Lajer - Latin American Journal of Energy Research
www.periodicos.ufes.br/lajer

Uma estratégia de implementação paralela eficiente de uma heurísticas de particionamento de grafos aplicado à simulação de escoamento multifásico

Leonardo Rogério Binda da Silva*, Roney Pignaton da Silva
Universidade Federal do Espírito Santo – UFES campus São Mateus, Rodovia BR 101 Norte, km 60, Litorâneo, CEP 29.932-540, São Mateus, ES, Brasil
*Autor para correspondência
Endereço eletrônico: leonardorbs@gmail.com (Silva, L.R.B.), roney.silva@ufes.br (Pignaton, R.S.)

O Problema de Particionamento de Grafos (PPG) possui várias aplicações em diferentes áreas, tal como no projeto de circuitos VLSI (Very-large-scale integration), resolução de métodos numéricos para simulação de problemas que incluem fatoração de matrizes esparsas e particionamento de malhas de elementos finitos para aplicação de programação paralela. Entre suas aplicações, o foco deste trabalho é o desenvolvimento de uma solução paralela para esse problema aplicado à simulação de fluxo multifásico em meios porosos para recuperação de petróleo. Como resultado, as partições criadas permitem particionar o espaço discretizado de maneira que os mesmos seja simulados em paralelo. O PPG tende a ser NP-difícil e soluções ótimas para o problema são impossíveis quando o número de vértices do grafo é muito grande. Muitas heuristicas e metaheurística já foram propostos e usados para resolver o PPG com o objetivo de alcançar bons resultados, uma vez que resultados garantidamente ótimos não são obtidos na prática. Este trabalho propõe uma solução paralela eficiente para o PPG baseado na implementação de heurísticas existentes em uma plataforma computacional paralela do tipo Cluster. A solução proposta melhora o tempo de execução do algoritmo e, através da introdução de algumas características de aleatoriedade na heurística original, melhora a qualidade das partições criadas.

Palavras-chave: Particionamento de Grafos. Algoritmos Paralelos. Simulação de reservatórios.

1. Introdução

O Problema do Particionamento de Grafos (PPG) é definido como a seguir: dados um grafo $G=(V, E)$, onde N é o conjunto de vértices com seus pesos atribuídos e E é o conjunto de arestas com seus pesos atribuídos e um inteiro positivo k, deve-se encontrar k subconjuntos N_1, N_2, ..., N_k, tais que (a) todos os vértices do grafo original estejam distribuídos entre os subconjuntos criados e esses subconjuntos sejam disjuntos, (b) a soma dos pesos dos vértices dos subconjuntos seja aproximadamente igual ao peso de todos os vértices dividido pelo número de subconjuntos k e (c) o *cut size*, ou seja, a soma dos pesos das arestas entre os subconjuntos seja minimizado (FJALLSTROM, 1998). O PPG-k é o problema de se encontrar k ($k > 1$) subconjuntos de vértices com o menor *cut size* possível. Para $k = 2$, em particular, o PPG encontra uma bisseção. Uma forma bem comum de se resolver esse problema é encontrando bisseções de maneira recursiva (AMARAL *et al*., 2010).

O PPG tem diversas aplicações práticas, tais como projeto de circuitos VLSI, fatoração de matrizes esparsas e particionamento de malhas de elementos finitos para aplicações de programação paralela (AMARAL *et al*., 2010). O foco deste trabalho é o desenvolvimento de uma solução paralela para esse problema aplicado à simulação de fluxo multifásico em meios porosos para recuperação de petróleo. O processo de simulação do escoamento multifásico implica a geração de malhas discretizadas muito grandes, resultando em um esforço computacional também muito grande. Com isso, soluções paralelas vem sendo aplicadas, surgindo a necessidade de particionamento do espaço discretizado de maneira a distribuir igualmente a carga de trabalho entre os nós de processamento, bem como diminuir a necessidade de comunicação entre os nós de processamento. Como resultado, as partições criadas permitem particionar o espaço discretizado de maneira que os mesmos seja simulados em paralelo Os PPG tendem a ser NP-difíceis (SCHAEFFER, 2007). Soluções

ótimas para a resolução dos mesmos são inviáveis quando o número de vértices do grafo é grande. Algoritmos heurísticos e metaheurísticos têm sido cada vez mais utilizados para a resolução do PPG, onde na prática, na impossibilidade de se encontrar soluções ótimas para grafos com muitos vértices, soluções boas podem ser aplicadas (AMARAL *et al.*, 2010).

