UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENERGIA

LEONARDO ROGÉRIO BINDA DA SILVA

PARTICIONADOR PARALELO DE GRAFOS UTILIZANDO ALGORITMOS HEURÍSTICOS PARA APLICAÇÃO EM SIMULADORES PARALELOS DE RESERVATÓRIOS DE PETRÓLEO

SÃO MATEUS

2014

LEONARDO ROGÉRIO BINDA DA SILVA

PARTICIONADOR PARALELO DE GRAFOS UTILIZANDO ALGORITMOS HEURÍSTICOS PARA APLICAÇÃO EM SIMULADORES PARALELOS DE RESERVATÓRIOS DE PETRÓLEO

Dissertação apresentada ao Programa de Pós-Graduação em Energia do Centro Universitário Norte do Espírito Santo da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do título de Mestre em Energia.
Orientador: Prof. Dr. Roney Pignaton da Silva.

SÃO MATEUS

2014

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Silva, Leonardo Rogério Binda da, 1974-
S586p Particionador paralelo de grafos utilizando algoritmos heurísticos para aplicação em simuladores paralelos de reservatórios de petróleo / Leonardo Rogério Binda da Silva. – 2014.
171 f. : il.

Orientador: Roney Pignaton da Silva.
Dissertação (Mestrado em Energia) – Universidade Federal do Espírito Santo, Centro Universitário Norte do Espírito Santo.

1. Engenharia do petróleo. 2. Heurística. 3. Programação paralela. 4. Representações dos grafos. I. Silva, Roney Pignaton da . II. Universidade Federal do Espírito Santo. Centro Universitário Norte do Espírito Santo. III. Título.

CDU: 620.9

LEONARDO ROGÉRIO BINDA DA SILVA

PARTICIONADOR PARALELO DE GRAFOS UTILIZANDO ALGORITMOS HEURÍSTICOS PARA APLICAÇÃO EM SIMULADORES PARALELOS DE RESERVATÓRIOS DE PETRÓLEO

Dissertação apresentada à Universidade Federal do Espírito Santo como parte das exigências do Programa de Pós-Graduação em Energia, para obtenção do título de Mestre em Energia.

Aprovada em 14 de Março de 2014.

COMISSÃO EXAMINADORA

_____

Prof. Dr. Roney Pignaton da Silva
Universidade Federal do Espírito Santo
Orientador

_____

Prof. Dr. Wanderley Cardoso Celeste
Universidade Federal do Espírito Santo

_____

Prof. Dr. Luciano Lessa Lorenzoni
Instituto Federal do Espírito Santo

Dedico esse trabalho à memória de Renato Stocco Bonnato, que apesar de não estar mais presente fisicamente, sua lembrança e os frutos de suas pesquisas acadêmicas permanecem entre nós.

AGRADECIMENTOS

Primeiramente a Deus, pelo dom da vida e pelas oportunidades que Ele tem me dado até hoje.

Aos meus queridos pais Angelo e Regina, irmão Giuliano e irmãs Paula e Angela, sem os quais não teria chegado a esse momento da minha vida.

À minha amada esposa Valéria, pelo apoio e compreensão das muitas horas que despendi para a realização desse trabalho e que por isso não pude estar ao seu lado.

Aos meus sogros Darcy e Sebastiana pelos inúmeros finais de semana que me hospedei na casa deles para estudar no dia seguinte.

À minha prima Maria das Graças por ter me apresentado o edital do Programa de Mestrado em Energia.

Ao meu orientador Roney Pignaton por todo o seu conhecimento transmitido, paciência, disponibilidade em me atender até nos fins de semana em sua casa e por todo companheirismo dispensado a mim.

A todos os professores e colegas do programa de mestrado em Energia, em especial ao coordenador, professor Fábio Ressel.

Aos senhores Pergentino Jr., Fabiano Chiepe e Neacil Broseghini, pelas oportunidades que me foram concedidas para que essa etapa pudesse ser conquistada.

À família Bonatto, nas pessoas de Antônio, Elza e Gustavo, por terem me cedido o material deixado pelo Renato para que eu iniciasse meus estudos.

“Talvez não tenhamos conseguido fazer o melhor, mas lutamos para que o melhor fosse feito. Não somos o que deveríamos ser, não somos o que desejamos ser, não somos o que iríamos ser, mas graças a Deus, não somos o que éramos.”

