XIII SIMMEC
Simpósio de Mecânica Computacional
29 de Outubro a 1º de Novembro de 2018
UFES - Campus Goiabeiras - Vitória, ES

PARALELIZAÇÃO DA MONTAGEM DE SISTEMAS DE EQUAÇÕES PARA O MÉTODO DOS ELEMENTOS FINITOS

Parallel System Equation Assembly for Finite Element Method

Gabriela Moreira Azevedo (1, P); Samuel Silva Penna (2); Ramon Pereira da Silva (2)

(1) Mestranda no Programa de Pós-Graduação de Engenharia de Estruturas, Universidade Federal de Minas Gerais, Belo Horizonte - MG, Brasil.
(2) Dr. Prof., Universidade Federal de Minas Gerais, Belo Horizonte - MG, Brasil.
Email para Correspondência: gabriela.azevedo12@gmail.com; (P) Apresentador

Resumo: Neste trabalho, apresenta-se a implementação de um procedimento de paralelização da montagem do sistema de equações lineares do método dos elementos finitos. A implementação foi realizada no sistema computacional INSANE (*INteractive Structural ANalysis Environment*), que é desenvolvido em Java segundo o paradigma de Programação Orientada a Objetos. Para tal fim, foram utilizadas as bibliotecas METIS, especializada em particionamento de grafos, e MPJ-Express, uma implementação JAVA para o padrão MPI (*Message Passing Interface*) de computação paralela.

Palavras chaves: Computação Paralela, Método dos Elementos Finitos, MPI.

Abstract: In this work, the implementation of a parallel procedure to assemble the finite element equations is presented. The implementation was performed in the INSANE (INteractive Structural ANalysis Environment) computational system developed in Java according to the Object Oriented Programming paradigm. For this, METIS, a specialized graph partition library, and MPJ-Express, a Java implementation for parallel computing based on MPI (Message Passing Interface) standard design, were used.

Keywords: Parallel Computing; Finite Element Method; MPI.

1 INTRODUÇÃO

Na análise estrutural por elementos finitos, a representação de fenômenos estruturais complexos é cada vez mais recorrente e acaba exigindo uma quantidade cada vez maior de graus de liberdades dos modelos. Na modelagem computacional, esta condição representa um problema significativo de performance, demandando elevado grau de processamento e consumo de memória, em particular nas etapas de montagem e resolução dos sistemas de equações. Desta forma, faz-se necessário o uso de computadores de alta tecnologia e alta performance. Contudo, o uso destas máquinas especializadas é de grande custo e, conforme o avanço da tecnologia, necessitam ser substituídas e atualizadas.

Visando resolver estas questões, tem-se dado atenção para soluções que façam uso de computadores paralelos. Nesta modalidade de computação, um problema grande e complexo é dividido em pequenas porções, que são solucionadas de forma independente em diferentes computadores. Assim, a computação paralela tem como vantagens a demanda menor 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, até então as práticas mais utilizadas, principalmente quando objetiva-se a alta performance. 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 processamento, juntamente com o mínimo de comunicação possível. No caso do cálculo das parcelas de rigidez dos elementos e da montagem da matriz de rigidez global, etapas facilmente paralelizáveis, devem ser utilizadas metodologias que realizem a divisão homogênea dos elementos entre os processos ao mesmo tempo que forme a menor divisão entre os elementos.

O sistema computacional INSANE para análise de estruturas é um *software* criado e mantido pelos alunos e professores do Departamento de Engenharia de Estruturas na Escola de Engenharia da Universidade Federal de Minas Gerais. Sendo usado tanto de forma didática para as aulas quanto para pesquisa e desenvolvimento, constantemente são adicionadas novas metodologias de análises. Quanto mais sofisticadas são essas metodologias, maiores são os problemas de performance. Portanto, é de interesse para o projeto que seja dado enfoque em implementações de códigos paralelos no *software*.

1.1 Arquitetura de computadores paralelos

A organização dos componentes dos computadores afetam seu funcionamento e o modelo de programação que devem ser utilizados de forma a conseguir melhor eficiência da máquina. Segundo a taxonomia de Flynn (Tannenbaum, 2011), existem duas possibilidades para a construção de computadores paralalelos.

Os computadores de uma única instrução e múltiplos dados, conhecido como SIMD (*Single Instruction Multiple Data*), é obtida por processadores que possuem várias unidades de lógica e aritmética e uma única unidade de controle. Assim, esse modelo permite que um conjunto de dados seja operado ao mesmo tempo, realizando a mesma operação lógica/matemática instruída na unidade de controle. Essa configuração é bem

aproveitada por execuções de laços de repetição, como as que ocorrem em matrizes e vetores. Contudo, sua vantagem é perdida quando temos dados que demandam diferentes procedimentos. Os computadores vetoriais, matriciais e as placas gráficas (GPUs) fazem uso deste tipo de arquitetura de processadores, utilizando a técnica denominada paralelismo de dados (*data parallelism*).