O presente trabalho propõe uma solução paralela eficiente para o problema PPG com base na implementação de heurísticas existentes em um cluster computacional. A solução proposta permite melhorar o desempenho geral das heurísticas apresentadas em Bonatto (2010) em dois aspectos: 1) melhoria do tempo de execução, com um considerável aumento de velocidade de execução quando comparado com sua implementação serial, e 2) melhoria da qualidade das partições criadas, através da introdução de algumas características de aleatoriedade às heurísticas originais.

As seções seguintes deste artigo estão organizadas da seguinte forma. Na Seção 2, apresentamos as métricas de desempenho utilizadas para medir a qualidade da solução paralela. Em seguida, na Seção 3 , descrevemos os principais métodos gerais utilizados para categorizar algoritmos de Particionamento Grafos. Nas seções 4 e 5, apresentamos as heurísticas propostas por Bonatto (2010) e nossa solução paralela, com a descrição da estratégia de paralelização proposta, e na Seção 6, mostramos os resultados de nossas experiências e comparamos o desempenho da implementação paralela com outras heurísticas. Finalmente, concluímos com uma discussão comparativa sobre a nossa abordagem paralela.

2. Métricas de performance

Conforme mencionado, a análise de desempenho baseia-se em duas métricas. A primeira delas considera o tamanho de corte do gráfico particionado. Em teoria de grafos, um corte é um particionamento dos vértices de um grafo em dois subconjuntos disjuntos unidos por, pelo menos, uma aresta. O de tamanho de corte, também chamado *cut-size*, é o número de arestas cujos pontos finais estão em diferentes subgrupos da partição. Assim, o PPG aplicado a um problema de computação paralela pode ser usado para dividir a carga de processamento em cada nó de processamento, de tal forma que minimizar o *cut-size* pode representar a minimização da troca de mensagens entre os nós de processamento.

A segunda métrica utilizada para avaliação de performance é o *speedup* ou taxa de *speedup*. O *speedup* refere-se a quão mais rápido um algoritmo paralelo é quando comparado com sua implementação serial.

O *speedup* é definido através da Eq. (1) seguinte

$$S_p = \frac{T_1}{T_p} \tag{1}$$

onde: p é o número de nós de processamento; T_1 é o tempo de execução do algoritmo seqüencial; T_p é o tempo de execução do algoritmo paralelo com p processadores

Medidas de speedup podem ser utilizado para fornecer uma estimativa de quão bom um código paralelo é comparado com sua implementação serial, e também para gerar um gráfico de tempo versus nós de processamento com o objetivo de entender o código paralelizado para estudar a curva de eficiência de uma implementação paralela com o objetivo de determinar o ponto onde a mesma passa a ser uma solução sem retorno de investimento. Com essas informações, você seria capaz de determinar, para um problema de tamanho fixo, qual o número ideal de nós de processamento utilizar ou mesmo se a implementação paralela deve ser abandonada.

3. Algoritmos de particionamento de grafos

Um grande número de algoritmos de particionamento Gráfico foram propostos e estão disponíveis na literatura de ciência da computação, com diferente campos de aplicação. Esta seção apresenta uma descrição geral desses algoritmos, evidenciando os principais métodos gerais de solução usados para classificá-los.

3.1. Métodos geométricos

Técnicas geométricas de particionamento de grafos encontram partições baseadas apenas nas informações de coordenadas dos vértices do grafo e não na conectividade dos mesmos. Portanto, não há o conceito de corte de arestas nesses métodos e sim minimizar métricas, como por exemplo, o número de vértices do grafo que são adjacentes a outros vértices não-locais (tamanho da fronteira). Tendem a ser mais rápidos que os métodos espectrais, mas retornam partições com piores cortes. São exemplos o *Recursive Coordinate Bissection* (RCB) e o *Recursive Inertial Bissection* (RIB) Schloegel *et al*., (2001), Ou *et al*., (1997), Fjallstrom, (1998), Karypis et al., (1998), Kumar *et al*., (1998).

