MeshTools: uma ferramenta de manipulação de malhas de elementos finitos com foco em alto desempenho

Guilherme M. F. Silva[1], **José J. Camata**[1]

[1]Universidade Federal de Juiz de Fora (UFJF)
Caixa Postal 20.010 – 36.036-230 – Juiz de Fora – MG – Brazil.

{guilhermemachado,camata}@ice.ufjf.br

Abstract. *Several physical phenomena and/or problems in engineering and science are modeled by partial differential equations. These equations can be solved using numerical methods, such as finite differences, finite elements, and finite volume. In common, these methods require some form of discretization of the problem domain, that is, it is necessary to determine specific points in the domain where the solution of the differential equation will be calculated. High-resolution discretizations allow obtaining solutions with high numerical precision. On the other hand, they can demand high computational power. Thereby, there is a need to use parallelism for timely execution. In this work, it will be focused on the implementation of computational framework that handles finite element meshes for shared and distributed parallel systems. Performance results demonstrate good parallel scalability.*

Resumo. *Diversos fenômenos físicos e/ou problemas da Engenharia e ciências são modelados por equações diferenciais parciais. Essas equações podem ser solucionadas através de métodos numéricos, tais como, diferenças finitas, elementos finitos e volumes finitos. Em comum, esses métodos requerem alguma forma de discretização de domínio do problema, ou seja, é preciso determinar pontos específicos do domínio onde a solução da equação diferencial será calculada. Discretizações de alta resolução permitem obter soluções com alta precisão numérica. Por outro lado, podem demandar alto poder computacional. Dessa forma, torna-se necessária a utilização do paralelismo para uma execução em tempo hábil. Este trabalho, tem como foco a implementação de um arcabouço computacional que prepara uma malha de elementos finitos para o processamento paralelo em sistemas de memória compartilhada e distribuída. Resultados de desempenho demonstram uma boa escalabilidade paralela.*

1. Introdução

Malhas de elementos finitos são processadas em diversas aplicações através do Método dos Elementos Finitos, uma técnica amplamente utilizada para solução computacional de equações diferenciais parciais [Hughes 2012]. Uma malha corresponde a discretização geométrica de um domínio, subdividindo-o em pequenas partes, denominados elementos, os quais passam a representar o domínio contínuo do problema de interesse. Devido a necessidade recorrente de resultados mais precisos, malhas mais refinadas, ou seja, com maior discretização espacial são imprescindíveis. Entretanto, o processamento dessas malhas em tempo hábil requer o emprego de técnicas de otimização tanto na definição da estrutura de armazenamento quanto nos algoritmos para sua manipulação.

O advento de sistemas computacionais de alto desempenho possibilitou o desenvolvimento de diversas aplicações que aproveitassem tanto o paralelismo em memória compartilhada quanto o paralelismo em memória distribuída embarcados nestes sistemas. No paralelismo de memória compartilhada, a memória pode ser acessada simultaneamente por diferentes fluxos de execução (*threads*) de uma dada aplicação. Contudo, é essencial contornar a condição de corrida, ou seja, a possibilidade de diferentes *threads* manipularem simultaneamente um mesmo endereço de memória, podendo assim, causar inconsistências nos cálculos computacionais [Netzer and Miller 1992]. Já em sistemas com memória distribuída, as estruturas de dados devem ser particionadas e distribuídas entre os processos. Para obter uma solução eficiente, o processo de particionamento deve maximizar o balanceamento de carga e minimizar o custo da comunicação entre os processos.

Neste trabalho é proposto a ferramenta de código-livre `MeshTools`, um arcabouço computacional escrito em C++ que permite pré-processar uma malha de elementos finitos preparando-a para o uso de sistemas computacionais híbridos massivamente paralelos. `MeshTools` faz uso de diferentes esquemas de reordenação nodal para melhoria de localidade dos dados na memória, incorpora a biblioteca estado-da-arte METIS [Karypis and Kumar 1997] para o particionamento de domínio e também implementa o algoritmo de coloração [Guo et al. 2015] para permitir o processamento da malha em ambientes de memória compartilhada.

O restante desse artigo está organizado da seguinte maneira. Seção 2, discute-se as principais técnicas de alto desempenho aplicadas em malhas de elementos finitos. Na seção 3, detalhes de implementação da ferramenta `MeshTools` são apresentados. Resultados de desempenho e escalabilidade são apresentados na Seção 4. Finalmente, na Seção 5 é discutida as principais conclusões do estudo e próximas etapas do trabalho.