Já o modelo de múltiplas instruções e múltiplos dados, chamado MIMD (*Multiple Instruction Multiple Data*), é a forma de paralelismo obtida a partir da composição de várias unidades de processamento central, que operam individualmente diferentes programas ou instruções sobre diferentes conjuntos de dados. Para sua maior eficiência, é essencial a independência entre os dados para evitar a necessidade de operações custosas de comunicação e sincronização. Este padrão de computador paralelo é presente em multiprocessadores, *multicores* e multicomputadores. A utilização deste tipo de paralelismo é denominada como paralelismo de tarefas (*task parallelism*).

Apesar da classificação, a procura por rapidez e eficiência fez com que surgissem computadores especializados tanto no paralelismo de tarefas quanto de dados. Isso é possível pois são incorporados componentes especializados em paralelismo de tarefas e e paralelismo de dados. Com isso, as unidades de processamento mais atuais já conseguem trabalhar com eficiência independente do tipo de paralelismo que o problema apresente. Contudo, no caso da linguagem de programação JAVA, não existe uma forma explícita de aproveitar o paralelismo de dados do processador, como é o caso da linguagem C.

1.2 Arquitetura de memória

Conforme Tanenbaum (2011), dentro da arquitetura MIMD, comumente encontrada nos computadores atuais, existem dois princípios de organização de memória: a memória compartilhada e a memória distribuída.

A primeira alternativa ocorre em multiprocessadores e multicores, pois, nestes casos, os componentes de processamento estão conectados à mesma memória. Desta forma, é possível que valores armazenados na memória sejam acessados por todos os processadores. Essa configuração traz como vantagem a comunicação direta entre processos através memória comum, esta sendo muito veloz. No entanto, como desvantagens, existem claras limitações no tamanho de memória disponível para cada processador além do número de processadores que conectam-se e acessam a mesma memória ao mesmo tempo.

A segunda opção, abrange os multicomputadores, também conhecidos como clusters ou processadores massivamente paralelos. Nestes casos, os computadores são verdadeiramente paralelos, pois são várias máquinas, com processador e memória próprios, conectadas por uma rede. Assim, cada computador possui seu próprio sistema operacional controlando suas atividades. Consequentemente, dados contidos em outros computadores não podem ser acessados diretamente, mas somente através do envio pela rede e cópia para sua própria memória. Os multicomputadores têm como vantagem a facilidade com que o sistema pode ser ampliado, em comparação aos multiprocessadores.

No entanto, a sua velocidade de comunicação é algumas muitas vezes inferior a da memória compartilhada.

Apesar dessa divisão, é bastante comum que clusters sejam organizados de forma híbrida, sendo composto por vários computadores multiprocessadores com núcleos de processamento multicore. Já os computadores pessoais são normalmente processadores multicore, e, portanto, seguem a linha de memória compartilhada.

1.3 Padrões de programação paralela

Existindo dois tipos de arquitetura de memória com abordagens bastante diferentes para a comunicação entre processos, os programas devem levar em consideração estes aspectos. Para tanto, existem dois padrões de programação paralela, o *multithreading* e a passagem de mensagens, conhecida por MPI.

As *threads* são uma abstração do funcionamento dos computadores que encarregam-se de guardar e executar tarefas conforme são gerenciadas pelo sistema operacional. Portanto, o *multithreading* ocorre quando várias *threads* são criadas para a execução de múltiplos processos. No entanto, o *multithreading*, por si só, não é uma forma de paralelismo, já que a execução simultânea das tarefas somente ocorre na existência de mais de um processador. Nestes casos, é o sistema operacional que se encarrega de distribuir as *threads* em diferentes unidades de processamento. E justamente devido a esse envolvimento do sistema operacional que o *multithreading* ocorre apenas em computadores de memória compartilhada (*multiprocessadores* e *multicore*). Consequentemente, o *multithreading* possui as mesmas facilidades e adversidades dos computadores de memória compartilhada.

O segundo padrão, MPI, abreviação de *Message Passing Interface*, é um modelo de de comunicação para multicomputadores. Neste paradigma, um único programa é executado ao mesmo tempo em vários processos, cada um sendo independente, exceto quando ocorrem comunicações. Dessa maneira, as diferentes atividades que cada processo deve realizar são atribuídas através do seu número de identificação, conhecido como *rank*, que é definido através do MPI. Quando for preciso a troca de dados e informações, a comunicação pode ser realizada ponto-a-ponto, de um *rank* para outro, ou coletiva, para todos os *ranks*. Apesar do MPI ser originalmente projetado para funcionar em arquiteturas distribuídas, seu funcionamento também ocorre em memória compartilhada sem o acesso direto à memória, somente com passagem de mensagens. No entanto, a partir de sua terceira versão já foram incluídos métodos exclusivos para atividades em memória compartilhada.

Como o MPI abrange a possibilidade de execução tanto em computadores de memória distribuída quanto de memória compartilhada, deu-se preferência por este padrão de comunicação. Entretanto, também existe a possibilidade de construir algoritmos híbridos que utilizem os dois padrões. Assim, utilizam-se as vantagens do MPI, nos componentes sem memória compartilhada, e as vantagens do multithreading, nas unidades com memória compartilhada.