3.2. Métodos espectrais

Métodos espectrais não particionam o grafo lidando com o mesmo propriamente dito, mas com sua representação matemática. Os grafos são modelados pela relaxação do problema de otimização de uma função quadrática discreta, sendo transformada em uma função contínua. A minimização do problema relaxado é então resolvido pelo cálculo do segundo autovetor da discretização Laplaciana do grafo. Produzem boas partições para uma extensa classe de problemas. São exemplos o *Recursive Spectral Bisection* (*RSB*) e o *Multilevel Recursive Spectral Bisection* (*Multilevel RSB*) Schloegel et al., (2001), Karypis *et al*., (1998), Guattery *et al*., (1995), Qiu *et al*., (2006).

3.3. Métodos combinatórios

Os métodos combinatórios recebem como uma entrada uma bisseção de um grafo e tentam diminuir o corte das arestas pela aplicação de algum método local de busca. São exemplos de métodos combinatórios o Kernighan-Lin (KL) e o Fidducia e Mattheyeses (FM). Ambos os métodos tentam trocar vértices entre as partições na tentativa de diminuir o corte, com a diferença que o método KL troca pares de vértices, enquanto que o método FM troca um vértice apenas, alternando entre os vértices de cada partição Fjallstrom (1998), Fiduccia *et al*., (1982), Kernighan *et al*., (1970).

3.4. Métodos multiníveis

Os métodos multiníveis de particionamento consistem de três fases: contração, particionamento e expansão do grafo. Na fase de contração, uma série de grafos é construída unindo vértices para formar um grafo menor. Esse grafo contraído recém construído é contraído novamente, e assim sucessivamente até que um grafo suficiente pequeno seja encontrado. Uma bisseção sobre esse grafo é construída de maneira bem rápida, já que o grafo é pequeno. Durante a fase de expansão, um método de refinamento é aplicado a cada nível do grafo à medida que o mesmo é expandido. Métodos multiníveis estão presentes em vários pacotes de softwares, tais como Chaco, Metis e SCOTCH Schloegel *et al*., (2001), Fjallstrom, (1998), Schloegel *et al*., (2001).

3.5. Metaheurísticas

Uma metaheurística é um conjunto de conceitos que possam ser utilizados para definir métodos heurísticos que podem ser aplicados a um grande conjunto de diferentes problemas. Em outras palavras, uma metaheurística pode ser vista como um framework algorítmico geral que pode ser aplicado a diferentes problemas de otimização com relativamente poucas modificações para torná-los adaptado para um problema específico. Diversas metaheurísticas foram adaptadas para o particionamento de grafos, tais como recozimento simulado, busca tabu e algoritmos genéticos, Johnson *et al*., (1989), Rolland *et al*., (1996).

4. Heurísticas propostas

Bonatto (2010 propôs quatro heurísticas combinatórias para PPG-*k*, onde as três primeiras constroem uma *k*-partição do grafo por vez até atingir *k* subconjuntos. A quarta heurística é uma versão da terceira heurística que utiliza bisseções recursivas para atingir os *k* subconjuntos. Todas as heurísticas, após o particionamento do grafo, executam uma rotina de melhoramento para refinar o corte da partição.

A idéia básica é construir uma partição do grafo acumulando vértices em seus subconjuntos a partir de um determinado critério, sendo esse critério da escolha do vértice a ser adicionado ao novo subconjunto em formação exatamente o diferencial entre as três primeiras heurísticas, exceto para a quarta heurística que é uma aplicação recursiva da terceira heurística.

4.1. Heurística 1

A cada iteração do método de construção do subconjunto da Heurística 1, um vértice v é adicionado ao subconjunto p e os seus vértices adjacentes são inseridos em uma lista chamada frontier, que define a fronteira do subconjunto p com os demais subconjuntos. O vértice a ser inserido no subconjunto p é selecionado aleatoriamente entre os vértices da fronteira. Após a inserção do vértice, o valor do corte de arestas (cut size) é atualizado com o ganho g(v) do vértice.

4.2. Heurística 2

Na Heurística 1, os vértices são adicionados ao novo conjunto que está sendo construído sem critério algum, de forma totalmente aleatória. Na Heurística 2, a principal diferença é que agora cada vértice pertencente a fronteira do vértice v tem o seu ganho g(v) calculado e armazenado em ordem decrescente de valores de ganho em uma estrutura de dados chamada bucket. A cada passo de execução, o vértice com maior ganho será inserido no conjunto em expansão. Após a inserção desse vértice no novo subconjunto, os ganhos dos vértices adjacentes a esse vértices que fazem parte da fronteira são atualizados e os ganhos dos vértices que ainda não estão na fronteira são inseridos no bucket.

