,25 s	24,23 s
2	3,16 s	2,40 s	28,98 s	4,75 s	0,10 s	0,18 s	39,56 s
3	2,03 s	1,42 s	23,82 s	2,57 s	0,06 s	0,12 s	30,03 s
6	1,36 s	0,79 s	8,63 s	1,39 s	0,04 s	0,06 s	12,27 s
12	1,25 s	0,61 s	7,37 s	0,89 s	0,03 s	0,04 s	10,18 s
18	1,19 s	0,92 s	6,41 s	0,51 s	0,02 s	0,04 s	9,10 s
24	1,50 s	0,85 s	8,39 s	0,58 s	0,02 s	0,05 s	11,38 s
30	0,93 s	0,75 s	8,76 s	0,39 s	0,01 s	0,04 s	10,88 s
36	1,01 s	0,43 s	9,46 s	0,32 s	0,01 s	0,03 s	11,25 s

Apêndice B

Padrão de implementação JNI no INSANE

Duas partes importantes do trabalho foram o desenvolvimento com bibliotecas nativas e o padrão elaborado para o uso destas no sistema INSANE. Portanto, este apêndice tem como finalidade expor, para futuros trabalhos do grupo de pesquisa, como foi estabelecida a implementação JNI (*Java Native Interface*) destas bibliotecas.

Aplicações nativas são programas/rotinas escritos em linguagens de programação nativas, como C e C++, que devem ser compiladas em código de máquina para um ambiente específico. Um ambiente específico é o conjunto de especificações de um computador, como o sistema operacional, o *hardware* e suas bibliotecas nativas. Felizmente no caso das duas maiores fabricantes de processadores, Intel e AMD, existe uma boa compatibilidade de instruções, sendo, geralmente, possível a portabilidade de programas, desde que o Sistema Operacional seja o mesmo.

Já no caso da linguagem de programação Java, que é dita portável, ao invés de compilar o programa diretamente para uma linguagem de máquina, que é específica do ambiente em que está sendo compilado, compila-se, primeiramente, em *bytecode*. As instruções *bytecode* são uma criação da plataforma Java que precedem a formação das instruções de máquina. Assim, para um programa Java ser executado, os códigos em *bytecode* são lidos pela Máquina Virtual Java (*Java Virtual Machine* ou JVM), que traduz estes para o código de máquina do computador em específico.

Dessa forma, as instruções de máquina são produzidas a medida que o programa é executado. Portanto, a portabilidade de um programa em Java somente depende da instalação prévia do JRE (*Java Runtime Environment*), que inclui a JVM.

Conforme Liang (1999) explica, o JNI é uma interface de duas mãos, que permite aplicações Java acionarem programas nativos e vice-versa. Ou seja, um programa Java pode chamar funções que são implementadas por linguagens nativas, bem como programas nativos podem iniciar uma Máquina Virtual (*Virtual Machine*) pra chamar métodos em Java.

Para tanto, cria-se um método Java, responsável pela associação com o código nativo, e uma biblioteca dinâmica que contém os métodos nativos. Ambos devem ser elaborados de forma que o padrão JNI seja estritamente seguido. Assim, se o desenvolvedor implementar corretamente a especificação, a JVM consegue identificar e chamar os métodos nativos adequadamente. Consequentemente, o desenvolvedor deve ter em mente que é sua obrigação criar e testar a biblioteca nativa para um ou vários ambientes específicos já que não existe garantia de portabilidade do JNI assim como em Java.

Para facilitar a explicação do projeto de JNI do INSANE, utilizou-se como exemplo o módulo do solucionador nativo UMFPACK da biblioteca SuiteSparse. O projeto foi criado com a ferramenta de gerenciamento Maven.

Na Figura B.1 é mostrada a organização final do módulo, contendo uma pasta para os arquivos para o programa Java (**src/main**), outra pasta para a rotina C que chama a biblioteca nativa (**jni**), duas pastas contendo os arquivos da biblioteca nativa (*headers* e bibliotecas estáticas) compilados para Linux e Windows e, por último, uma pasta que guarda o código fonte da biblioteca nativa. Para a compilação existem dois mecanismos, o arquivo *makefile*, que guarda as regras de compilação do código nativo, e o arquivo **pom.xml**, da ferramenta Maven que gerencia a execução da compilação tanto dos arquivos Java quanto dos arquivos nativos.

A seguir é detalhado o passo-a-passo da construção do módulo e do JNI.