2. Técnicas de alto desempenho voltadas ao Método de Elementos Finitos

Através do Método dos Elementos Finitos (MEF) é possível resolver diversos problemas físicos de interesse modelados através de Equações Diferenciais Parciais. Como qualquer método numérico, o MEF também tem sua precisão associada a resolução da malha usada na discretização do domínio real. A malha é representada por três arranjos principais: *coord*, *conn* e *offset*. O vetor de *coord* armazena as coordenadas nodais x, y, z de cada ponto da malha no espaço. O vetor de *conn* armazena os nós que delimitam cada elemento da malha, ou seja, as conectividades de cada elemento. E por fim, o vetor *offset* indica a posição inicial de cada elemento no vetor de conectividades.

Figura 1 exemplifica a representação de uma malha referente à geometria de uma artéria. Note que, o nó 1, em azul na Figura 1, tem suas coordenadas x, y e z armazenadas nas três primeiras posições respectivamente do arranjo de coordenadas. Já o nó 2 tem suas informações guardadas nas três próximas posições. Para a representação dos elementos da malha, utilizam-se os arranjos de conectividades e *offset* em conjunto. Para o elemento 1, em vermelho na Figura 1, armazena-se os nós delimitadores 1, 2 e 5 nas três posições sequencias do vetor *conn* . Agora, para o elemento 2, os valores 2, 3 e 5 são armazenados no arranjo de conectividade nas três próximas posições. Porém, como em uma malha podem haver diferentes tipos de elementos como triângulos, quadrados, tetraedros ou ainda hexaedros, os quais têm diferentes números de nós delimitantes, torna-se necessário a

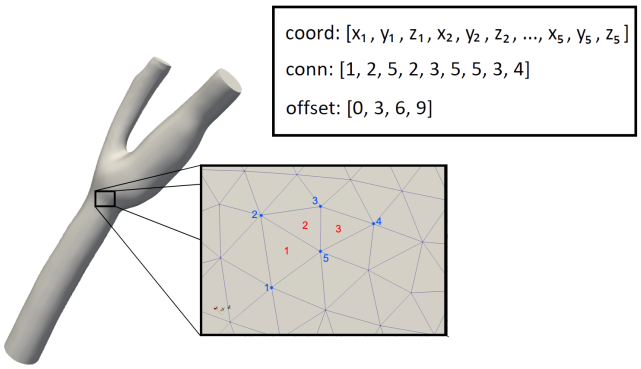


Figura 1. Representação computacional de uma malha de elementos finitos.

utilização do arranjo auxiliar *offset* que mapeia os elementos no vetor de conectividades. Logo, nesse exemplo da Figura 1, o arranjo *offset* é preenchido com os valores 0, 3, 6 e 9 indicando que o primeiro elemento inicia com suas conectividades na posição 0 do vetor conectividades e termina na posição 2. Já, para o elemento 2 são armazenadas suas conectividades nas posições 3, 4 e 5 do vetor de conectividades. E por fim, o elemento 3 tem suas conectividades guardadas da posição 6 até a posição 8 do vetor de conectividades.

Em uma aplicação típica de elementos finitos, percorrer a lista de elementos para a montagem de um sistemas de equações algébricas e sua solução posterior são as operações mais frequentes. Otimizar essas etapas é algo desejado. Em malhas com altas resoluções espaciais, a utilização do poder dos supercomputadores e o desenvolvimento de algoritmos paralelos escaláveis tornaram-se essenciais para manter o tempo de execução do método em níveis razoáveis. Nesse sentido, técnicas computacionais de alto desempenho que empregam esquemas que visam melhorar a localidade dos dados dentro da hierarquia de memória e o processamento em sistemas *multithreads* e distribuídos devem ser considerados. Essas técnicas serão abordadas nas subseções seguintes.

2.1. Reordenação

A reordenação nodal consiste na técnica de renumerar nós da malha de tal forma que os elementos tenham conectividades numericamente mais próximas. Um dos principais objetivos é reduzir a largura de banda das matrizes geradas pelo método. Essa reorganização aumenta a eficiência da busca de informações na hierarquia de memória pelo fato de aprimorar a localidade dos dados, reduzindo erros de acesso a cache e melhorando o desempenho global da aplicação. Um dos benefícios traduz na redução do esforço de execução do produto esparso matriz-vetor em solucionadores de sistemas lineares.

Considerando N o número total de nós e E o número total de elementos em uma malha, esse trabalho considera as seguintes técnicas de renumeração:

1. *First Touch*: neste esquema os nós são renumerados à medida que eles são acessados. Assim, para cada elemento $e \in [1, E]$ recupera-se sua conectividade nodal C. Para cada nó em C atribua um novo número caso ainda não tenha sido atribuído.
2. *Reverse Cuthill-McKee* : o algoritmo *Cuthill-McKee* reduz o tamanho da largura de banda por meio da reordenação do grafo de adjacências da matriz. Os nós adjacentes a um nó visitado são sempre percorridos em ordem crescente de grau. Posteriormente, observou-se que a reversão da ordem de *Cuthill-McKee* produz

um melhor esquema de permutação para problemas de reordenamento de matrizes [Cuthill and McKee 1969].