4.3. Heurística 3

Assim como na Heurística 2, a Heurística 3 calcula e armazena os ganhos g(v) de cada vértice da fronteira do vértice v em ordem decrescente. A diferença entre elas é que a Heurística 3, ao invés de tomar simplesmente o vértice com o maior ganho para compor o novo subconjunto p em formação como faz a Heurística 2, escolhe um vértice de forma aleatória a partir de um subconjunto restrito formado por alguns vértices (ou todos) que compõem a fronteira. Esses vértices são aqueles que implicarão num menor aumento no corte do grafo. Este subconjunto é chamado de lista de candidatos restrita (LCR), cujo tamanho é definido pelo parâmetro α, com α situado no intervalo [0, 1]. Esse parâmetro controla a qualidade dos vértices da LCR. Quando $\alpha = 0$, a escolha do vértice a ser adicionado ao subconjunto em formação é totalmente gulosa, fazendo com que o algoritmo comporte-se exatamente igual à Heurística 2. Por outro lado, $\alpha = 1$ implica em uma escolha totalmente aleatória, fazendo com que o algoritmo se comporte igual à Heurística 1.

4.4. Heurística 4

A Heurística 4 é uma aplicação da Heurística 3 de maneira recursiva. Essa quarta heurística forma uma k-partição do grafo através da aplicação de bisseções recursivas. O grafo inicialmente é bi-partido, depois o método de melhoramento é chamado para refinar a bisseção formada, e recursivamente essa estratégia é aplicada sobre os dois subconjuntos resultantes, e assim por diante. O algoritmo constrói a bisseção adicionando os vértices um por vez até que a metade dos vértices livres daquele subconjunto tenha sido inserida.

4.5. Método de melhoramento

O método de melhoramento é utilizado por todas as heurísticas propostas para refinar o corte parcial do grafo. A sub-rotina de melhoramento recebe como parâmetro duas listas, sendo uma delas o subconjunto construído e a outra a fronteira desse subconjunto. A sub-rotina tenta trocar vértices de uma lista para outra baseada no ganho g(i) de cada vértice i. Tal ganho representa o quanto o corte do grafo diminui se o vértice i for movido de um subconjunto para outro. Tal técnica de refinamento é semelhante ao algoritmo Fiduccia e Mattheyeses, tendo como principal diferença o fato de que enquanto o algoritmo FM avalia todos os vértices da bisseção, o método de melhoramento proposto avalia apenas os vértices do subconjunto formado e da sua fronteira.

5. Estratégia de paralelização proposta

Todas as heurísticas propostas por Bonatto (2010) foram implementadas em uma configuração *multistart*, ou seja, o algoritmo particionador constrói várias partições e utiliza apenas aquelas que resultam no menor corte. Sendo assim, quanto mais repetições do algoritmo, maior a probabilidade de se encontrar as partições que geram os menores cortes. Em uma implementação serial, um elevado número de repetições do algoritmo poderia gerar um custo elevado de recursos computacionais e tempo de execução.

Sendo assim, uma implementação paralela desse mesmo algoritmo para ser executado em um *cluster* de computadores, onde a quantidade total de iterações pudesse ser dividida entre os diversos nós processadores, seria uma solução para o problema do custo computacional e do tempo de execução.

As Heurísticas 1, 2 e 3 foram implementadas de forma paralela baseando-se no conceito da Heurística 4, que é o de criar bisseções recursivas do grafo até que *k* subconjuntos de vértices sejam formados.

Os algoritmos paralelos foram implementados utilizando a linguagem *Java* e uma biblioteca de envio de mensagens entre os nós do *cluster* que segue o padrão *MPI* chamada *MPJ Express*. Além disso, o algoritmo contempla o uso de *threads* para o caso de execução em *clusters* com nós processadores multiprocessados e/ou multinúcleos.

O algoritmo principal do particionador é composto por duas partes principais. A primeira delas é executada caso a quantidade de nós utilizados no *cluster* seja igual a um, configurando assim, uma execução serial. A segunda parte do algoritmo principal é executada caso a quantidade de nós que serão utilizados no cluster seja maior do que um, configurando assim, uma execução paralela.

O algoritmo paralelo é executado como a seguir: dois laços determinam a quantidade de vezes que o algoritmo será executado. Um laço mais exterior de *h* rodadas será executado $h_{max}=log_2 k$ vezes, onde *k* é o número final de partições do grafo. Para cada iteração desse laço externo, um laço interior executa 2^h vezes.