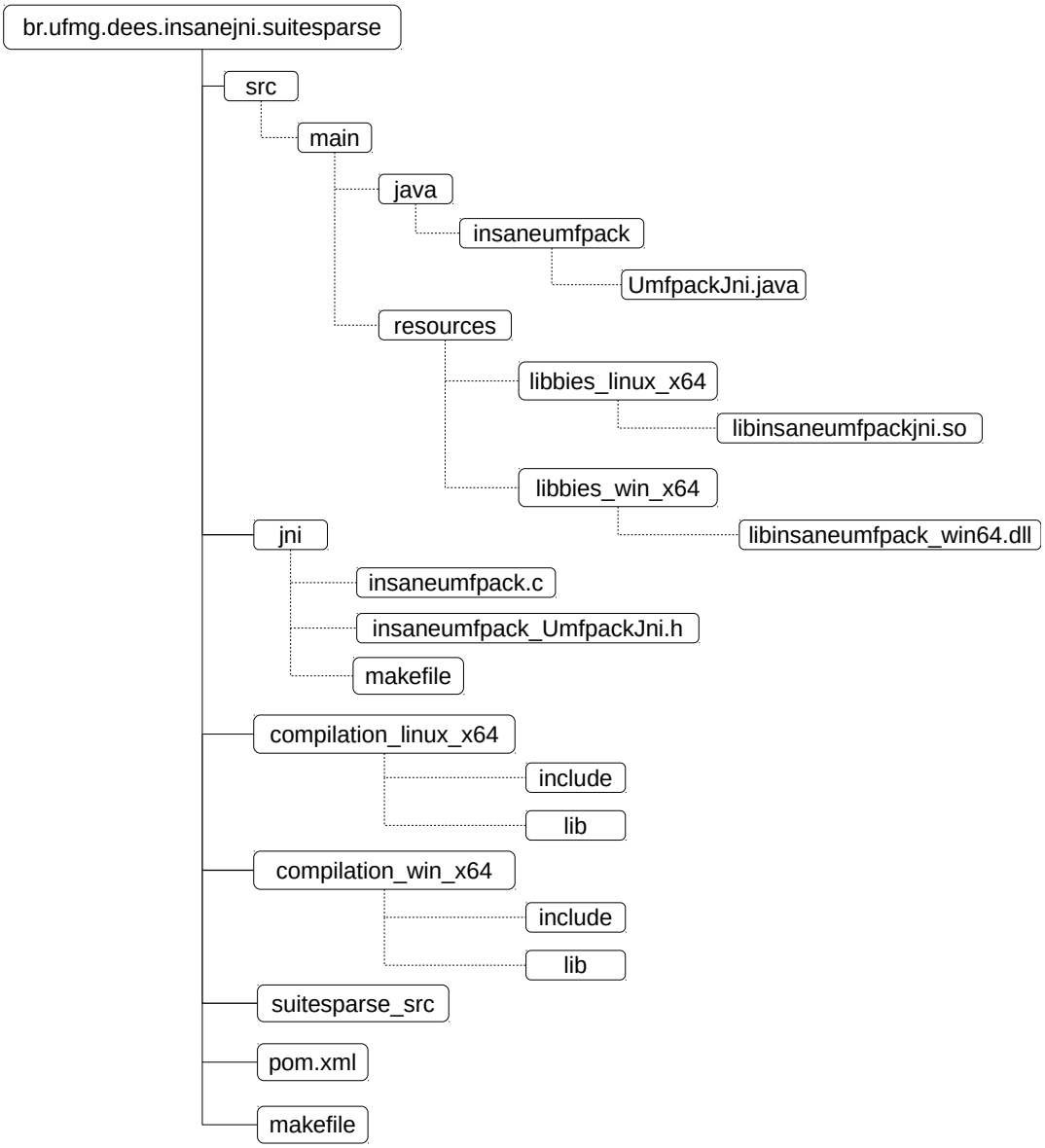


Figura B.1: Organização de um módulo no projeto JNI no sistema INSANE

B.1 Primeiro Passo

Primeiramente cria-se a classe Java que irá realizar as chamadas nativas. Para isto, após já estabelecer um novo módulo Maven no projeto, que pode ser feito automaticamente por alguma IDE, dentro do diretório *src/main/java* é criado um pacote, no exemplo nomeado de *insaneumfpack*. Neste pacote é que o arquivo Java é criado com o nome que referencie a funcionalidade da implementação (*UmfpackJni*, no exemplo). Neste arquivo é que residem as chamadas dos métodos nativos em Java

contendo os parâmetros que forem necessários para a execução do método. Além disto, o método deve possuir as palavras-chave *public static final native*, conforme mostrado na Listagem B.1. É através da palavra-chave *native* que o compilador identifica que trata-se de um método que deverá chamar uma biblioteca dinâmica nativa.