3. *Nested-Dissection*: o algoritmo transforma a malha em um grafo nodal que é dividido em dois subgrafos disjuntos a partir de um conjunto de vértices separadores. Por fim, seus subgrafos são reordenados recursivamente aplicando o algoritmo *Minimum Degree* [Teng 1997, Hendrickson and Rothberg 1998].

Figura 2 ilustra a largura de banda da malha da artéria (Figura 1) para os diferentes esquemas de reordenação. Na prática cada posição (i, j) em preto na matriz de adjacência indica uma posição ocupada por elementos não nulos. Nota-se, que o RCM é o que produz a maior redução da largura de banda quando comparado com a ordenação original (Fig. 2 (a)).

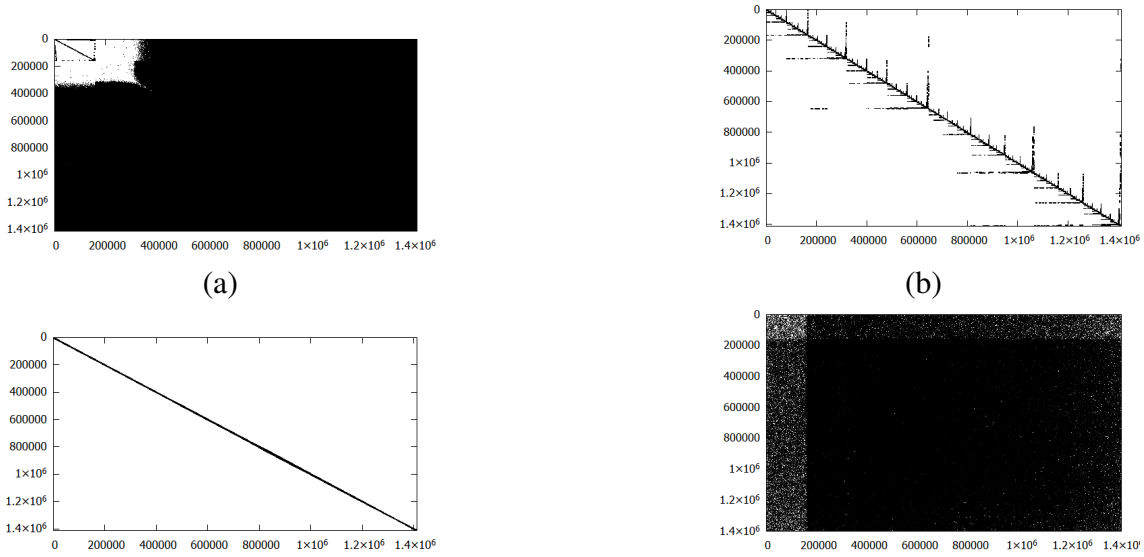


Figura 2. Matriz de Adjacência dos nós da malha artéria sendo (a) ordenação nodal original, (b) após a aplicação com algoritmo *Nested-Dissection*, (c) após a reordenação com algoritmo *Reverse Cuthill-McKee* e (d) após o algoritmo *First Touch*.

2.2. Particionamento de domínio

Devido à utilização de malhas cada vez mais refinadas, os requisitos de memória tornam-se elevados ocasionando uma limitação para a aplicação do Método dos Elementos Finitos em sistemas seriais. Diante disso, torna-se necessário o uso de técnicas de paralelismo em sistemas de memória distribuída. Nesse sentido, na computação paralela distribuída, o domínio original precisa ser particionado e enviado a processos diferentes para que sejam resolvidos concorrentemente. O desempenho está diretamente relacionado ao particionamento do domínio a qual deve visar uma bom balanceamento de carga entre os processos e minimizar o número total de nós de interface entre as partições, fazendo que o custo de sincronização entre os processos seja minimizado.

Neste trabalho utiliza-se a biblioteca de código-livre METIS [Karypis and Kumar 1997]. METIS é uma biblioteca de particionamento de grafos, malhas e matrizes. Destaca-se por utilizar algoritmos otimizados que permitem o

particionamento de malhas não estruturadas além de possuir rotinas para a otimização de largura de banda em matrizes esparsas. O METIS trata o particionamento de malha como um problema de particionamento de grafo. Duas metodologias podem ser empregadas: particionamento do grafo dual e particionamento do grafo nodal. Na primeira técnica, cada elemento da malha torna-se um vértice do grafo enquanto que, na segunda técnica, cada nó da malha torna-se um vértice do grafo. Para este trabalho, o particionamento empregado usa o grafo nodal. A partir das informações do particionamento, determina-se o mapa de comunicação entre as partições a qual relaciona os nós que são compartilhados por mais de uma partição.