O laço mais interior inicializa a quantidade de *threads* determinada para a execução. Dentro de cada *thread* serão executadas as iterações de particionamento determinadas para a execução e as chamadas à rotina de melhoramento do corte, caso isso tenha sido configurado antes da execução do algoritmo. Após a execução das iterações de cada *thread*, cada uma delas terá obtido o menor corte daquela rodada. Esse esquema está representado na Fig. 1.

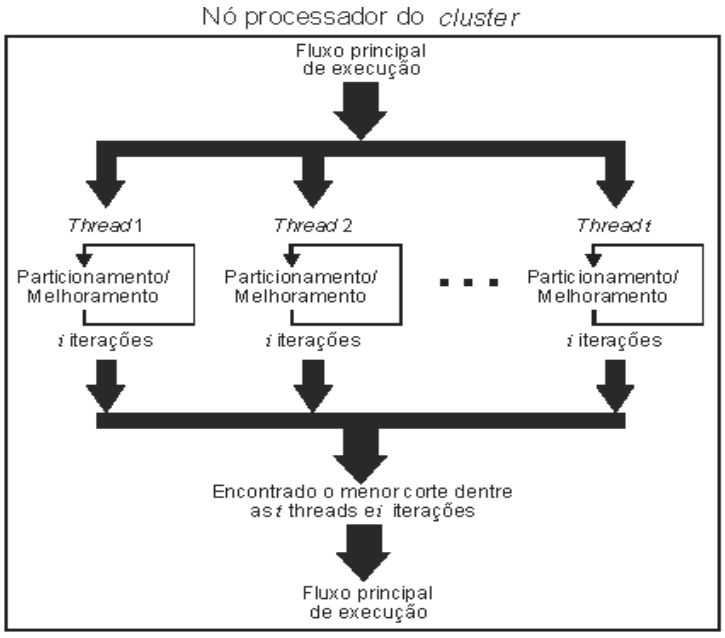

Figura 1: Threads e iterações sendo executadas em um nó processador.

O algoritmo obtém o menor corte dentre todas as *threads*, além de armazenar também as partições inicial e final que resultaram no menor corte e a fronteira com a nova partição criada, já que as mesmas servirão como parâmetros de entrada para a próxima rodada, caso o número de partições *k* do grafo seja maior do que 2.

A partição inicial possui todos os vértices daquela futura bisseção do grafo e a partição final está vazia antes da execução da iteração. Ao término da iteração, ambas as partições estão com a metade dos vértices da partição inicial, com uma diferença de no máximo um vértice.

Ao valor total do corte de arestas do grafo é adicionado o valor desse menor corte obtido após a execução das *threads* e suas iterações. Uma nova rodada *h* tem início, caso $k > 2$.

A ideia principal por trás do particionador paralelo é distribuir os pares de partições de vértices (inicial e final) que compõem um grafo naquele estágio do particionamento para cada um dos nós do *cluster* para eles encontrem um particionamento cujo corte é o menor possível dentro daquele número determinado de iterações para aquela rodada.

Na implementação paralela, o particionador paralelo também executa o laço exterior principal $h_{max}=log_2 k$ vezes, onde *k* é o número final de partições do grafo. Uma diferença importante com relação ao particionador serial é que o particionador paralelo distribui os pares de partições seguindo uma regra de distribuição particular a ser explicada a seguir, enquanto que no algoritmo particionador serial o único nó trabalha com todas as partições sem distinção.

Na primeira rodada de execução ($h = 0$), todos os p nós do *cluster* recebem as partições inicial e final iguais a 0 (contendo os vértices antes do particionamento) e 1 (vazia antes do particionamento), respectivamente. Todos os nós trabalham em paralelo tentando encontrar o menor corte e as partições correspondentes a esse menor corte. Todos os nós enviam ao *root* seus cortes encontrados e o nó de menor corte envia ao *root* as partições 0 e 1 que resultaram nesse menor corte.Na rodada seguinte de execução ($h = 1$), os nós com *rank* entre 0 e $p/2 - 1$ recebem as partições 0 (contendo os vértices obtidos no particionamento da rodada anterior) e 2 (vazia antes do particionamento), enquanto os nós com *rank* entre $p/2$ e $p - 1$ recebem as partições 1 (contendo os vértices obtidos no particionamento da rodada anterior) e 3 (vazia antes do particionamento). Agora, existem duas metades dos nós do cluster trabalhando em paralelo para calcularem o menor corte entre as partições 0 e 2 e o menor corte entre as partições 1 e 3. Todos os nós enviam ao *root* seus cortes encontrados e os nós de menores cortes entre as respectivas partições enviam ao *root* as partições 0 e 2 e as partições 1 e 3 que resultaram nesses menores cortes.