2 PARALELIZAÇÃO DA MONTAGEM DO SISTEMA DE EQUAÇÕES PARA ANÁLISE ESTRUTURAL VIA MEF

A aplicação da computação paralela juntamente com o Método dos Elementos Finitos (MEF) já é estudada a algumas décadas e, conforme Adeli et al. (1993), pode ser convenientemente empregada em todas as principais etapas de uma análise, que constituem em:

i) geração das matrizes de rigidez e vetores de forças dos elementos;

ii) montagem do sistema global de equações

iii) solução do sistema global de equações;

iv) obtenção das tensões e deformações.

Observa-se que nas etapas i e iv é possível criar algoritmos perfeitamente paralelos, que não dependem de comunicação, uma vez que são necessários dados contidos apenas em cada elemento. No entanto, as etapas ii e iii são procedimentos que demandam comunicação e sincronização das tarefas, aspectos considerados perdas no processo, e que, portanto, devem ser estudadas e minimizadas.

Conforme Adeli et al. (1993) a solução do sistema global de equações pode configurar um problema de performance em análises estruturais lineares. No entanto, quando consideram-se análises não-lineares, as outras etapas também passam a despender uma parcela considerável do tempo de resolução. Isso ocorre em razão do procedimento iterativo de resolução, que exige a geração de novas matrizes de rigidez e verificação das forças internas dos elementos a cada passo do procedimento. Contudo, percebe-se que para a paralelização de qualquer uma destas etapas é preciso que ocorra uma divisão balanceada dos elementos entre os processadores. Isso porque deseja-se que todos os processadores tenham a mesma carga de trabalho e ocupação de memória. Portanto, antes de serem elaborados algoritmos paralelos de geração de matrizes e resolução do sistema de equações, deve-se ter um procedimento eficiente de divisão dos elementos. Todavia, uma simples divisão da lista de elementos entre os processadores não é suficiente, pois questões de comunicação e sincronização tornam-se significativas na etapa de solução do problema. Isso ocorre porque quanto maiores forem as divisões entre elementos vizinhos maior é a comunicação. Deste modo, é preciso aplicar técnicas de partição de domínio que ponderem a comunicação, buscando a divisão igualitária de elementos ao mesmo tempo que reduza a divisão dos elementos vizinhos.

2.1 Particionamento de malhas

O problema de particionamento de malhas é tratado com o uso de grafos e suas metodologias de decomposição. Para isso, é possível a geração de grafos tanto a partir dos nós da malha e suas ligações com os nós adjacentes quanto dos elementos e suas ligações com os elementos adjacentes. Então, através do grafo da malha, dos elementos ou dos nós, utilizam-se procedimentos que procurem a sua decomposição em grafos

menores que minimize o corte das comunicações. Schloegel et al. (2003) identifica as quatro principais metodologias empregadas neste tipo de problema:

a) técnicas geométricas: levam em consideração apenas as coordenadas da malha para realizar a divisão, não avaliando a comunicação do grafo;

b) técnicas combinatórias: realizam divisões avaliando as parcelas do grafo que estão altamente conectadas, assim, produzindo a menor divisão das comunicações;

c) técnicas espectrais: calculam os autovalores da matriz do grafo para gerar sua divisão, sendo as técnicas mais custosas computacionalmente;

d) técnicas multinível: não são propriamente técnicas de partição, pos se tratam de uma única metodologia em três estágios; primeiro, o grafo original é reduzido de tamanho, e então, na segunda etapa, é dividido utilizando alguma das técnicas de particionamento citadas anteriormente, e, por último, o grafo original é reestabelecido com as divisões obtidas anteriormente.

Com o resultado da partição do domínio da malha, distribuem-se as listas de elementos/nós para os processadores, que, de forma paralela, calculam a matriz de rigidez. A partir daí, o que decorre no algoritmo depende do que deseja-se implementar. Se houver um algoritmo paralelo de resolução de sistema linear, as submatrizes devem comunicar-se e transferir as parcelas de rigidez dos graus de liberdade que foram separados. Após, o *solver* paralelo pode ser executado, resolvendo concorrentemente cada porção da matriz global contida em cada processador. Caso não exista o *solver* paralelo, os fragmentos da matriz de rigidez devem ser enviadas para um único processador. Assim, forma-se uma única matriz de rigidez global, em um único computdor, que deve ser solucionada. Percebe-se que neste segundo cenário perde-se a capacidade de modelagem de problemas maiores, já que a análise retorna para apenas uma memória de um computador.

3 BIBLIOTECAS DE PARTICIONAMENTO DE MALHAS

Como o particionamento de grafos é um problema complexo e e ainda muito estudado, existem diferentes programas especializados em realizar esta tarefa. Algumas das bibliotecas livremente disponíveis, que utilizam diferentes abordagens, são: METIS (Karypis e Kumar, 1999), ParMetis (Schloegel et al., 2000), SCOTCH (Pellegrini & Roman, 1996), Zoltan (Devine et al., 2006) e Kahip (Sanders & Schulz, 2013). Para este trabalho, foi escolhida a biblioteca METIS devido sua constante atualização, fácil utilização, acesso ao código e um conversor próprio de malhas em grafos.