Listagem B.1: Código Java para a chamada do método nativo contido na biblioteca dinâmica

```
1  package insaneumfpack;
2
3  public class UmfpackJni {
4    static {
5      System.out.println(System.getProperty("os.name") + "-" + System.getProperty("os.arch
           "));
6      if (System.getProperty("os.name").trim().toLowerCase().contains("windows")) {
7        if (System.getProperty("os.arch").trim().toLowerCase().contains("64")) {
8          System.out.println("*** Loading Windows x64 sparse solver classes ***");
9          System.loadLibrary("libinsaneumfpack_win64");
10       } else {
11         System.out.println("*** NOT Loading Windows x32 sparse solver classes ***");
12       }
13     } else {
14       System.out.println("*** Loading default x64 sparse solver class ***");
15       System.loadLibrary("insaneumfpack");
16     }
17   }
18
19 public static final native double[] solver(int[] ap, int[] ai, double[] ax, double[] b,
       double[] x);
20 public static final native double[] solver2(int[] ap, int[] ai, double[] ax, double[] b,
        double[] x, double[] info, double[] control);
21 }
```

Além dos métodos de chamada nativa, optou-se por adicionar um bloco estático que faz a escolha da biblioteca dinâmica conforme o sistema operacional. Assim, quando um método da classe for chamado, a biblioteca adequada já é automaticamente carregada pela própria classe.

B.2 Segundo Passo

Na classe Java o compilador já possui a informação dos métodos que são nativos, contudo é preciso uma forma de informar qual das funções nativas devem ser associadas a quais métodos Java. Esta ligação é feita através da utilização de padrão de nome das funções. Esta especificação segue a seguinte ordem, utilizando como separador o caractere *underscore*

1. Inicia-se o nome do método pelo prefixo Java
2. Os nomes dos pacotes
3. O nome da classe que possui o método Java
4. Por fim, o nome do método Java

Assim, no caso do exemplo mostrado na implementação do UMFPACK, os métodos nativos devem ser nomeados em C por $Java\_insaneumfpack\_UmfpackJni\_solver$ e $Java\_insaneumfpack\_UmfpackJni\_solver2$. Contudo, uma forma mais fácil, e menos propensa a erros, de obter a chamada da função nativa é utilizando o comando do compilador *javac*, mostrado abaixo, para gerar o arquivo *header*.

$$javac \;\; arquivo.java \;\; \text{-}h \;\; < diretório >$$

No caso do exemplo UMFPACK:

$$javac \;\; UmfpackJni.java \;\; \text{-}h \;\; ../../jni$$

Que resulta no seguinte arquivo:

Listagem B.2: Arquivo *header* gerado automaticamente pelo compilador Java

```
1 /* DO NOT EDIT THIS FILE - it is machine generated */
2 #include <jni.h>
3 /* Header for class insaneumfpack_UmfpackJni */
4
```

```
5  #ifndef _Included_insaneumfpack_UmfpackJni
6  #define _Included_insaneumfpack_UmfpackJni
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:     insaneumfpack_UmfpackJni
12  * Method:    solver
13  * Signature: ([I[I[D[D[D)[D
14  */
15 JNIEXPORT jdoubleArray JNICALL Java_insaneumfpack_UmfpackJni_solver
16   (JNIEnv *, jclass, jintArray, jintArray, jdoubleArray, jdoubleArray, jdoubleArray);
17
18 /*
19  * Class:     insaneumfpack_UmfpackJni
20  * Method:    solver2
21  * Signature: ([I[I[D[D[D[D[D[D
22  */
23 JNIEXPORT jdoubleArray JNICALL Java_insaneumfpack_UmfpackJni_solver2
24   (JNIEnv *, jclass, jintArray, jintArray, jdoubleArray, jdoubleArray, jdoubleArray,
       jdoubleArray, jdoubleArray);
25
26 #ifdef __cplusplus
27 }
28 #endif
29 #endif
```

Como é possível perceber pela Listagem B.2, além do nome padrão do método nativo em C, os parâmetros de chamada e de retorno do método também possuem um padrão a se seguir. Isto ocorre porque é preciso fazer uma correlação entre variáveis primitivas em Java e variáveis primitivas em C. Na tabela a seguir são mostrados alguns nomes de tipos primitivos Java chamados em C. É claro que ao gerar o arquivo *header* pelo compilador Java estas variáveis em C já são criadas na assinatura do método, como mostrado anteriormente.

Primitivos Java	boolean	byte	char	short	int	long	float	double
Chamada em C	jboolean	jbyte	jchar	jshort	jint	jlong	jfloat	jdouble

Além disso, percebe-se a adição de dois outros argumentos aos parâmetros. Conforme explica Liang (1999), o parâmetro *JNIEnv* é um *pointer* de uma localização que contém outro *pointer* para uma tabela de funções. Estas funções é que permitem manipular os dados Java em C e C++, e que serão brevemente explicadas no passo a seguir. Já o segundo argumento, *jobject*, serve como referência ao objeto que fez a chamada nativa. Claro que se o método for estático, como é o caso da implementação, a variável trata-se de uma referência à classe, e não ao objeto.