Nas rodadas seguintes de execução o processo se repete, ou seja, a cada rodada o número de nós que executam o particionamento de cada par de partições cai pela metade enquanto que o número de partições que é obtido nessa rodada dobra. Um exemplo do particionador serial sendo executado com 8 partições é mostrado na Fig. 2.

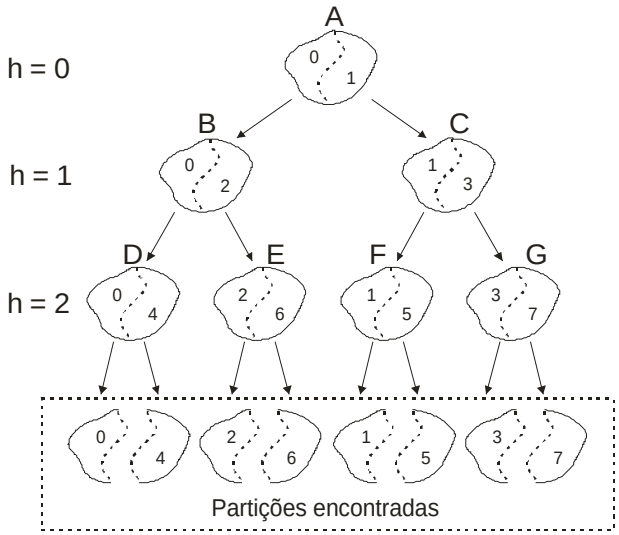

Figura 2: Exemplo de particionamento serial em 8 partições.

6. Resultados

Os algoritmos propostos em Bonatto (2010) apresentaram, em sua grande maioria, melhorias nos cortes dos grafos para uma bisseção se comparados aos algoritmos multiníveis Metis e Chaco. De acordo com o estudo proposto por Bonatto (2010) e Amaral *et al*., (2010), a Heurística 3 apresentou os melhores resultados em geral. Por serem algoritmos *multistart*, a versão serial foi executada 10 vezes, cada uma delas com 100 iterações. Dessas 10 execuções, foi tomado o menor corte.

Da mesma forma, a versão paralela da Heurística 3 foi executada 10 vezes, cada uma delas com 100 iterações, com a diferença de que a execução foi realizada em um *cluster* com 16 nós, cada um deles executando 16 *threads* simultaneamente, resultando em um total de 16 x 16 x 100 = 25.600 iterações. Dessas 10 execuções foram tomados os menores cortes, conforme Tabela 1, mostrada a seguir.

Tabela 1: Comparação dos resultados de corte para diferentes heurísticas em grafos exemplo**.**

Grafo	Metis	Chaco	H3 Serial	H3 Paralela
144	6871	6994	7248	7575
3elt	98	103	93	90
598a	2470	2484	2476	2463
add20	741	742	715	646
add32	19	11	11	11
airfoil1	85	82	77	74
bcsstk29	2923	3140	2885	2851
bcsstk33	12620	10172	10171	10171

A Heurística 3 paralela não apresentou melhoria no corte do grafo 144. Uma variação na configuração do algoritmo paralelo para utilizar a Heurística 2 (puramente gulosa) fez com esse grafo apresentasse um corte melhor. Com apenas 10 execuções de 10 iterações cada, rodando na mesma configuração de 16 nós e 16 *threads* por nó no *cluster*, o corte do referido (144) grafo foi melhorado, alcançando o valor de 6856.

Além das melhorias apresentadas nos cortes das bisseções, a paralelização do algoritmo trouxe ganhos significativos de *speedup*. Para um aumento do número de iterações, a execução puramente serial tornaria-se inviável na prática, porém, ao se paralelizar o algoritmo utilizando tanto o *cluster* quanto as *threads*, os tempos de execução diminuíram consideravelmente.