Martin Luther King

RESUMO

O petróleo é atualmente o combustível mais utilizado no mundo. Recuperá-lo com a maior viabilidade econômica possível é uma busca incessante das companhias produtoras. Nesse cenário, a simulação numérica de reservatórios utilizando computadores paralelos de memória distribuída (*clusters*) desponta como uma importante ferramenta. Esses aplicativos manipulam malhas de pontos discretizados que representam o domínio do reservatório de petróleo. Uma etapa importante da simulação utilizando *clusters* é o particionamento dessa malha para que cada um dos nós processadores possa executar seus cálculos sobre uma porção da mesma. As malhas de domínio podem ser representadas por grafos. Particionar malhas, então, torna-se um problema de particionamento de grafos. Caso o número de vértices do grafo que representa a malha seja muito elevado, particionadores seriais podem apresentar problemas de desempenho. Particionadores de grafos utilizando *clusters* surgem como alternativas interessantes nessa situação, minimizando os tempos gastos nos particionamentos. Trata da implementação de um particionador paralelo de grafos para ser utilizado em *clusters* baseado nas Heurísticas de particionamento propostas e implementadas de maneira serial por Bonatto (2010). O particionador paralelo foi desenvolvido utilizando a linguagem de programação *Java* e a biblioteca de passagem de mensagens *MPJ Express*. Tipos abstratos de dados eficientes foram propostos e implementados para que o desempenho fosse otimizado. O particionador de grafos paralelo realizou o corte de diversos grafos, obtendo em sua grande maioria cortes menores do que os encontrados pelo particionador serial de Bonatto (2010) e por programas como o *METIS* e o *CHACO*. Melhorias ao particionador serial de Bonatto (2010) foram propostas. Análises de *speedup* e eficiência paralela foram realizadas para constatar os ganhos de tempos obtidos com a paralelização das heurísticas.

Palavras-chave: Engenharia de Petróleo. Particionamento de Grafos. Heurísticas. Computação Paralela. Simulação de Reservatórios. *Clusters* de computadores.

ABSTRACT

Oil is currently the most widely fuel used in the world. To obtain it to the greatest possible economic viability is a relentless pursuit of the producing companies. In this scenario, the numerical reservoir simulation using parallel computers with distributed memory (clusters) is emerging as an important tool. These applications handle meshes of discrete points that represent the field of oil reservoir. An important step of the simulation using clusters is the partitioning of this mesh points so that each cluster processor node can perform its calculations on a portion of this mesh. The domain meshes can be represented by graphs. Partitioning meshes then becomes a problem of graph partitioning. If the graph vertices number that represents the mesh is very high, serial partitioners can have performance problems. Graph partitioners using clusters appear as interesting alternatives in this situation, minimizing the time spent in partitioning. This research deals with the implementation of a parallel graph partitioner to be used in clusters based on partitioning heuristics proposed and implemented serially by Bonatto (2010). The parallel partitioner has been developed using the Java programming language and MPJ Express messages passing library. Efficient abstract data types have been proposed and implemented in order to optimize the performance. The parallel graph partitioner performed the cutting of different graphs, obtaining, most of the time, smaller cuts than the ones found by serial partitioner of Bonatto (2010) and by programs such as METIS and CHACO. Improvements to the Bonatto (2010) serial partitioner have been proposed. Analysis of speedup and parallel efficiency have been performed to find out the gains of times obtained with the parallelization of the heuristics.

Keywords: Petroleum Engineering. Graphs Partitioning. Heuristics. Parallel Computing. Reservoir Simulation. Clusters of Computers.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE SIGLAS E ACRÔNIMOS

API	*Application Programming Interface* (Interface de Programação de Aplicativos)
ASP	*Alkaline-Surfactant-Polymer* (Álcali-Surfactante-Polímero)
CD	*Compact Disk* (Disco Compacto)
CPU	*Central Processing Unit* (Unidade de Processamento Central)
CSR	*Compressed Sparse Row* (Matriz de Linhas Esparsas Comprimida)
CSRG	*Compressed Sparse Row Graph* (Grafo de Matriz de Linhas Esparsas Comprimida)
DRDB	*Distributed Replicated Block Device* (Dispositivo de Bloco Replicado Distribuído)
DVD	*Digital Versatile Disc* (Disco Digital Versátil)
E	Equiprovável
EDPs	Equações Diferenciais Parciais
EG	Equiprovável Gulosa
EP	Entidade Processadora
FCS	*Fiber Channel Standard (Padrão de Canal de Fibra)*
FM	Fidducia e Mattheyeses
FPG	Fixo Próximo da escolha Gulosa
GHz	Giga Hertz
GLP	Gás Liquefeito de Petróleo
GNU	*GNU is Not Unix* (GNU Não é Unix)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
GT/s	*Giga Transfers per Second* (Bilhões de Transferências por Segundo)
GUI	*Graphical User Interface* (Interface Gráfica do Usuário)
H	Híbrido
HA	*High Availability* (Alta Disponibilidade)
HPC	*High Performance Computing* (Computação de Alto Desempenho)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
JIT	*Just-In-Time* (Sob demanda)
KL	Kernighan-Lin
LCR	Lista de Candidatos Restrita
LGR	*Local Grid Refinement* (Refinamento da Malha Local)
LVS	*Linux Virtual Server* (Servidor Virtual Linux)
MB	*Mega Bytes*
MB/s	*Mega Bytes per Second* (Mega Bytes por Segundo)
MEOR	*Microbial Enhanced Oil Recovery* (Recuperação de Óleo Avançada