A biblioteca METIS permite que sejam realizados corte nos elementos, *element-cut*, ou nos nós, *node-cut*, atribuição de pesos diferentes para elementos/nós além de comprometer-se com a velocidade da resolução. Também, são disponibilizadas duas metodologias de partição de grafos a partir da bisseção recursiva multinível ou *k-way* multinível.

4 MONTAGEM DO SISTEMA DE EQUAÇÕES

Esta seção trata da rotina de criação da matriz de rigidez nos problemas de elementos finitos, mostrando uma ideia geral dos recursos utilizados no *software* INSANE. Nos trabalhos de Fonseca (2008) e Silva (2016) encontram-se mais detalhes a respeito da implementação do programa e seu funcionamento.

No sistema INSANE, a montagem do sistema de equações que representa um problema de elementos finitos estático, do formato da Eq. (1), é desempenhada pela interface *Assembler*.

$$\underline{K} \cdot \underline{d} = \underline{F} \tag{1}$$

onde $\underline{K}$ é a matriz de rigidez global do modelo, $\underline{d}$ é o vetor de deslocamentos e $\underline{F}$ é o vetor de forças.

Para modelos baseados no MEF, a classe *FemAssembler* é responsável pela montagem do sistema de equações escrito na forma

$$\underline{A} \cdot \ddot{\underline{X}} + \underline{B} \cdot \dot{\underline{X}} + \underline{C} \cdot \underline{X} = \underline{D} \tag{2}$$

onde $\underline{X}$ é o vetor com as variáveis de estado, $\underline{D}$ é o vetor com as variáveis duais à variável de estado, $\dot{\underline{X}}$ e $\ddot{\underline{X}}$ são a primeira e segunda variações temporais da variável de estado respectivamente. As matrizes $\underline{A}$, $\underline{B}$ e $\underline{C}$ são matrizes correspondentes a $\ddot{\underline{X}}$, $\dot{\underline{X}}$ e $\underline{X}$ respectivamente.

Considerando um problema de mecânica dos sólidos para análise estrutural estática linear, tem-se que o sistema reduz a

$$\underline{C}\ \underline{X} = \underline{D} \tag{2}$$

onde $\underline{C} = K$, $\underline{X} = \underline{d}$ e $\underline{D} = \underline{F}$.

Portanto, na tarefa de montagem da matriz de rigidez, a interface *Assembler* possui métodos que calculam e retornam submatrizes que compõem a matriz $\underline{C}$, como mostrada na Eq. (3)

$$\begin{bmatrix} \underline{C_{uu}} & \underline{C_{up}} \\ \underline{C_{pu}} & \underline{C_{pp}} \end{bmatrix} \cdot \begin{Bmatrix} \underline{X_u} \\ \underline{X_p} \end{Bmatrix} = \begin{Bmatrix} \underline{R_p} \\ \underline{R_u} \end{Bmatrix} - \begin{Bmatrix} \underline{F_p} \\ \underline{F_u} \end{Bmatrix}. \tag{3}$$

onde submatriz $\underline{C}$ é uma matriz de rigidez obtida conforme os graus de liberdade representados por **u** e **p**. A letra **u** distingue os graus de liberdade desconhecidos, ou seja, sem restrição, enquanto que a letra **p** identifica os graus de liberdade com valores prescritos. O vetor $\underline{X}$ representa os deslocamentos, também separados pelos graus de liberdade desconhecidos e prescritos. Os vetores $\underline{R}$ e $\underline{F}$ são, respectivamente, os vetores de cargas nodais e cargas nodais equivalentes.

Os algoritmos para criar as submatrizes são equivalentes, diferenciando apenas os índices **u** e **p**. O pseudocódigo, apresentado na Figura 1, e a implementação do INSANE, na Figura 2, são rotinas para obter a submatriz $\underline{C_{uu}}$. No entanto, o algoritmo é análogo

para todas as parcelas da matriz de rigidez global que, no caso do sistema INSANE, são diferenciadas pelo sinal, positivo ou negativo, dos graus de liberdade.

```
Rotina para Calcular Cuu
  E := vetor de elementos
  Cuu := matriz de rigidez global dos graus de liberdade livres
  para E[0] até E[máx]
    Ce := matriz de rigidez do elemento E
    Eq := vetor de equações do elemento E
    para i=Eq[0] até Eq[máx]
      para j = Eq[0] até Eq[máx]
        se i > 0 e j > 0
          Cuu[i][j] = Cuu[i][j] + Ce[i][j]
       fim se
     fim para
   fim para
  fim para
 retorne Cuu
```

Figura 1. Pseudocódigo da montagem da submatriz C_{uu}.