2.3. Coloração

O advento de sistemas computacionais com múltiplos núcleos possibilitou o desenvolvimento de diversas aplicações que aproveitassem o paralelismo em memória compartilhada embarcado nestes sistemas. Entretanto, inconsistências nos cálculos computacionais podem ser ocasionados devido ao acesso simultâneo à memória por diferentes fluxos de execução, caracterizando, assim, uma condição de corrida entre as *threads* [Netzer and Miller 1992].

Dessa forma, a utilização do paralelismo de memória compartilhada requer certos cuidados uma vez que elementos vizinhos compartilham mesmas posições de memória. Uma das formas de contornar tal situação é a aplicação da técnica de coloração de grafos [Guo et al. 2015]. Nessa técnica, uma malha de elementos finitos é tratada como um grafo onde os elementos são os vértices. Os vértices são numerados a partir das informações das cores dos seus vértices adjacentes. Esse valor representa uma cor. Com a finalização do algoritmo, vértices vizinhos tem cores diferentes possibilitando então a paralelização de elementos com mesma cor e dessa forma contornando a inconsistência ocasionada pela condição de corrida. Elementos vizinhos são definidos como quaisquer par de elementos com pelo menos um nó compartilhado. Após a aplicação da técnica de coloração, os elementos são reordenados no *array* de conectividades posicionando elementos de mesma cor de forma consecutiva visando melhor utilização da memória cache.

Para a aplicação da técnica de coloração é necessário obter um grafo dual a partir da malha, ou seja, um grafo com as informações das adjacências dos elementos, o qual tem como vértices os elementos da malha e arestas as adjacências entre esses elementos. Figura 3 representa um domínio quadrado com dimensões unitárias $[0, 1] \times [0, 1]$ discretizado por uma malha estruturada de elementos finitos com 3 subdivisões nos eixos x e y, totalizando nove elementos do tipo quadrilátero e seu respectivo grafo dual.

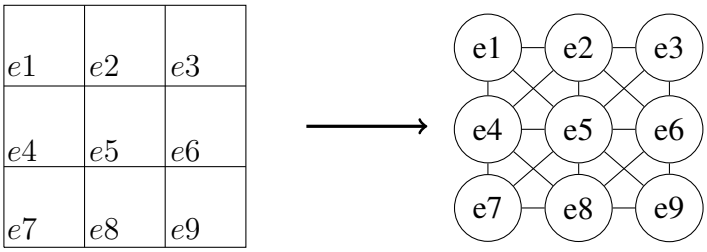


Figura 3. Malha de elementos finitos e sua representação em um grafo dual.

O problema de coloração de malhas é classificado como um problema NP-

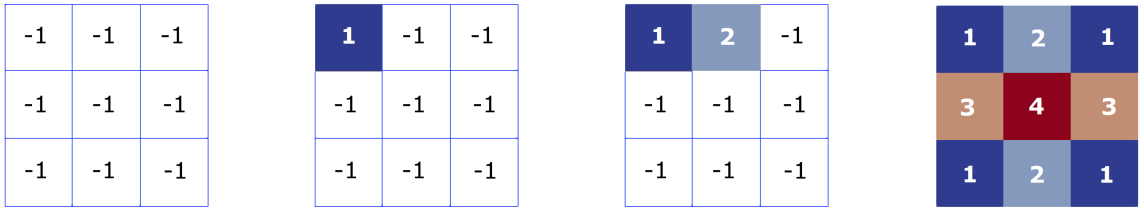
Completo [Garey and Johnson 1990]. Entretanto, existem inúmeras heurísticas que permitem obter uma solução não ótima em tempo polinomial. Um dos métodos mais simples é o algoritmo conhecido como *Greedy* [Rokos et al. 2015]. Neste esquema, o método visita os vértices do grafo atribuindo a menor cor possível, isto é, a menor cor ainda não atribuída ao vértices vizinhos. O pseudo-código é apresentado no Algoritmo 1. O algoritmo tem como entrada um Grafo G com V vértices e E arestas e retorna uma lista c contendo as cores atribuídas para todos os vértices de G.