	Microbial)
MIMD	*Multiple Instruction, Multiple Data* (Múltiplas Instruções, Múltiplos Fluxos de Dados)
MISD	*Multiple Instruction, Single Data* (Múltiplas Instruções, Único Fluxo de Dados)
MPI	*Message Passing Interface* (Interface de Passagem de Mensagens)
MPJ	*Message Passing For Java* (Passagem de Mensagens para *Java*)
NP	*Non-deterministic Polynomial-time* (Não-determinístico em Tempo Polinomial)
Op/s	Operações por Segundo
PC	*Personal Computer* (Computador Pessoal)
PoPC	*Pile-of-PCs* (Pilha de Computadores Pessoais)
PPG	Problema de Particionamento de Grafos
PPG-*k*	Problema de Particionamento de Grafos em k partições
PVM	*Parallel Virtual Machine* (Máquina Paralela Virtual)
RAID	*Redundant Array of Independent Disks* (Conjunto Redundante de Discos Independentes)
RAM	*Random Access Memory* (Memória de Acesso Aleatório)
RCB	*Recursive Coordinate Bisection* (Bisseção Coordenada Recursiva)
RIB	*Recursive Inertial Bisection* (Bisseção Inercial Recursiva)
RPM	Rotações Por Minuto
RSB	*Recursive Spectral Bisection* (Bisseção Espectral Recursiva)
SA	*Simulated Annealing* (Recozimento Simulado)
SAGD	*Steam Assisted Gravity Drainage* (Drenagem Gravitacional Assistida por Vapor)
SATA	*Serial Advanced Technology Attachment* (Conector de Tecnologia Avançada Serial)
SCI	*Scalable Coherent Interface* (Interface Coerente Escalável)
SIMD	*Single Instruction, Multiple Data* (Única Instrução, Múltiplos Fluxos de Dados)
SISD	*Single Instruction, Single Data* (Única Instrução, Único Fluxo de Dados)
SMP	*Symmetric Multi-Processing* (Multiprocessamento Simétrico)
TAD	Tipo Abstrato de Dados
TB	*Tera Bytes*
TCP/IP	*Transmission Control Protocol/Internet Protocol* (Protocolo de Controle de Transmissão/Protocolo de Interconexão)
VLIW	*Very Long Instruction Word* (Palavra de Instrução Muito Longa)
VLSI	*Very Large Scale Integration* (Integração em Muito Larga Escala)

LISTA DE SÍMBOLOS

Φ	Porosidade da rocha reservatório.
k	Permeabilidade do meio poroso.
V_p	Volume dos poros de um reservatório.
V_t	Volume total do reservatório.
h	Altura de um reservatório de petróleo.
u	Razão de fluxo ou velocidade de um fluido através de um meio poroso.
q	Vazão de fluido.
A	Área da seção transversal de um meio poroso.
Δp	Diferencial de pressão.
μ	Viscosidade do fluido.
L	Comprimento do meio poroso.
d	Distância entre cada uma das linhas de poços numa malha de injeção.
a	Distância entre cada um dos poços numa malha de injeção.
T_P	Tempo de Execução do algoritmo paralelo em P nós processadores.
T_S	Tempo de Execução do algoritmo serial.
f	Fração não paralelizável de um algoritmo.
E	Eficiência Paralela.
$P, p, size$	Número de nós processadores do *cluster*.
V, N	Conjunto de vértices de um grafo.
E	Conjunto de arestas de um grafo.
$\|G\|, \|V\|$	Ordem de um grafo, representada pelo número de vértices do mesmo.
$\|\|G\|\|, \|E\|$	Número de arestas de um grafo.
k	Número de subconjuntos nos quais um grafo é particionado.
A	Matriz de adjacência correspondente a um grafo.
D	Matriz que contém os graus dos vértices de um grafo.
g	Ganho obtido no *cut size* ao se trocar dois vértices em diferentes subconjuntos do grafo.
p	Subconjunto de vértices sendo construído a partir do subconjunto original durante o particionamento do grafo.
v	Vértice do grafo.
α	Parâmetro que controla a qualidade dos vértices numa LCR.
h	Altura da árvore de bissecionamento de um grafo.
$rank$	Número de identificação de cada nó processador do *cluster*.
t	Número de *threads* que cada nó processador executa simultaneamente.
$M(G)$	Matriz de incidência de um grafo.
$A(G)$	Matriz de adjacência de um grafo.