A partir do arquivo header, é possível copiar o nome padrão que a rotina em C deve seguir e então começar a desenvolver a implementação. Contudo, como pode ser visto na Listagem B.2, os argumentos da função C não estão completo, estando nomeados apenas os tipos. É preciso, ainda, completar o nome de cada parâmetro, sendo apropriado utilizar os mesmos nomes da classe Java.

B.3 Terceiro Passo

A criação do código C depende, primeiramente, da compilação das bibliotecas nativas que serão utilizadas na implementação. Para tanto, no padrão aqui idealizado, o código fonte destas são armazenadas dentro de uma pasta com o sufixo *_src*, como mostrado na Figura B.1, e as regras de compilação localizam-se no arquivo *makefile* do projeto. Os arquivos gerados pela compilação do código fonte (geralmente pastas como *include* e *lib*, conforme aa Figura B.1) são armazenados nas pastas *compilation_linux_x64* e *compilation_windows_x64*. Portanto, quando o *linker* e o *compiler* forem acionados, as dependências já estão organizadas nestes dois diretórios.

Com a biblioteca devidamente compilada, pode-se começar a implementação C. A utilização dos parâmetros passados na chamada da função, que são objetos Java, é bastante simples no caso de tipos primitivos, podendo-se trabalhar diretamente com estas variáveis. Contudo, no caso de de *Arrays* ou *Strings*, a transmissão de Java para

C, ou C++, precisa ser realizada através de funções especiais. Estas são encontradas através do ponteiro *JNIenv* e declaradas no *header jni.h*, localizado na pasta *include* do JDK (*Java Development Kit*), e são *GetStringUTFChars*, *GetIntArrayElements*, *GetDoubleArrayElements*, *GetObjectArrayElements*, etc.

Após o uso e, se necessário, a alteração dos *Arrays* e *Strings*, ao final da função, quando estes elementos já não são mais necessários, é preciso liberá-los da memória, para o uso em Java, com os métodos *ReleaseStringUTFChars*, *ReleaseIntArrayElements*, *ReleaseDoubleArrayElements*, *ReleaseObjectArrayElements*, etc. Caso este procedimento não seja realizado, ocorre um vazamento de memória (*memory leak*).

Listagem B.3: Código C contido na biblioteca dinâmica

```
1  #include <stdio.h>
2  #include <jni.h>
3  #include "umfpack.h"
4
5  JNIEXPORT jdoubleArray JNICALL Java_insaneumfpack_UmfpackJni_solver
6  (JNIEnv *env2, jobject jobj, jintArray Apj, jintArray Aij, jdoubleArray Axj,
       jdoubleArray bj, jdoubleArray xj){
7      jsize   n= (*env2) -> GetArrayLength(env2, Apj); n=n-1;
8      jint    *Ap= (*env2) -> GetIntArrayElements(env2,Apj,0);
9      jint    *Ai= (*env2) -> GetIntArrayElements(env2,Aij,0);
10     jdouble *Ax = (*env2) -> GetDoubleArrayElements(env2,Axj,0);
11     jdouble *b =(*env2) -> GetDoubleArrayElements(env2,bj,0);
12     jdouble *x =(*env2) -> GetDoubleArrayElements(env2,xj,0);
13
14     int status=0;
15     int failed = 0;
16     void *Symbolic, *Numeric ;
17     double Control[UMFPACK_CONTROL], Info[UMFPACK_INFO];
18
19     umfpack_di_defaults (Control) ;
20
21     Control[UMFPACK_PRL] = 2;
22
23     status = umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic, Control, Info) ;
24     failed = failed && (status != 0);
25
26     if (status != 0) {
```

```
27        printf("\n*** ERROR: after umfpack_di_symbolic! *** \n");
28      }
29
30      if (!failed) {
31        status = umfpack_di_numeric(Ap, Ai, Ax, Symbolic, &Numeric, Control, Info) ;
32        failed = failed && (status != 0);
33        if (status != 0) {
34          printf("\n*** ERROR: after umfpack_di_numeric! *** \n");
35        }
36      }
37
38      umfpack_di_free_symbolic (&Symbolic) ;
39
40      if (!failed) {
41        status = umfpack_di_solve(UMFPACK_A, Ap, Ai, Ax, x, b, Numeric, Control, Info) ;
42        failed = failed && (status != 0);
43        if (status != 0) {
44          printf("\n*** ERROR: after umfpack_di_solve! *** \n");
45        }
46      }
47
48      if (failed) {
49        umfpack_di_report_info(Control, Info);
50        umfpack_di_report_control(Control);
51        umfpack_di_report_status(Control, status);
52      }
53
54      umfpack_di_free_numeric (&Numeric) ;
55
56      (*env2) -> ReleaseIntArrayElements(env2, Apj, Ap, 0);
57      (*env2) -> ReleaseIntArrayElements(env2, Aij, Ai, 0);
58      (*env2) -> ReleaseDoubleArrayElements(env2, Axj, Ax, 0);
59      (*env2) -> ReleaseDoubleArrayElements(env2, bj, b, 0);
60      (*env2) -> ReleaseDoubleArrayElements(env2, xj, x, 0);
61
62      return xj;
63  }
```