Fonte: (Autores, 2018)

```
public IMatrix getCuu() {
  Matrix kr = new IMatrix(this.getSizeOfXu(), this.getSizeOfXu());
  ListIterator<Element> elements = femmodel.getElementsList().listIterator();
  while (elements.hasNext()) {
    Element element = elements.next();
    int[] redEquations = this.getElementEquations(element);
    IMatrix c = element.getC();
    for (int i = 0; i < element.getNumberOfDegreesOfFreedom(); i++) {
      int a = redEquations[i];
      for (int j = 0; j < element.getNumberOfDegreesOfFreedom(); j++) {
        int b = redEquations[j];
        if (a > 0 && b > 0) {
           kr.setElement(a - 1, b - 1, kr.getElement(a - 1, b - 1) + c.getElement(i, j));
         }
       }
     }
   }
  return kr;
}
```

Figura 2. Implementação no INSANE para o cálculo da submatriz C_{uu}.

Fonte: (Autores, 2018)

5 IMPLEMENTAÇÃO

O núcleo numérico do sistema INSANE é formado por quatro principais classes e interfaces, mostradas na Figura 3: *Persistence*, *Model*, *Assembler* e *Solution*.

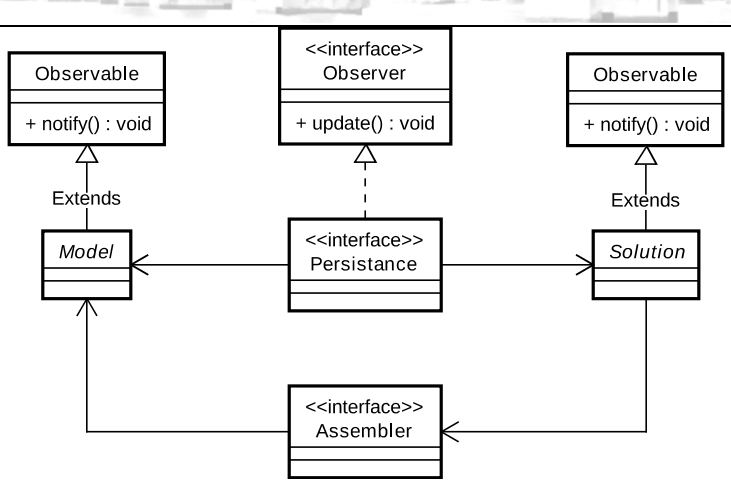

Figura 3. Diagrama de classes do sistema INSANE.

Fonte: (Autores, 2018)

A interface *Persistence* tem como função a leitura dos dados de entrada, geração dos arquivosde saída e intermediar informações entre a classe *Model* e a interface *Assembler*. Esta comunicação entre esses objetos tem como propósito captar alterações que ocorram em qualquer um deles, repassando-as às classes de interesse. Esta conversa é permitida através da implementação da interface *Observer*, pela interface Persistence, e da classeObservable, pelas classes Model e *Solution*.

A classe abstrata *Model* representa o problema estrutural concebido pelo usuário pois guarda todos os dados necessários para a construção do modelo. Suas classes filhas representam diferentes tipos de modelos, como, por exemplo, problemas de elementos finitos, elementos finitos generalizados, elementos de contorno, etc.

A interface *Assembler* interpreta as informações contidas na classe *Model*, criando os sistemas de equações necessários para a resolução do problema proposto. Desta forma, os métodos para a criação destes componentes são definidos nesta interface.

Por último, a classe abstrata Solution tem como função solucionar o sistema de equações do problema, empregando-se, para tanto, rotinas apropriadas à resolução. Estas rotinas são implementadas nas classes filhas da herança.

A interface *Assembler* é implementada por diferentes classes no INSANE, como, por exemplo, a classe *FemAssembler* para o problema de elementos finitos. Para elaborar a matriz de rigidez global de forma paralela, decidiu-se formular uma nova classe, *FemAssemblerPar*, como mostrado na Figura 4. Implementou-se, também, uma nova classe responsável por manipular e guardar todos os dados necessários para os processos paralelos do MPI, nomeada *AssemblerDataManager*.

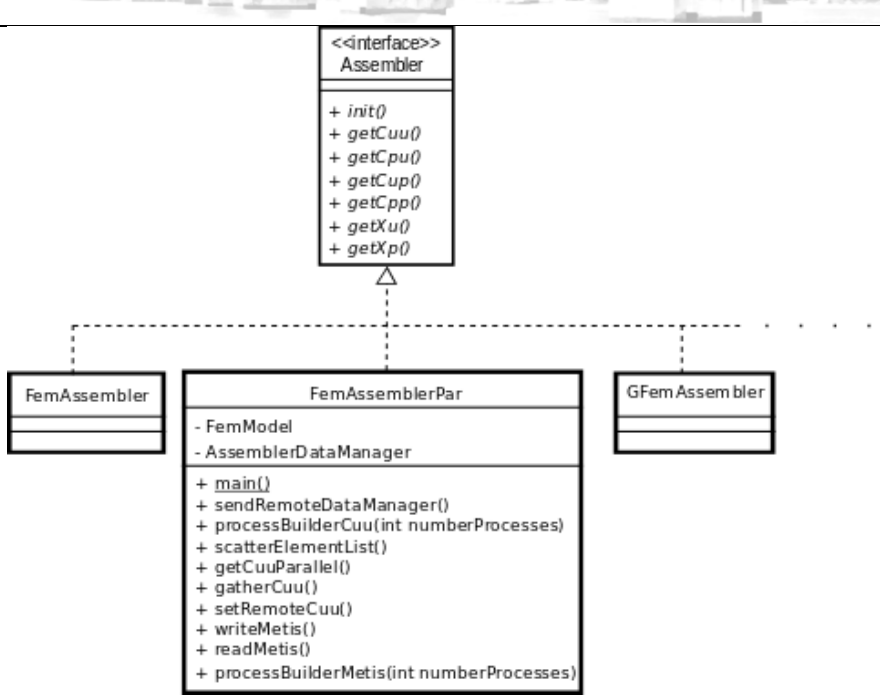

Figura 4. Método *init()* da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