AA	Vetor de elementos não nulos da matriz de adjacência do grafo.
Nz	Quantidade de elementos não nulos de uma matriz de adjacência que representa um grafo.
JA	Vetor que contém os índices das colunas dos elementos não nulos da matriz de adjacência que representa um grafo.
IA	Vetor que contém ponteiros para os inícios de cada linha da matriz no vetor que representa a matriz de adjacência do grafo.
d	Grau de um vértice/grafo.

SUMÁRIO

ÍNDICE

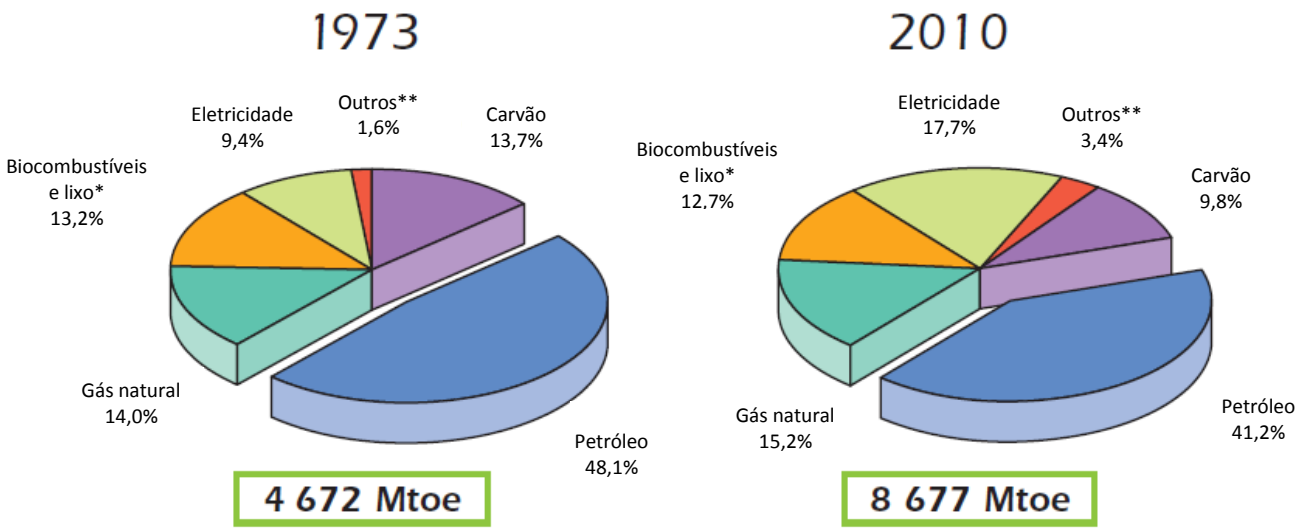
1 INTRODUÇÃO

Nas seções desse capítulo são apresentados a motivação e as considerações gerais para a realização deste trabalho, os objetivos gerais e os objetivos específicos do mesmo. Uma breve explanação é feita para que o problema do particionamento de grafos e sua paralelização sejam inseridos no contexto da Engenharia de Petróleo.

1.1 MOTIVAÇÃO E CONSIDERAÇÕES GERAIS

Atualmente, o petróleo é a fonte de energia mais utilizada em todo o mundo. Apesar de ser não renovável e apresentar desvantagens tais como a emissão de gases e a contribuição para o aumento do efeito estufa, e, além disso, diversos outros combustíveis renováveis terem despontado nos últimos anos como alternativas ao mesmo, o principal combustível que move o mundo moderno ainda é o petróleo.

Uma comparação entre o uso dos diversos combustíveis em todo o mundo em 1973 e 2010 é apresentada na Figura 1. No ano de 1973, o petróleo era responsável por 48,1% do consumo de combustíveis no mundo. Apesar de no ano de 2010 tal parcela ter diminuído para 41,2%, o petróleo ainda detém a maior fatia de participação no consumo mundial de combustíveis.



*Dados antes de 1994 para consumo final de biocombustíveis foram estimados.
**Outros incluindo geotérmica, solar, eólica, calor, etc.

Figura 1 – Utilização dos combustíveis no mundo em 1973 e em 2010
Fonte: International Energy Agency (2012, p. 28, tradução nossa).

Dada tal importância do petróleo, a descoberta de novos reservatórios e a maneira mais econômica de se extrair o petróleo deles têm sido cada vez mais objeto de estudo pelas companhias produtoras. Técnicas avançadas de exploração e explotação dos reservatórios têm sido desenvolvidas ao longo dos últimos anos, de forma a assegurar altas taxas de produtividade com menores custos de produção.