B.4 Quarto Passo

Com os métodos criados na parcela C da implementação, o próximo passo é compilá-los em uma biblioteca dinâmica (extensão **.dll** para Windows e **.so** para Linux). Esta etapa irá depender da biblioteca nativa externa e as opções que os criadores indicam para sua compilação. Entretanto, o importante é que, para a *linkagem* da biblioteca externa, deve-se utilizar uma compilação estática (extensão **.lib** para Windows e **.a** para Linux). Assim, a biblioteca dinâmica gerada para o JNI realiza uma cópia da biblioteca estática externa, tornando o carregamento desta automático quando o JNI for acionado. As regras para a compilação do JNI devem ser mantidas no arquivo *makefile* na própria pasta jni.

B.5 Quinto Passo

Por fim, compila-se o arquivo Java que faz a chamada para a biblioteca nativa a partir do comando *javac*. No caso do projeto JNI do sistema INSANE, os arquivos são gerenciados através da ferramenta Maven, que automaticamente compila os arquivos diretamente pelo comando *mvn package*. Assim, são empacotados tanto os arquivos Java *.class* quanto as bibliotecas dinâmicas nativas criadas para a chamada JNI. Para isto, é criado o arquivo *pom.xml* para cada módulo, e atualizado o *pom.xml* do projeto principal.

Para executar o código criado, é preciso informar à JVM a localização da biblioteca dinâmica. Para isto existem duas opções. Uma possibilidade é de usar o argumento *-Djava.library.path= <diretório>* para iniciar a JVM. Outra alternativa é incluir em alguma parte do programa o comando *System.setProperty("java.library.path", "diretório")*, não sendo a opção ideal já que o programa torna-se um código rígido.

Referências Bibliográficas

Adeli, H., Kamat, M., Kulkarni, G. e Vanluchene, R. 1993, 'High-performance computing in structural mechanics and engineering', *Journal of Aerospace Engineering* **6**(3), 249–267.

Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S. e Tomov, S. 2010, Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, *in* W. mei W. Hwu, ed., 'GPU Computing Gems', Vol. 2, Morgan Kaufmann. `https://hal.inria.fr/inria-00547847/file/gpucomputinggems_plagma.pdf`.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., Van de Geijn, R. e Wu, Y.-J. 1997, Plapack: Parallel linear algebra package, *in* 'Proc. of the SIAM Parallel Processing Conference', p. 8.

Alves, P. D. 2012, Estratégia global-local aplicada ao método dos elementos finitos generalizados, Dissertação de Mestrado, UFMG. Disponível em: `http://pos.dees.ufmg.br/diss_defesas_detalhes.php?aluno=692`. Acessado em: 20/05/2019.

Amdahl, G. M. 1967, Validity of the single processor approach to achieving large scale computing capabilities, *in* 'Proceedings of the April 18-20, 1967, spring joint computer conference', ACM, pp. 483–485.

Amestoy, P. R., Duff, I. S. e L'excellent, J.-Y. 2000, 'Multifrontal parallel distributed symmetric and unsymmetric solvers', *Computer methods in applied mechanics and engineering* **184**(2-4), 501–520.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H. e Zhang, H. 2019*a*, PETSc users manual, Technical Report ANL-95/11 - Revision 3.11, Argonne National Laboratory.
URL: *www.mcs.anl.gov/petsc*

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H. e Zhang, H. 2019*b*, 'PETSc Web page', `www.mcs.anl.gov/petsc`.
URL: *www.mcs.anl.gov/petsc*

Balay, S., Gropp, W. D., McInnes, L. C. e Smith, B. F. 1997, Efficient management of parallelism in object oriented numerical software libraries, *in* E. Arge, A. M. Bruaset e H. P. Langtangen, eds, 'Modern Software Tools in Scientific Computing', Birkhäuser Press, pp. 163–202.

Barth, T. J., Chan, T. F. e Tang, W.-P. 1998, 'A parallel non-overlapping domain-decomposition algorithm for compressible fluid flow problems on triangulated domains', *Contemporary Mathematics* **218**, 23–41.

Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G. et al. 2002, 'An updated set of basic linear algebra subprograms (blas)', *ACM Transactions on Mathematical Software* **28**(2), 135–151.

Blatt, M. e Bastian, P. 2007, The iterative solver template library, *in* B. Kagström, E. Elmroth, J. Dongarra e J. Wasniewski, eds, 'Applied Parallel Computing – State of the Art in Scientific Computing', Springer, Berlin/Heidelberg, pp. 666–675.