A biblioteca METIS é chamada no sistema através do método *init()* da classe *FemAssemblerPar*. Originalmente, este método tem apenas a função chamar a rotina para iniciar as variáveis do modelo *FemModel*, classe herdeira de *Model*, numerar as equaçoes de cada grau de liberdade, através de *numberEquations()*, e, por último, estabelecer as forças nodais do problema. Com o METIS, foram acrescentados os métodos *writeMetis()*, *processBuilderMetis()* e *readMetis()*, mostrados na Figura 5. Os métodos *writeMetis()* e *readMetis()* tem como função, respectivamente, escrever o arquivo de entrada para a execução do programa e ler o arquivo de resposta. A biblioteca é chamada através do método *processBuilderMetis()*, que executa os comandos necessários através da classe *ProcessBuilder*, presente no próprio pacote de desenvolvimento Java. Após, as listas dos elementos, ou nós, que devem ser distribuídas para cada processador são transcrita para a classe *AssemblerDataManager*. Visando melhor desempenho desta etapa, pretende-se implementar, futuramente, uma chamada direta da biblioteca METIS através de uma implementação JNI (*Java Native Interface*). Desta forma, torna-se possível executar, transmitir e receber dados desta biblioteca, que é escrita na linguagem de programação C.

```
1   public void init(){
3     this.femmodel.init();
4     this.numberEquations();
5     this.initLoading();
6     int numberProcesses = Runtime.getRuntime.availableProcessors();
7     this.writeMetis();
8     this.processBuilderMetis(numberProcesses)
9     this.readMetisElements(numberProcesses);
10  }
```

Figura 5. Método *init()* da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

Para a inclusão dos métodos MPI no INSANE, disponibilizados pela biblioteca MPJ-Express (Shafi et al., 2009), optou-se por criar uma aplicação cuja execução é independente do programa INSANE. Essa decisão ocorre porque, no MPI, existe a obrigatoriedade de criarem-se todos os processos paralelos durante a inicialização do programa. Para a isso, seria necessário reformular todo o código inicial do programa, alterando os fluxos de chamadas para adequarem-se aos vários processos executados ao mesmo tempo. Portanto, decidiu-se separar a execução dos métodos MPI com a chamada de um programa independente, também realizado através da classe *ProcessBuilder*. A comunicação entre estes dois processos é realizada através do pacote RMI (*Remote Method Invocation*), também presente na própria linguagem Java. A Figura 6 ilustra o fluxograma de execuções para o processo de paralelização das tarefas.

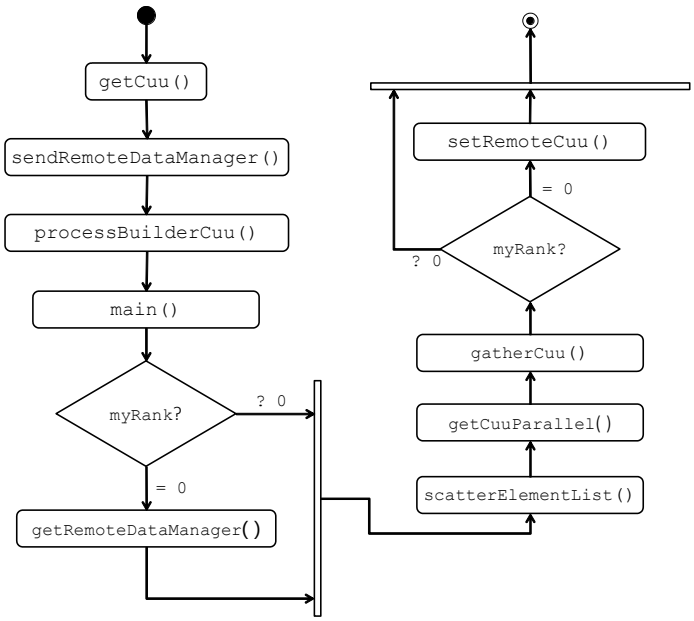

Figura 6. Fluxograma das atividades paralelas.