Algoritmo 1: *Greedy*

Entrada: G - Grafo com V vértices e E arestas
Saída: c - Arranjo com a coloração dos vértices

Para $\forall v_i \in V$ **Faça**
  $C \leftarrow$ {Cores de todos os vértices coloridos $v_j \in adj(v_i)$} ;
  $c(v_i) \leftarrow$ {menor cor $\notin$ C} ;
Fim-Para
retorne c;

O algoritmo inicia atribuindo para cada elemento (vértice no grafo dual) um valor que indica a ausência de cor (-1), como pode ser visto na Figura 4(a). As cores que podem ser atribuídas para um elemento variam entre 1 até, no máximo, o número de elementos da malha. Para o primeiro elemento, Figura 4(b), verifica-se as cores dos seus elementos adjacentes que, neste caso são elementos que não têm cor. Assim, é escolhido a menor cor possível, representado pelo valor 1. Para o segundo elemento, Figura 4(c), novamente é verificado as cores dos seus elementos adjacentes, sendo um com cor 1 e os outros sem cor, logo a menor cor possível para esse elemento é representado pelo valor 2. Note que o valor 1 não seria possível pois quebraria a regra de não ter elementos vizinhos com a mesma cor. No final da execução do algoritmo, obtém a malha colorida representada pela Figura 4(d).



(a) (b) (c) (d)

Figura 4. Malha bidimensional exemplar.

3. `MeshTools`

O `MeshTools` é um arcabouço computacional para manipulação de malhas de elementos finitos desenvolvido com o intuito de possibilitar a utilização de Computação de Alto Desempenho durante simulações utilizando o Método dos Elementos Finitos. Para tanto, aplica-se as técnicas como reordenação nodal, particionamento de domínio e coloração discutidas anteriormente. A Figura 5 demonstra o fluxograma de execução do `MeshTools`.

`MeshTools` pré-processa malhas no formato `Gmsh` (versão 2.0). O `Gmsh` [Geuzaine and Remacle 2009] é um software *open-source* de geração e visualização de

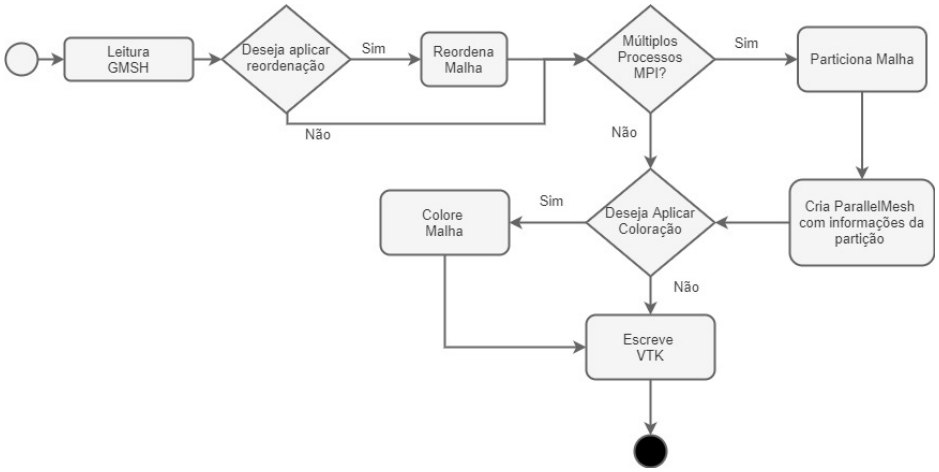


Figura 5. Fluxograma de execução do `MeshTools`.

malhas de elementos finitos tridimensionais, além de disponibilizar um solver acoplado e um módulo de pós-processamento pelo qual é possível analisar o resultado da solução do problema na malha gerada. O `Gmsh` também permite a geração e utilização de modelos CAD, facilitando o reuso de malhas.

`MeshTools` pode ser executado tanto sequencialmente quanto em múltiplos processos MPI. O processo mestre é responsável pela leitura da malha original. No passo seguinte, o mesmo processo aplica uma das técnicas de reordenação nodal selecionado pelo usuário da aplicação, sendo elas: *Natural* (mantém a ordenação original do `Gmsh`), *First-Touch*, *Reverse Cuthill-McKee* ou ainda *Nested Dissection* que é oferecida pela biblioteca METIS [Karypis and Kumar 1997].

Se o `MeshTools` estiver sendo executado por mais de um processo MPI, o módulo de particionamento de domínio será executado . Neste módulo, o processo mestre chama a rotina `METIS_PartMeshNodal` também disponibilizado pela biblioteca METIS. Para cada partição, o processo mestre obtém as informações locais da malha de uma determinada partição, constrói o mapa de comunicação e envia os dados ao processo correspondente. Os demais processos por sua vez, ao receber suas informações locais, inicializam suas estruturas de dados e o mapa de comunicação. Após essa etapa, os processos possuem as informações necessárias que permitem a comunicação via rotinas MPI.