Bording, R. 1981, Parallel processing and finite element methods, *in* 'Conference Proceedings Southeastcon'81.', IEEE, pp. 189–193.

Carpenter, B., Getov, V., Judd, G., Skjellum, A. e Fox, G. 2000, 'Mpj: Mpi-like message passing for java', *Concurrency: Practice and Experience* **12**(11), 1019–1038.

Chan, A., Gropp, W. e Lusk, E. 2008, 'An efficient format for nearly constant-time access to arbitrary time intervals in large trace files', *Scientific Programming* **16**, 155–165.

Chevalier, C. e Pellegrini, F. 2008, 'Pt-scotch: A tool for efficient parallel graph ordering', *Parallel computing* **34**(6-8), 318–331.

Chiang, K. e Fulton, R. 1990, 'Concepts and implementation of parallel finite element analysis', *Computers & Structures* **36**(6), 1039–1046.

Chien, L. e Sun, C. 1989, 'Parallel processing techniques for finite element analysis of nonlinear large truss structures', *Computers & structures* **31**(6), 1023–1029.

Choi, J., Dongarra, J. J., Pozo, R. e Walker, D. W. 1992, Scalapack: A scalable linear algebra library for distributed memory concurrent computers, *in* 'Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation', IEEE Computer Society Press, pp. 120–127.

Dangorra, J. J., Moler, C. B., Bunch, J. R. e Stewart, G. W. 1979, *Linkpack User's Guide*, Siam.

Davis, T. A. 2004*a*, 'Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method', *ACM Transactions on Mathematical Software (TOMS)* **30**(2), 196–199. Disponível em: `https://dl.acm.org/citation.cfm?doid=992200.992206`. Acessado em 31/05/2019.

Davis, T. A. 2004*b*, 'A column pre-ordering strategy for the unsymmetric-pattern multifrontal method', *ACM Transactions on Mathematical Software (TOMS)* **30**(2), 165–195. Disponível em: `https://dl.acm.org/citation.cfm?doid=992200.992205`. Acessado em: 31/05/2019.

Davis, T. A. 2006, *Direct Methods for Sparse Linear Systems*, Siam, Philadelphia, PA.

Devine, K. D., Boman, E. G., Riesen, L. A., Catalyurek, U. V. e Chevalier, C. 2009, Getting started with zoltan: A short tutorial, *in* U. Naumann, O. Schenk, H. D. Simon e S. Toledo, eds, 'Combinatorial Scientific Computing', number 09061 *in* 'Dagstuhl Seminar Proceedings', Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany.
URL: *http://drops.dagstuhl.de/opus/volltexte/2009/2088*

Dongarra, J., Graybill, R., Harrod, W., Lucas, R., Luks, E., Luszcek, P., Mcmahon, J., Snavely, A., Vetter, J., Yelick, K., Alam, S., Campbell, R., Carrington, L. e Chen, T.-Y. 2008, Darpa's hpcs program: History, models, tools, languages, *in* M. V. Zelkowitz, ed., 'Advances in Computers: High Performance Computing', Vol. 6, Elsevier, London, chapter 1, pp. 491–540.

Dongarra, J. e Luszczek, P. 2011, Lapack, *in* D. Padua, ed., 'Encyclopedia of Parallel Computing', Springer, pp. 1005–1006.

dos Santos, K. F. 2018, Método dos elementos finitos generalizados aplicado a problemas de fratura elástica em 3d, Dissertação de Mestrado, UFMG. Disponível em: `http://pos.dees.ufmg.br/diss_defesas_detalhes.php?aluno=1289`. Acessado em 20/05/2019.

Duncan, R. 1992, Parallel computer architectures, *in* 'Advances in computers', Vol. 34, Elsevier, pp. 113–157.

Eager, D. L., Zahorjan, J. e Lazowska, E. D. 1989, 'Speedup versus efficiency in parallel systems', *IEEE Transactions on Computers* **38**(3), 408–423.

Expósito, R. R., Taboada, G. L., Touriño, J. e Doallo, R. 2012, 'Design of scalable java message-passing communications over infiniband', *The Journal of Supercomputing* **61**(1), 141–165.

Falgout, R. e Yang, U. 2002, hypre: a library of high performance preconditioners, *in* P. Sloot, C. Tan., J. Dongarra e A. Hoekstra, eds, 'Computational Science', Vol. 2331 of *Lecture Notes in Computer Science*, Springer-Verlags, pp. 163–202.

Farhat, C. e Wilson, E. 1987, 'Solution of finite element systems on concurrent processing computers', *International Journal for Numerical Methods in Engineering* **24**, 1771–1792.

Flynn, M. 2011, Flynn's taxonomy, *in* D. Padua, ed., 'Encyclopedia of Parallel Computing', Springer, pp. 689–697.