Para se decidir pela viabilidade ou não de um reservatório de petróleo, muitos parâmetros e fatores devem ser avaliados antes de iniciar a explotação de um campo petrolífero (CORDAZZO, 2006). Todo um estudo preliminar deve ser feito antes de a produção ter início. Esse estudo é denominado Estudo de Gerenciamento de Reservatórios.

Segundo Fanchi (2006, p. 2, tradução nossa), “[...] o principal objetivo de um Estudo de Gerenciamento de Reservatórios é determinar as condições ideais necessárias para maximizar a recuperação econômica de hidrocarbonetos a partir de um campo operado com prudência”.

Dentre as diversas técnicas de explotação, uma de grande importância para prever o comportamento do reservatório ao longo da sua vida útil de produção é a simulação numérica de reservatórios. Tal ferramenta é a mais sofisticada para se atingir o objetivo principal do Estudo de Gerenciamento do Reservatório.

O Quadro 1 apresenta diversas razões para o uso de simulações numéricas no Estudo de Gerenciamento do Reservatório.

Impacto corporativo • Previsão de Fluxo de Caixa   o Necessidade de previsão econômica do preço dos hidrocarbonetos

Gerenciamento do Reservatório
• Coordenar atividades de gerenciamento do reservatório
• Avaliar desempenho de projetos
  o Interpretar/Entender o comportamento do reservatório
• Sensibilidade do modelo para dados estimados
  o Determinar a necessidade de dados adicionais
• Estimar o projeto de vida
• Prever recuperação x tempo
• Comparar diversos processos de recuperação
• Planejar mudanças operacionais ou de desenvolvimento
• Selecionar e otimizar planejamentos
• Maximizar a recuperação econômica

Quadro 1 – Por que simular?
Fonte: Fanchi (2006, p. 2, tradução nossa)

Simuladores numéricos de reservatórios são utilizados para prever o comportamento do reservatório de petróleo antes e durante as recuperações primária e secundária de óleo.

A recuperação primária é a quantidade de óleo que é retirada de um reservatório por meio das energias naturais do próprio reservatório. Já a recuperação secundária é a quantidade adicional de óleo que é obtida pela suplementação da energia primária, com energia secundária transferida artificialmente para a jazida (ROSA; CARVALHO; XAVIER, 2006, p. 561).

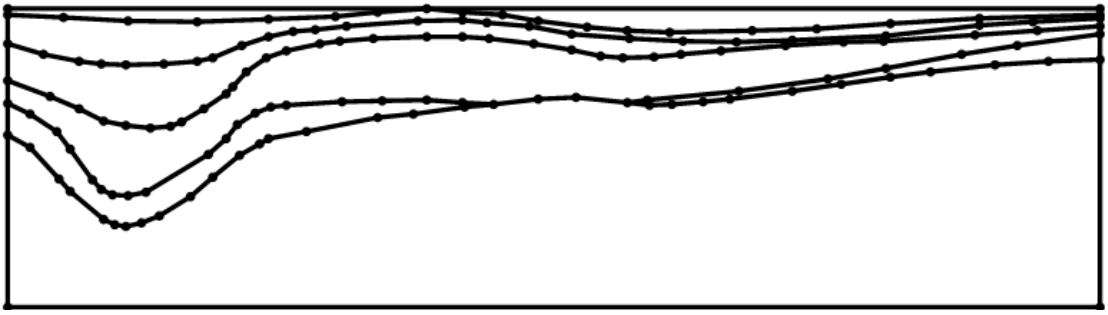
Em ambos os casos, os modelos matemáticos que regem o comportamento da recuperação geralmente são construídos utilizando-se equações diferenciais, que são demasiadamente complexas para serem resolvidos analiticamente. Por isso, utilizam-se métodos numéricos que aproximam um sistema de EDPs, por exemplo, em um sistema linear de equações algébricas (CORDAZZO, 2006, p. 3).

Nesse caso da aproximação por equações lineares, a solução passa a ser obtida para um número discreto de pontos definidos por uma malha, onde assume-se um determinado erro que diminui à medida que a quantidade de pontos amostrados aumenta (CORDAZZO, 2006, p. 3).