Por fim, cada processo aplica o algoritmo de coloração em sua malha local. `MeshTools` implementa os métodos *Greedy* e *Greedy Blocked*. Este segundo corresponde ao algoritmo *Greedy* limitando o número máximo de elementos inseridos em cada cor. Há ainda a versão OpenMP do algoritmo desenvolvido por *Rokos* [Rokos et al. 2015]. Neste momento, a conectividade dos elementos serão agrupados por cores. Os primeiros c_0 elementos serão aqueles marcados com a cor 0, seguidos pelos c_1 elementos marcados com a cor 1, etc. Após essa etapa, todos elementos em cada grupo de cores c_i podem ser acessados paralelamente em ambientes *multithreads*.

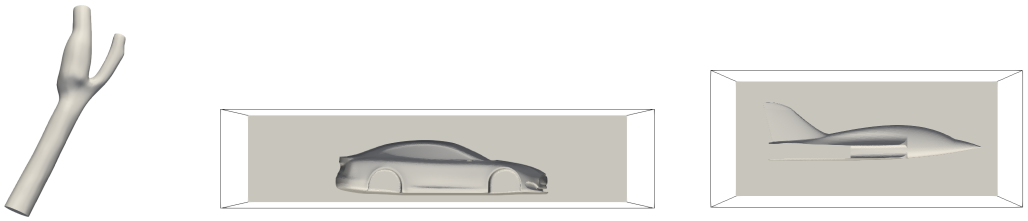
Como saída, `MeshTools` permite a escrita em formato VTK [Hanwell et al. 2015]. Sendo o `MeshTools` executado por mais de um processo, essa escrita da malha pode ser feita em um arquivo VTK paralelo. O `MeshTools` pode ser encontrado no GitLab, um site para controle de versionamento e compartilhamento de

códigos, através do seguinte link: `https://gitlab.com/camata/meshtools`.

4. Análise de Desempenho

Os testes de desempenho foram executados no supercomputador Lobo Carneiro presente na Coppe/UFRJ. Este *cluster* é composto por 504 CPUs Intel Xeon E5-2670v3. O nó computacional é composto por dois CPUs com 12 *cores* físicos cada, sendo possível então utilizar até 48 *cores* com o auxílio da tecnologia *Hyper-Threading*. Cada nó possui também 64GB de memória RAM. Para compilação foi utilizado do GNU versão 9.3.0 com as flags de compilação `-O3 -fopenmp` e a biblioteca MPI OpenMPI versão 4.1.1.

As malhas utilizadas para os testes são malhas realísticas representadas na Figura 6, e podem ser aplicáveis ao estudo da Mecânica dos Fluidos. As três malhas foram geradas no `Gmsh` utilizando arquivo CAD [1] como base. A malha do carro é composta por 11.451.904 elementos e 2.095.626 nós, do avião tem 35.422.208 de elementos com 17.702.593 de nós, e finalmente da artéria é composta por 65.568.768 de elementos e 11.251.681 de nós.



(a) (b) (c)

Figura 6. Malhas utilizadas nos testes sendo (a) artéria, (b) carro e (c) avião.

Para verificarmos os ganhos de desempenho com o paralelismo, usou-se a etapa de montagem do sistemas matricial oriunda da formulação variacional do método de elementos finitos para a solução da Equação de Laplace, representado pelo Algoritmo 2. Essa etapa, juntamente com a solução do sistema linear (ou não-linear) resultante da discretização, constituem as principais demandas computacionais de um solucionador de elementos finitos.

Algoritmo 2: Montagem da matriz com coloração

Entrada: A - Matriz Global
M - Malha de elementos finitos
c - Coloração
Saída: A - Matriz Global atualizada

$A \leftarrow 0$;
Para $k \leftarrow 1$ *até* N_{cores} **Faça**
  $L_e \leftarrow \{\forall e \in M | c(e) \equiv k\}$;
  Para $\forall e \in L_e$ **Faça**
    A += A_e;
  Fim-Para
Fim-Para
retorne A;

[1] Website: `https://grabcad.com/`

Para a verificação da escalabilidade paralela, a métrica de desempenho utilizada na execução do *Assembly* paralelizado foi o *speedup*, o qual representa o quanto a paralelização do código escalou em relação ao número de *threads* e/ou processos MPI utilizados [Rodgers 1985]. Essa métrica é formulada na Equação 1 em que é feita a divisão do tempo de execução em série pelo tempo de execução em paralelo. Para uma máquina paralela com p processadores, *speedup* ideal seria p.

$$Sp = \frac{TempoSerial}{TempoParalelo} \tag{1}$$

As Tabelas 1, 2, 3 e 4 mostram o tempo de execução do *Assembly*, representado no Algoritmo 2, e o *speedup* para as três malhas testes após a aplicação das técnicas de otimização.