Fonseca, M. T. 2006, Aplição orientada a objetos para análise fisicamente não-linear com modelos reticulados de seções transversais compostas, Dissertação de Mestrado, UFMG. Disponível em: `http://pos.dees.ufmg.br/diss_defesas_detalhes.php?aluno=415`. Acessado em 31/05/2019.

Foster, I. 2003, 'Designing and building parallel programs', Disponível em: `http://www-unix.mcs.anl.gov/dbpp/text/book.html`. Acessado em: 14/05/2019.

Gittens, A., Devarakonda, A., Racah, E., Ringenburg, M., Gerhardt, L., Kottalam, J., Liu, J., Maschhoff, K., Canon, S., Chhugani, J., Sharma, P., Yang, J., Demmel, J., Harrell, J., Krishnamurthy, V., Mahoney, M. W. e Prabhat 2016, Matrix factorizations at scale: a comparison of scientific data analytics in spark and C+MPI using three case studies, *in* '2016 IEEE International Conference on Big Data', Washingtion, DC, USA, pp. 204–213. DOI: 10.1109/BigData.2016.7840606.

Goto, K. e Van de Geijn, R. A. 2008, 'Anatomy of high-performance matrix multiplication', *ACM Transactions on Mathematical Software* **34**(3), 12:1–12:25.

Gustafson, J. L. 1988, 'Reevaluating amdahl's law', *Communications of the ACM* **31**(5), 532–533.

Hennessy, J. L. e Patterson, D. A. 2017, *Computer Architecture: A Quantitative Approach*, 6th edn, Morgan Kaufmann, Cambridge, MA.

Hénon, P., Ramet, P. e Roman, J. 2002, 'PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems', *Parallel Computing* **28**(2), 301–321.

Irons, B. M. 1970, 'A frontal solution program for finite element analysis', *International Journal for Numerical Methods in Engineering* .

Jha, S., Qiu, J., Luckow, A., Mantha, P. e Fox, G. C. 2014, A tale of two data-intensive paradigms: applications, abstractions and architectures, *in* 'IEEE International Congress on Big Data', pp. 645–652.

Jordan, H. F. 1978, A special purpose architecture for finite element analysis, Technical report, NASA Langley Research Center, Hampton, VA. `https://ntrs.nasa.gov/search.jsp?R=19790010474`, Acessado em: 20/05/2019.

Karypis, G. e Kumar, V. 1998, 'A parallel algorithm for multilevel graph partitioning and sparse matrix ordering.', *Journal of Parallel and Distributed Computing* .

Karypis, G. e Kumar, V. 1999, 'A fast and highly quality multilevel scheme for partitioning irregular graphs', *SIAM Journal on Scientific Computing* .

King, R. B. e Sonnad, V. 1987, 'Implementation of an element-by-element solution algorithm for the finite element method on a coarse-grained parallel computer', *Computer Methods in Applied Mechanics and Engineering* .

Krishnaprasad, S. 2001, 'Uses and abuses of amdahl's law', *Journal of Computing Sciences in colleges* **17**(2), 288–293.

Krysl, P. e Bittnar, Z. 2001, 'Parallel explicit finite element solid dynamics with domain decomposition and message passing: dual partitioning scalability', *Computers & Structures* .

Kshemkalyani, A. D. e Singhai, M. 2008, *Distributed Computing: Principles, Algorithms, and Systems*, New York.

Law, K. H. 1986, 'A parallel finite element solution method', *Computers & Structures* **23**(6), 845–858.

Li, X., Demmel, J., Bailey, D., Hida, Y., Iskandar, J., Kapur, A., Martin, M., Thompson, B., Tung, T. e Yoo, D. 2008, 'Xblas homepage', online. Disponível em: `http://xianyi.github.com/OpenBLAS/`. Acessado em: 27/05/2019.

Li, X. S. 2005, 'An overview of SuperLU: Algorithms, implementation, and user interface', *ACM Transactions on Mathematical Software* **31**(3), 302–325.

Li, Z., Saad, Y. e Sosonkina, M. 2003, 'parms: a parallel version of the algebraic recursive multilevel solver', *Numerical Linear Algebra With Applications* .

Liang, S. 1999, *The Java native interface: programmer's guide and specification*, Addison-Wesley Professional.

M. Nowicki, P. B. 2012, Parallel computations in java with pcj library, *in* W. W. Smari e V. Zeljkovic, eds, 'Proceedings of International Conference on High Performance Computing & Simulation (HPCS)', pp. 381–387.

Malekan, M. 2017, Crack propagation modeling in plane structures using two-scale generalized/extended finite element method, Tese de Doutorado, UFMG.

Markoff, J. 2005, 'Writing the fastest code, by hand, for fun: A human computer keeps speeding up chips', online. Disponível em: `https://www.nytimes.com/2005/11/28/technology/writing-the-fastest-code-by-hand-for-fun-a-human-computer-keeps.html`. Acessado em: 27/05/2019.