Fonte: (Autores, 2018)

O fluxograma, mostrado na Figura 6, começa pelo método *getCuu()*, para o exemplo da montagem da submatriz C_{uu}, que é sobrescrito da interface *Assembler* e executado por *FemAssemblerPar*. Com a chamada da função, Figura 7, inicia-se o método *sendRemoteSolution()*, que cria o ambiente remoto de transferência dos objetos, instanciada no INSANE, da classe *AssemblerDataManager*. Assim, quando o *ProcessBuilder* for executado, chamando a nova aplicação com o MPI, o novo fluxo de processamento criado solicita os dados para criar seu próprio objeto *AssemblerDataManager* através de *getRemoteDataManager()*. Para poder iniciar a biblioteca MPI pelo *ProcessBuilder*, a aplicação é chamada utilizando o arquivo starter.jar, da biblioteca MPJ Express, e o parâmetro de número de processos requisitados.

```
1  public IMatrix getCuu() {
2    this.sendRemoteDataManager();
3    int numberProcesses = this.dataManager.getNumberProcesses();
4    this.processBuilderCuu(cores);
5    return this.dataManager.getCuu();
6  }
```

Figura 7. Método *init()* da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

```
1  public static void main(String[] args) throws Exception {
2    MPI.Init(args);
3    int myRank  = MPI.COMM_WORLD.Rank();
4    int size    = MPI.COMM_WORLD.Size();
5    FemAssemblerPar assemb = new FemAssemblerPar();
6    if (myRank == 0) {
7      assemb.getRemoteDataManager();
8    }
9    assemb.scatterElementList(myRank, size);
10   IMatrix partialCuu = assemb.getCuuParallel();
11   IMatrix finalCuu = assemb.gatherCuu(partialCuu);
12   if (myRank == 0) {
13     assemb.setRemoteCuu(finalCuu);
14   }
15   MPI.Finalize();
16 }
```

Figura 8. Método main da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

A aplicação paralela, mostrada no método *main()* da Figura 8, permite que apenas um processo receba a lista completa de elementos. Para isso, a chamada *getRemoteDataManager()* é desempenhada apenas pelo processo identificado pelo número, ou *rank*, 0 na biblioteca MPI. Após, são distribuídas as listas de elementos através do *scatterLocalAssembler()*, Figura 9. Então, cada processo utiliza o método *getCuuParallel()* sobre sua lista elementos e nós, realizando a rotina apresentada anteriormente na Figura 1. Acabadas a rotina, todos os processos retornam suas respectivas matrizes de rigidez para o *rank* 0 com o *gatherLocalAssembler()*, demonstrado na Figura 9. Dessa forma, o processo 0 precisa adicionar todos os resultados

parciais a uma única matriz de rigidez final. Por último, Antes de finalizar a solução paralela, as respostas são devolvidas através do método remoto *setRemoteCuu()*.

```
1  public void scatterElementList(int myRank, int size) {
2    Object[] rcvElementList  = new Object[1];
3    Object[] rcvNodeList = new Object[1];
4    Object[] sendElementList = new Object[size];
5    Object[] sendNodeList = new Object[size];
7    int[] xu = new int[1];
8    if (myRank == 0) {
9      sendElementList = this.dataManager.getElementArray();
10     sendNodeList = this.dataManager.getNodeArray();
12   }
15   MPI.COMM_WORLD.Scatter(sendElementList, 0, 1, MPI.OBJECT, rcvElementList, 0, 1, MPI.OBJECT, 0);
16   MPI.COMM_WORLD.Scatter(sendNodeList, 0, 1, MPI.OBJECT, rcvNodeList, 0, 1, MPI.OBJECT, 0);
17   this.dataManager.setElementList((ArrayList<Element>) rcvElementList[0]);
18   this.dataManager.setNodeList((ArrayList<Node>) rcvNodeList[0]);
19 }
```

Figura 9. Método scatterElementList da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

```
1  public IMatrix gatherCuu(IMatrix partialCuu) throws Exception {
2    Object[] sendPartialCuu = new Object[1];
3    Object[] rcvPartialCuu = new Object[size];
4    send[0] = partialCuu;
5    MPI.COMM_WORLD.Gather(sendPartialCuu, 0, 1, MPI.OBJECT, rcvPartialCuu, 0, 1, MPI.OBJECT, 0);
6    IMatrix finalCuu = null;
     if (myRank == 0) {
7      finalCuu = new IMatrix(assemb.sizeOfXu, assemb.sizeOfXu);
8      for (int i = 0; i < rcv.length; i++) {
9        finalCuu.add((IMatrix) rcvPartialCuu[i]);
10     }
11   this.sendRemoteCuu(finalCuu)
12   }
12   return finalCuu;
13 }
```

Figura 10. Método gatherCuu da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

6 RESULTADOS

Os resultados alcançados com esta implementação inicial não foram satisfatórios, atingindo um tempo muitas vezes maior que o da execução original. Os principais motivos para este fato são:

a) Serialização dos objetos;

b) Tempo de transmissão dos dados entre os dois programas;

c) Tempo de transmissão entre os processos da biblioteca MPI;

d) Representação ineficiente das matrizes de rigidez;

Descobriu-se que a rotina de serialização própria da linguagem Java é bastante ineficiente se comparado a uma serialização personalizada criada para os objetos do sistema INSANE. Isso não ocorre apenas no aspecto da performance de tempo, mas também no tamanho de bytes gerados, sendo muito maiores que os criados através de uma serialização própria. Também, essa grande quantidade de bytes aumenta o tempo de transmisssão de dados, tanto na comunicação entre os programas quanto na comunicação MPI. Ainda, a existência do programa paralelo e a necessidade de transferir os dados, dita comunicação entre processos, gera uma perda para esta solução. Portanto, deve-se analisar melhor esta alternativa e, talvez, considerar uma completa integração do MPI no sistema INSANE. Por último, a montagem de matrizes ainda é realizada de forma ineficiente, guardando muitos elementos de valor zero, já que se trata de uma matriz esparsa. Provavelmente uma melhoria de representação das matrizes esparsas trará melhores resultados.