A Tabela 2 mostra diferentes configurações em termos de numero de nos, numero de threads e numero de iterações que foram consideradas para a avaliação de performance da implementação. Para cada configuração denominada pelo atributo "Nome", são executadas 6.400 iterações. Note que a configuração 01n01t simula a execução serial do algoritmo, sendo as demais sua paralelização em termos de número de nos de processamento e número de threads usados em cada no.

Tabela 2: Cenários de teste.

Nome	Nós	Threads	Iterações	Total
01n01t	1	1	6.400	6.400
01n16t	1	16	400	6.400
02n16t	2	16	200	6.400
04n16t	4	16	100	6.400
08n16t	8	16	50	6.400

A Tabela 3 mostra os tempos de execução, em segundos, para as diferentes configurações executadas para cada um dos grafos de estudo.

Tabela 3: Tempo de execução para os cenários da Tabela 2.

Grafo	01n01t	01n16t	02n16t	04n16t	08n16t
144	9.987,645	2.267,888	1.825,586	902,987	446,436
3elt	21,511	5,924	3,569	3,318	1,859
598a	4.884,235	1.222,244	868,995	428,393	218,654
add20	12,979	3,608	2,506	1,796	1,431
add32	12,979	3,608	2,506	1,796	1,431
airfoil1	19,142	4,206	3,226	2,367	2,419
bcsstk29	866,845	158,465	118,764	60,845	30,567
bcsstk33	1.367,146	254,094	187,402	96,084	49,175

A Fig. 3 mostra o tempo médio de execução de todos os grafos para cada uma das configurações. A curva mostra uma queda natural considerável a medida que o número de nós de processamento e o número de threads de execução aumenta.

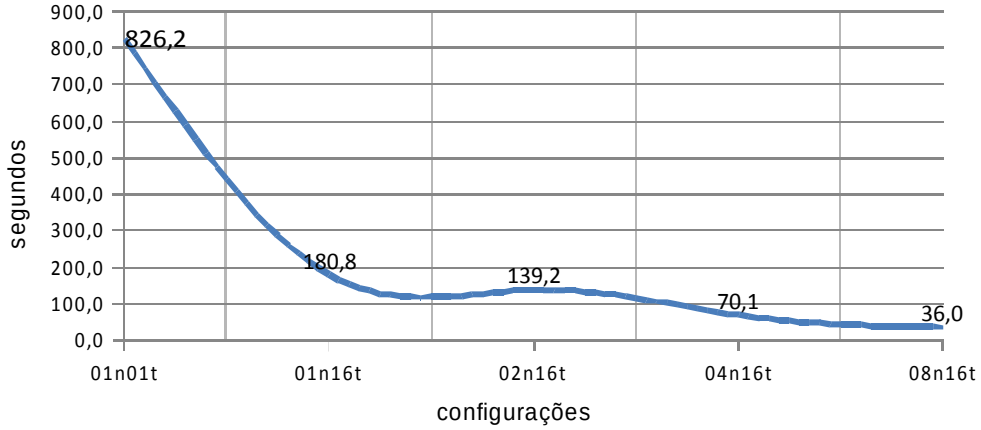

Figure 3: Tempo médio de execução para os diferentes cenários listados na Tabela 2.

Como resultado, a paralelização atinge uma boa taxa de aumento de velocidade, conforme pode ser mostrado na Fig. 4. Por exemplo, no cenário mais simples, rodando a heurística usando um nó e 16 threads (01n16t), a velocidade é multiplicada por 4. À medida que mais nós são adicionados, a velocidade de execução aumenta, até atingir o valor de 15,6 vezes a velocidade do algoritmo serial (08n16t).