Martínez-Frutos, J., Martínez-Castejón, P. J. e Herrero-Pérez, D. 2015, 'Fine-grained gpu implementation of assembly-free iterative solver for finite element problems', *Computers and Structures* .

Message Passing Interface Forum 2015, 'Mpi: A message-passing interface standard – version 3.1'. Disponível em: `www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. Acessado em: 17/07/2019.

Mytkowicz, T., Diwan, A., Hauswirth, M. e Sweeney, P. F. 2010, 'Evaluating the accuracy of java profilers', *Programming Language Design and Implementation* pp. 561–571.

Noor, A. K. 1997, 'New computing systems and future high-performance computing environment and their impact on structural analysis on design', *Computers & Structures* **64**(1–4), 1–30.

Noor, A. K. e Hartley, S. J. 1978, 'Evaluation of element stiffness matrices on cdc star-100 computer', *Computers & Structures* **9**, 151–161.

Noor, A. K. e Lambiotte, J. J. 1979, 'Finite element dynamic analysis on cdc star-100 computer', *Computers & Structures* **10**, 7–19.

Oaks, S. 2014, *Java Performance – The Definitive Guide*, 1 edn, O'Reilly.

Pellegrini, F. e Roman, J. 1996, 'Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs', *HPCN-Europe, Springer LNCS 1067* .

Przemieniecki, J. S. 1963, 'Matrix structural analysis of substructures', *Air Force Institute of Technology* **1**(1), 138–147.

Reyes-Ortiz, J. L., Oneto, L. e Anguita, D. 2015, Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf, *in* 'INNS Conference on Big Data', Vol. 53, Elsevier, pp. 121–130. DOI: 10.1016/j.procs.2015.07.286.

Sanders, P. e Schulz, C. 2013, Think Locally, Act Globally: Highly Balanced Graph Partitioning, *in* 'Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)', Vol. 7933 of *LNCS*, Springer, pp. 164–175.

Schloegel, L., Karypis, G. e Kumar, V. 2003, Graph partitioning for high-performance scientific simulations, *in* J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon e A. White, eds, 'Sorcebook of parallel computing', Morgan Kaufmann Publishers, San Francisco, chapter 18, pp. 491–540.

Shi, Y. 1996, 'Reevaluating amdahl's law and gustafson's law', *Computer Sciences Department, Temple University (MS: 38-24)* .

Smith, B. T., Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C. e Moler, C. B. 1976, Matrix eigensystem routines – eispack guide, *in* 'Lecture Notes in Computer Science', Vol. 6, Springer, Berlin.

Storaasli, O. O., Peebles, S. W., Crockett, T. W., Knott, J. D. e Adams, L. 1982, The finite element machine: an experiment in parallel processing, Technical report, NASA Langley Research Center, Hampton, VA. Disponível em: `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19820024127.pdf`, Acessado em: 20/05/2019.

Tanenbaum, A. S. e Austin, T. 2013, *Structured Computer Organization*, 6th edn, Pearson Education, New Jersey.

Toselli, A. e Widlund, O. 2005, *Domain Decomposition Methods – Algorithms and Theory*, 1st edn, Springer, Berlin.

Ungerer, T., Robič, B. e Šilc, J. 2003, 'A survey of processors with explicit multithreading', *ACM Computing Surveys* **34**(1), 29–63. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.9105&rep=rep1&type=pdf` Acessado em: 18/05/2019.

Van Zee, F. G., Chan, E., Van de Geijn, R., Quintana-Orti, E. S. e Quintana-Orti, G. 2009, 'Introducing: The libflame library for dense matrix computations', *IEEE Computing in Science & Engineering* .

Van-Zee, F. G. e Van de Geijin, R. A. 2015, 'Blis: A framework for rapidly instantiating blas functionality', *ACM Transactions on Mathematical Software* .

Vetter, J. e Chambreau, C. 2014, 'mpip: Lightweight, scalable mpi profiling'. Disponível em: `mpip.sourceforge.net`. Acessado em: 22/07/2019.

Wendykier, P. e Nagy, J. G. 2010, 'Parallel colt: A high-performance java library for scientific computing and image processing', *ACM Transactions on Mathematical Software* .

Whaley, R. C. e Dongarra, J. J. 1998, Automatically tuned linear algebra software, *in* 'Proceedings of the 1998 ACM/IEEE Conference on Supercomputing', SC '98, IEEE Computer Society, Washington, DC, USA, pp. 1–27. Disponível em: `http://dl.acm.org/citation.cfm?id=509058.509096`. Acessado em: 27/05/2019.

Williams, S. W. 2008, Auto-tuning Performance on Multicore Computers, Tese de Doutorado, University of California, Berkeley.

Zhang, X., Wang, Q., e Zhang., Y. 2012, Model-driven level 3 blas performance optimization on loongson 3a processor, *in* '2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)'.