Já a implementação da biblioteca METIS apresentou bons resultados, seu tempo de execução através de escrita e leitura de arquivos não representou um grande problema para a implementação de modo geral. A malha da Figura 11, utilizada de exemplo, contém 3122 elementos divididos em 4 partições com 800, 759, 777 e 786 elementos cada. No entanto, espera-se que haja uma melhora significativa com a implementação JNI da biblioteca.

Portanto, espera-se que com a melhoria destes problemas constatados e a adição de um *solver* paralelo, esta implementação apresente a vantagem de possibilitar a montagem problemas completamente em paralelo. Com isso, serão viabilizados estudos de modelos, em *cluster*, com um grande número de elementos, numa situação que seja impossível para computadores comuns.

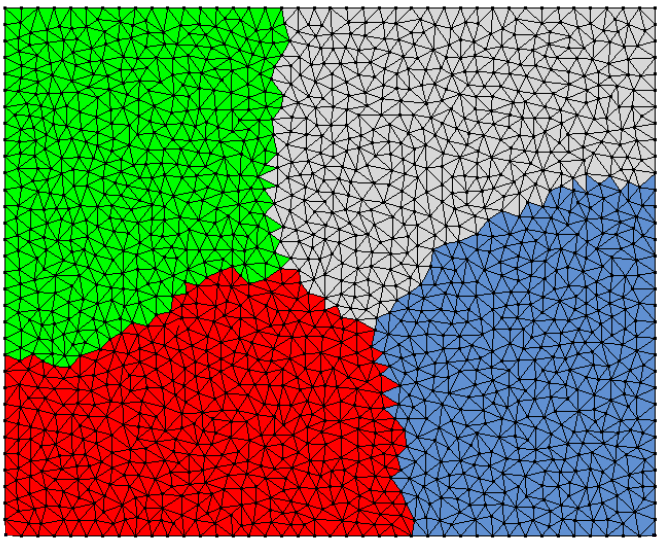

Figura 11. Método *init()* da classe *FemAssemblerPar*.

Fonte: (Autores, 2018)

AGRADECIMENTOS

Os autores agradecem o apoio financeiro em forma de fomento à pesquisa concedido pela FAPEMIG (Fundação de Amparo à Pesquisa do Estado de Minas Gerais) e pela CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

REFERÊNCIAS

Adeli H., Kamat M.P., Kulkarni G., & Vanluchene R.D, 1993. High-performance computing in structural mechanics and engineering. J. *Aerospacial Engineering*, vol. 6, pp. 249–26.

Devine, K. D., Boman, E.G., Heaphy, R. T., Bisseling, R. H., & Catalyurek, U. V., 2006. Parallel hypergraph partitioning for scientific computing. *International Parallel & Distributed Processing Symposium*.

Fonseca, F. T., 2008. Sistema computacional para a análise dinâmica geometricamente não-linear através do método dos elementos finitos. *Dissertação de mestrado*, Universidade Federal de Minas Gerais.

Gummadi, L. N. B., & Palazotto, A. N, 1997. Nonlinear finite element analysis of beams and arches using parallel processors. *Computers & Structures*, vol. 63, n. 3, p. 413428.

Karypis G., & Kumar, V., 1999. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, vol. 20, n. 1, pp. 359–392.

Pellegrini, F. & Roman, J., 1996. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *HPCN-Europe*, pp. 493–498. Springer LNCS.

Sanders, P., & Schulz, C., 2013. Think locally, act globally: highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms*, v. 7933, pp. 164–175. Springer LNCS.

Schloegel, L., Karypis, G., & Kumar, V., 2003. Graph partitioning for high-performance scientific simulations. In: Dongarra, J., Foster, I., Fox, G.; Gropp, W., Kennedy, K., Torczon, & L., White, A., eds, *Sorcebook of parallel computing*, cap. 18, pp. 491–540. Morgan Kaufmann Publishers.

Schloegel, K., Karypis, G., & Kumar, V., 2000. Parallel multilevel algorithms for multi-constraint graph partitioning. *Euro-par*, pp. 296-310.

Shafi A., Carpenter B., & Baker M, 2009. Nested parallelism for multi-core HPC systems using java. *Journal o Parallel and Distributed Computing*, vol. 69, n. 6, pp. 532–545.

Silva, L. L., 2016. Sistema gráfico interativo para análise de nucleação e propagação de trincas. *Dissertação de mestrado*, Universidade Federal de Minas Gerais.