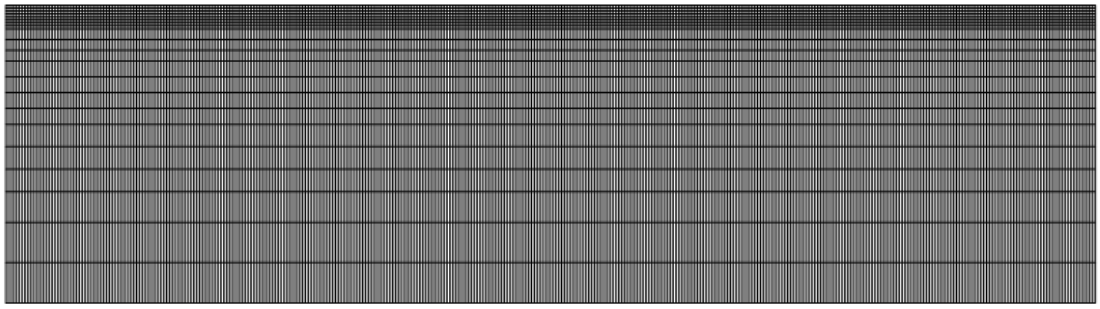
A principal ideia por detrás de qualquer método numérico aproximado é substituir o problema analítico original por outro que seja mais fácil de ser resolvido e cuja solução é próxima da solução do problema original (AZIZ; SETTARI, 1979, *apud* SILVA, 2008, p. 16).

Com a discretização do domínio do reservatório por meio de uma malha, é desejável que se tenha o maior número de pontos possíveis para que o erro durante a execução da simulação seja o menor possível.

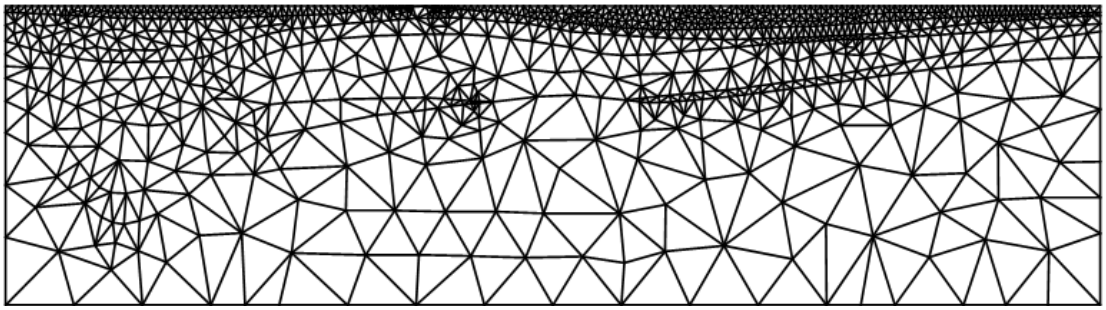
Um exemplo de discretização de um domínio utilizando malhas estruturadas e não-estruturadas é apresentado na Figura 2.



(a) Domínio real



(b) Discretização do domínio utilizando malha estruturada



(c) Discretização do domínio utilizando malha não-estruturada

Figura 2 – Representações discretizadas de um domínio
Fonte: Shewchuk (1999, p. 7).

O aumento do número de pontos para a representação do domínio do reservatório por meio da malha acarreta o aumento da complexidade dos cálculos a serem realizados pelo simulador, demandando um poder de processamento computacional consequentemente maior.

A computação paralela de memória distribuída (*clusters* de computadores) tornou-se uma alternativa interessante para se realizar simulações de grande porte em reservatórios de petróleo (SILVA, 2008, p. 17).

A computação paralela vem se tornando atualmente a plataforma de suporte para a execução de aplicações complexas. O ganho de desempenho proporcionado pela execução concorrente pode ser conseguido pelo uso de arquiteturas como *SMP*, *Cluster Computing* ou mesmo por arquiteturas híbridas, ou seja, uma combinação das duas primeiras.

Estas arquiteturas introduzem ainda a característica de alta performance a baixo custo. Mas para se obter o máximo desempenho dessas configurações, torna-se imprescindível a implementação de aplicações usando linguagens de programação paralela.

No contexto das disciplinas de Eficiência Energética e Produção de Petróleo, Gás e Energias Renováveis, uma série de fenômenos físicos são modelados analiticamente por EDPs. Por exemplo, a dispersão de calor em uma chapa de aço (KISCHINHEVSKY *et al*., 2005), a propagação de ondas eletromagnéticas (equações de Maxwell na forma diferencial) (PEREIRA, 2011), o escoamento de gás e líquido ao longo de uma rede de dutos (SOUZA, 2010) e, especialmente de interesse deste trabalho, o escoamento em reservatórios de petróleo (SILVA, 2008), entre outros.

Durante a execução de uma simulação em um computador paralelo, cada nó processador do *cluster* fica encarregado de realizar cálculos sobre uma porção de pontos da malha, sendo o resultado final combinado a partir dos resultados parciais obtidos em cada nó processador.

Para a comunicação entre os nós do *cluster* é utilizada uma rede local de computadores. Informações referentes aos pontos fronteiriços da malha devem ser trocadas entre os nós processadores vizinhos utilizando essa rede de comunicação, que por razões físicas, é muito mais lenta que os processadores e a memória local dos nós.