4.1. Execução puramente MPI

Foi considerado duas configurações de alocação de processos. A primeira todos os processos MPI são alocados em um único nó computacional até 16 processos (Tab. 1). Já a segunda configuração considerou-se a alocação em dois nós computacionais (Tab. 2). O objetivo dessa análise é verificar se a utilização dos recursos de memória e rede afetariam o desempenho da aplicação.

Pelas Tabelas 1 e 2 é possível visualizar que em todas as malhas o *speedup* foi próximo do linear. Para 16 processos é observado um *speedup* superlinear. O *speedup* superlinear ocorre quando os dados casam com o tamanho da memória cache. Além disso, outra explicação se deve a característica do processo de *assembly* onde não há a necessidade de comunicação interna com sincronização entre os processos. Também é possível observar que alocação de processos MPI em nós distintos não reduz o tempo significativamente em relação à alocação em um mesmo nó. Essa alocação em nós distintos pode ser benéfica quando os requisitos de memória por processo forem maiores devido a alocação de estruturas de dados auxiliares em aplicações de elementos finitos como matrizes robustas.

Tabela 1. Tempo de execução do *Assembly* em um nó computacional, com diferentes números de processos MPI sem paralelismo em memória compartilhada.

	Carro		Artéria		Avião	
Nós × MPI Processos	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*
1 × 1	1.56101	1.00	8.89359	1.00	3.22057	1.00
1 × 2	0.84614	1.84	4.55255	1.95	1.62987	1.98
1 × 4	0.42504	3.67	2.39338	3.72	0.58783	5.48
1 × 8	0.16132	9.68	1.11512	7.98	0.40376	7.98
1 × 16	0.0807142	19.34	0.477092	18.64	0.198259	16.24

4.2. Execução puramente OpenMP

A Tabela 3 mostra os dados de desempenho para a execução puramente com OpenMP usando diferentes números de *threads*. Observa-se para todas as malhas que o *speedup* ficou abaixo do linear. Isso se deve aos custos de criação da região paralela, a necessidade de utilizar barreiras de sincronização entre as *threads* e na latência de utilização do barramento CPU/memória. Diante disso, é possível perceber um maior ganho de eficiência

Tabela 2. Tempo de execução do *Assembly* em dois nós computacionais, com diferentes números de processos MPI sem paralelismo em memória compartilhada.

	Carro		Artéria		Avião	
Nós × MPI Processos	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*
2×1	0.68156	2.29	4.44417	2.00	1.60670	2.00
2×2	0.35178	4.44	2.61384	3.40	0.86447	3.73
2×4	0.13994	11.15	1.11523	7.97	0.40392	7.97
2×8	0.07931	19.68	0.53221	16.71	0.19922	16.17

entre 6 e 12 *threads* e entre 12 e 24 *threads*. Porém entre 24 e 48 *threads* obtém-se um menor incremento do *speedup*. Esse comportamento pode ser explicado pela arquitetura dos processadores intel disponíveis no *cluster* que possuem somente 24 *cores* físicos e até 48 *cores* virtuais através da tecnologia *Hyper-Threading*.

Tabela 3. Tempo de execução do *Assembly* em um nó computacional, em um processo MPI com paralelismo em memória compartilhada.

	Carro		Artéria		Avião	
Threads	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*
1	1.56101	1.00	8.89359	1.00	3.22057	1.00
6	1.19496	1.31	6.86799	1.29	2.30817	1.40
12	0.18440	8.47	1.26331	7.04	0.31140	10.34
24	0.16261	9.60	0.71765	12.39	0.18200	17.70
48	0.15413	10.13	0.59307	15.00	0.16964	18.98

4.3. Execução híbrida: MPI × OpenMP

A Tabela 4 mostra as métricas de processamento das malhas utilizando tanto o paralelismo com MPI quanto com OpenMP. Foram analisadas diferentes configurações de alocação número de nós × números de processos MPI × número de *threads* de tal modo que fossem alocados todos os *cores* no nó computacional. Nesse caso é possível visualizar que temos melhor desempenho com utilização de 1 nó, em 8 processos MPI e 6 *threads*. Considerando dois nós, o melhor desempenho foi com 4 processos MPI e seis *threads*. É observado também que a relação entre alocação de um ou dois nós não altera de forma significativa o desempenho da aplicação. Veja que o *speedup* para configuração $1 \times 8 \times 6$ é similar à configuração com $2 \times 4 \times 6$. Considerando a alocação completa dos nós computacionais ($2 \times 8 \times 6$), foi alcançado *speedup* superior a 21 em todas as malhas.