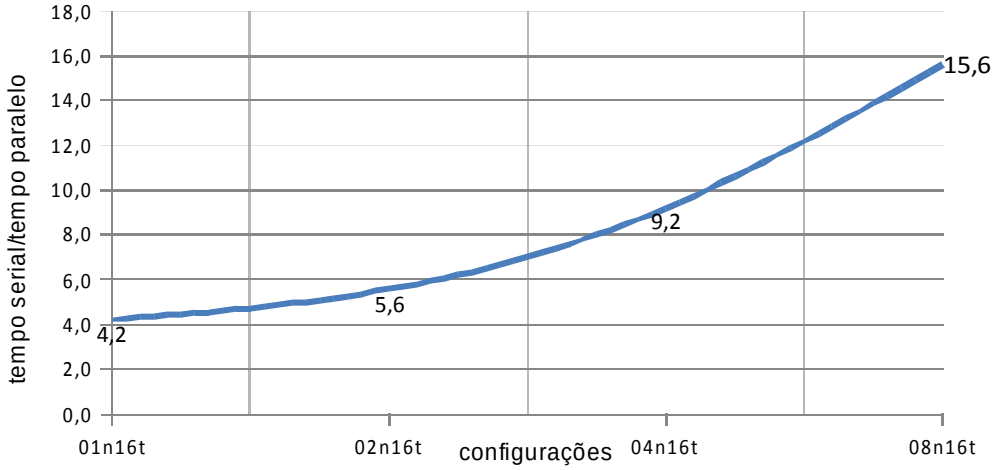

Figure 4: *Speedup* médio para os cenários listados na Tabela 2.

7. Conclusões

Soluções ótimas para particionamento de grafos com elevados números de vértices tornam-se inviáveis computacionalmente na prática. Diversas heurísticas e metaheurísticas têm sido propostas para contornar essa dificuldade.

Adicionalmente, técnicas de paralelização de algoritmos para execução em arquiteturas de maquinas paralelas podem ser usadas para diminuir o tempo computacional de execução. No presente trabalho, foi feita a paralelização de heurísticas de particionamento de grafos utilizando a plataforma de programação do MPIJava e uma plataforma computacional baseada em cluster de computadores, com nos de processamento idênticos baseados em processadores Intel QuadCore.

Após a paralelização das heurísticas propostas por Bonatto (2010), a execução da Heurística 3 de forma paralela apresentou melhora em 50 % dos cortes do grafos se comparada com os demais algoritmos seriais. Nos 50 % restantes, apesar de não ter havido melhora, os cortes foram iguais, nunca piores.

A execução da Heurística 3 apresentou os melhores resultados, exceto para o grafo 144, que dentre todos os grafos analisados, é o que apresenta o maior número de vértices. Nesse caso, a Heurística 2, puramente gulosa, apresentou melhora no corte.

Referências

SCHLOEGEL, K. , Karypis, G., Kumar, V., (2001). "Graph partitioning for high performance scientific simulations," CRPC Parallel Computing Handbook. Morgan Kaufmann.

OU, C., Ranka, S., (1997) "SPRINT: Scalable Partitioning, Refinement, and INcremental partitioning Techniques," unpublished.

FJALLSTROM, P.-O., (1998). "Algorithms for graph partitioning: a survey," in Linkoping Electronic Articles in Computer and Information Science, vol. 3, no. 10.

BONATTO, R.S., (2010). "Algoritmos heurísticos para partição de grafos com aplicação em processamento paralelo," Dissertação de mestrado, Universidade Federal do Espírito Santo, Vitória.

KARYPIS, G., Kumar, V., (1998). "A fast and high quality multilevel scheme for partitioning irregular graphs," in Siam J. Sci Comput., vol 20, no 1, pp 359-392.

KUMAR, V., Karypis, G., (1998). "Multilevel k-way partitioning scheme for irregular graphs," in Journal Of Parallel And Distributed Computing, no 48, pp 96-129.

GUATTERY, S., Miller, G.L., (1995). "On the performance of spectral graph partitioning methods," in Sixth Annual ACM/SIAM Symposium on Discrete Algorithms.

QIU, H., Hancock, E.R., (2006). "Graph matching and clustering using spectral partitions," in Pattern Recognition, no 39, pp 22-24.

FIDUCCIA, C., Mattheyeses, R., (1982). "A linear-time heuristic for improving network partitions," in 19th IEEE Design Automation Conference, pp 175-181.

KERNIGHAN, B.W., Lin, S., (1970). "An efficient heuristic procedure for partitioning graphs," in The Bell System Technical Journal, pp 291-307.

JOHNSON, D.S., Aragon, C.R., (1989), et al. "Optimizationby simulated annealing: an experimental evaluation; part I, graph partitioning," Oper. Res., no 37, pp 865-892.

ROLLAND, E., Pirkul, H., Glover, F., (1996). "Tabu search for graph partiotining," in Ann. Oper. Res., no 63, pp 209-232.

AMARAL, A.R.S., Bonatto, R.S., (2010). "Algoritmo heurístico para partição de grafos com aplicação em processamento paralelo," apresentado no XLII Congresso da Sociedade Brasileira de Pesquisa Operacional, Bento Gonçalves, Brazil.