Portanto, ao se particionar a malha do domínio em subdomínios que serão enviadas para cada nó processador, é fundamental para o bom desempenho do simulador que essas partes da malha tenham o menor número possível de pontos fronteiriços

comunicando-se com os pontos fronteiriços da malha que estão em outro nó processador do *cluster*, por meio da divisão da carga de trabalho via partição dos dados que deverão ser distribuídos e processados pelos respectivos nós de processamento do computador paralelo.

Uma das maneiras de se particionar essa malha gerando essa menor comunicação durante o processamento do simulador é utilizando um grafo para representá-la.

> O problema de particionamento de domínio pode ser modelado como um problema de particionamento de grafos. Neste tipo de aplicação, os vértices do grafo representam as células do domínio e as arestas representam os dados trocados entre os subdomínios. A comunicação total entre os subdomínios é representada, então, pelo total de arestas entre os subdomínios (DORNELES, 2003, p. 19).

Uma vez que um grafo representa uma malha, particionar tal grafo de forma que a quantidade de arestas que conectam as partições em diferentes nós seja mínima (corte mínimo) fará com que a comunicação entre nós processadores também seja minimizada durante a execução do simulador em um computador paralelo de memória distribuída.

> [...] Os vértices do grafo representam unidades de cálculo e as arestas descrevem a dependência de dados (necessidade de trocas de informações). Assim, o objetivo é dividir o conjunto de vértices do grafo em k subconjuntos de aproximadamente mesma cardinalidade (ou peso), minimizando o corte de arestas. Neste caso, k é o número de processadores paralelos disponíveis. Desse modo, o particionamento do grafo provoca uma distribuição balanceada da carga de trabalho entre os k processadores paralelos e minimiza a comunicação entre eles (BONATTO, 2010, p. 15).

Os problemas de particionamento de grafos são *NP*-difíceis e para grafos com um número elevado de vértices, algoritmos particionadores seriais podem ter seu desempenho comprometido. A computação paralela torna-se uma excelente ferramenta para a resolução de tais problemas de particionamento.

O processo de particionamento de grafos utilizando computadores paralelos envolve uma série de etapas, dentre elas a definição da estratégia de particionamento para que a mesma seja eficiente e a definição de métricas capazes de mensurar tal eficiência.

O foco deste trabalho é a proposta de um algoritmo particionador de grafos para ser utilizado em um computador paralelo de memória distribuída com o objetivo de se obter o menor corte possível entre as arestas de vértices de diferentes partições.

Tais grafos são representações de malhas de domínios discretizados utilizadas em simuladores paralelos de reservatórios de petróleo.

Na próxima seção, tais objetivos são apresentados e detalhados, assim como os objetivos e as contribuições desse trabalho.

1.2 OBJETIVOS E CONTRIBUIÇÕES

Diversos pacotes de aplicativos para particionamento de grafos estão disponíveis atualmente, dentre os quais se pode citar o *METIS*, *JOSTLE*, *SCOTCH* e *CHACO* (GUEDES, 2009, p. 61).

Bonatto (2010) propôs quatro heurísticas combinatórias para resolver o PPG e obteve em seus testes computacionais partições de grafos, em sua maioria, com cortes menores do que os obtidos pelos *softwares METIS* e *CHACO*.

Apesar de ter obtido partições de grafos com cortes com qualidade superior ao *METIS* e *CHACO*, as heurísticas combinatórias propostas por Bonatto (2010) despenderam um tempo de execução um maior do que os tempos de execução dos demais *softwares* não combinatórios.

As heurísticas combinatórias propostas são *multistart*, o que significa que os algoritmos executam várias vezes o particionamento e retornam as partições do grafo cujo corte foi o menor.

As implementações de Bonatto (2010) são seriais, ou seja, projetadas para serem executadas em apenas um processador *monothreaded*.

Para se obter o melhor corte possível utilizando-se heurísticas *multistart*, um número elevado de partições iniciais deve ser obtido, porém, em computadores seriais isso tende a despender um tempo elevado de processamento.

A paralelização desse tipo de cenário é ideal, quando um elevado número de execuções de um processador serial pode ser substituído por uma quantidade menor de execuções em cada nó processador utilizando-se um *cluster*, e por consequência, diminuindo o tempo total de execução do particionamento.

Dessa forma, o objetivo geral deste trabalho é:

- Desenvolvimento de uma solução paralela eficiente para o particionamento de grafos, usando como base as heurísticas definidas em Bonato (2010).

Os objetivos específicos deste trabalho são:

- Realização de uma revisão bibliográfica acerca dos assuntos relacionados ao desenvolvimento do trabalho;
- Implementação paralela das heurísticas combinatórias propostas por Bonatto (2010);
- Definição de estratégias capazes de melhorar a eficiência das soluções obtidas;
- Avaliação do desempenho das soluções paralelas desde o ponto de vista de qualidade do corte, bem como avaliar a eficiência com relação à solução serial usando como base um conjunto de métricas de desempenho (*speedup* e eficiência paralela).