5. Conclusão

A manipulação eficiente de malhas de elementos finitos em aplicações numéricas voltadas para solução computacional de equações diferenciais parciais é desejável. Um ponto de ganho de desempenho dessas aplicações é adaptar os algoritmos sequencias para arquiteturas computacionais híbridas que permitem paralelismo em memória compartilhada e distribuída. Nesse sentido, a paralelização de laços que varrem as lista de elementos é uma primeira estratégia a ser adotada. Para sistemas de memória compartilhada, a aplicação de esquema de coloração contorna a condição de corrida que é devida ao compartilhamento

Tabela 4. Tempo de execução do *Assembly* em diferentes nós computacionais, em oito processos MPI com paralelismo em memória compartilhada.

	Carro		Artéria		Avião	
Nó × MPI × *Threads*	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*	*CPU Time*	*Speedup*
1 × 2 × 24	0.629348	2.48	3.46573	2.57	1.167360	2.76
1 × 4 × 12	0.31161	5.01	1.902910	4.67	0.588603	5.47
1 × 8 × 6	0.134477	11.61	0.862280	10.31	0.303395	10.62
2 × 1 × 24	0.556521	2.80	3.436360	2.59	1.16589	2.76
2 × 2 × 12	0.279820	5.58	2.316790	3.84	0.67518	4.77
2 × 4 × 6	0.14199	10.99	0.85141	10.45	0.29498	10.92
2 × 8 × 6	0.069114	22.59	0.423105	21.02	0.145277	22.17

de informações nodais entre elementos vizinhos. Além disso, a necessidade de obter soluções em alta fidelidade requer a utilização de malhas com altas resoluções. Devido ao fator limitante da capacidade da memória em nós computacionais, torna-se necessário o particionamento do domínio para divisão dessa execução entre diferentes nós. Nesse sentido, esse trabalho apresenta o `MeshTools`, uma ferramenta *open-source* implementada em C++ que processa uma malha de elementos finitos, preparando-a para ambientes de alto desempenho. O estudo de desempenho em umas das operações centrais em solucionador de elementos finitos demonstram ganhos consideráveis de eficiência. Pretende-se futuramente acoplar ao `MeshTools` solucionadores lineares paralelos, tais como PETSc [Balay et al. 2021] e Trilinos [Trilinos Project Team 2021]. Além disso, deseja-se otimizar o armazenamento em disco com a leitura e escrita de dados com compressão, sem e com perdas [Fox et al. 2020].

Agradecimento

A Pró-Reitoria de Pós-Graduação e Pesquisa e ao seu programa de Bolsas de Iniciação Científica da Universidade Federal de Juiz de Fora (BIC/UFJF) e a FAPEMIG.

Referências

[Balay et al. 2021] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. 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., and Zhang, H. (2021). PETSc Web page. https://petsc.org/.

[Cuthill and McKee 1969] Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA. Association for Computing Machinery.

[Fox et al. 2020] Fox, A., Diffenderfer, J., Hittinger, J., Sanders, G., and Lindstrom, P. (2020). Stability analysis of inline zfp compression for floating-point data in iterative methods. *SIAM Journal on Scientific Computing*, 42(5):A2701–A2730.

[Garey and Johnson 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman Co., USA.

[Geuzaine and Remacle 2009] Geuzaine, C. and Remacle, J.-F. (2009). Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331.

[Guo et al. 2015] Guo, X., Lange, M., Gorman, G., Mitchell, L., and Weiland, M. (2015). Developing a scalable hybrid mpi/openmp unstructured finite element model. *Computers Fluids*, 110:227–234. ParCFD 2013.

[Hanwell et al. 2015] Hanwell, M. D., Martin, K. M., Chaudhary, A., and Avila, L. S. (2015). The visualization toolkit (vtk): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1-2:9–12.

[Hendrickson and Rothberg 1998] Hendrickson, B. and Rothberg, E. (1998). Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489.

[Hughes 2012] Hughes, T. J. (2012). *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation.

[Karypis and Kumar 1997] Karypis, G. and Kumar, V. (1997). Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.

[Netzer and Miller 1992] Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions? some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88.

[Rodgers 1985] Rodgers, D. P. (1985). Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231.

[Rokos et al. 2015] Rokos, G., Gorman, G., and Kelly, P. H. (2015). A fast and scalable graph coloring algorithm for multi-core and many-core architectures. In *European Conference on Parallel Processing*, pages 414–425. Springer.

[Teng 1997] Teng, S.-H. (1997). Fast nested dissection for finite element meshes. *SIAM Journal on Matrix Analysis and Applications*, 18(3):552–565.

[Trilinos Project Team 2021] Trilinos Project Team (2021). The trilinos project website: https://trilinos.github.io (accessed august 17, 2021).