Dentre as principais contribuições deste trabalho estão:

- Definição de uma estratégia de paralelização para o particionamento de grafos;
- Implementação das heurísticas propostas por Bonatto (2010) para serem executadas em um computador paralelo de memória distribuída. Essas

implementações serão modificadas com a introdução de características de maneira a diversificar as soluções *multistart*, possibilitando uma maior variedade de soluções de particionamento de grafos e, consequentemente, melhoria da qualidade do corte.

1.3 ORGANIZAÇÃO GERAL DO TRABALHO

No capítulo 2 do presente trabalho, encontra-se a revisão bibliográfica necessária para o embasamento teórico dessa pesquisa. Tal referência serviu para um melhor entendimento dos conceitos aplicados na implementação prática do trabalho.

No capítulo 3, está descrita a metodologia utilizada para a implementação dos algoritmos paralelos e seu funcionamento, além das melhorias propostas por este trabalho.

No capítulo 4, são apresentados e discutidos os testes computacionais realizados a partir do algoritmo paralelo proposto para o particionamento de grafos.

No capítulo 5, são apresentadas as conclusões deste trabalho.

Por fim, no capítulo 6, são apresentadas as sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta a revisão bibliográfica pesquisada para o embasamento teórico do trabalho.

Explica de forma sucinta os elementos básicos envolvidos na produção de petróleo, tais como o petróleo em si, a rocha reservatório, a recuperação primária e secundária e a importância da simulação numérica de reservatórios para o contexto desta pesquisa.

Tais conceitos foram fundamentais para o entendimento dos reservatórios de petróleo e consequentemente, do funcionamento dos simuladores de reservatórios de petróleo.

Além disso, discorre sobre os principais tópicos de computação paralela, tais como implementações de *clusters* de computadores e métricas de desempenho computacional paralelo. Trata também das linguagens e tecnologias utilizadas para a implementação do particionador versado por este trabalho.

Apresenta também os conceitos relacionados a grafos, tais como suas definições, definição do Problema de Particionamento de Grafos, além de uma comparação entre os principais métodos utilizados para resolver tal problema.

2.1 PETRÓLEO

Uma definição bem sucinta de petróleo é dada por Rosa, Carvalho e Xavier (2006, p. 1): “Petróleo (do latim *petra* = rocha e *oleum* = óleo) é o nome dado às misturas naturais de hidrocarbonetos que podem ser encontradas no estado sólido, líquido ou gasoso, a depender das condições de pressão e temperatura a que estejam submetidas”

O petróleo já era utilizado pelas antigas civilizações:

> O registro da participação do petróleo na vida do homem remonta a tempos bíblicos. Na antiga Babilônia, os tijolos eram assentados com asfalto e o betume era largamente utilizado pelos fenícios na calafetação de embarcações. Os egípcios o usaram na pavimentação de estradas, para embalsamar os mortos e na construção das pirâmides, enquanto gregos e romanos dele lançaram mão para fins bélicos. No Novo Mundo, o petróleo era conhecido pelos índios pré-colombianos, que o utilizavam para decorar e impermeabilizar seus potes de cerâmica. Os incas, os maias e outras civilizações antigas também estavam familiarizados com o petróleo, dele se aproveitando para diversos fins (THOMAS *et al*., 2001, p. 1).

Apesar de ser conhecido e utilizado há tanto tempo, a era do petróleo começou na segunda metade do século XIX. Segundo Thomas *et al*. (2001, p. 1), dois fatos marcaram o início da era do petróleo:

- A perfuração de um poço com 21 metros de profundidade por meio de um sistema de percussão movido a vapor que produziu 2 m³/dia de óleo por Cel. Drake em 1859, em Tittusville, Pensilvânia;
- A descoberta que a destilação do petróleo resultava em produtos que substituíam o querosene e o óleo de baleia com grande margem de lucro.

Desde então, o petróleo firmou-se como uma importante fonte de energia até os dias atuais. Além de ser utilizado como fonte de energia, diversos outros produtos são obtidos a partir do petróleo e seus derivados.

De acordo com Thomas *et al.* (2001, p. 2),

> [...] o petróleo foi se impondo como fonte de energia. Hoje, com o advento da petroquímica, além da utilização dos seus derivados, centenas de novos compostos são produzidos, muitos deles diariamente utilizados, como plásticos, borrachas sintéticas, tintas, corantes, adesivos, solventes, detergentes, explosivos, produtos farmacêuticos, cosméticos, etc.

O petróleo é formado por diversos componentes. Segundo Petroleum (2013), apesar da composição do petróleo conter muitos traços de vários elementos, os componentes chave são: carbono (93% - 97%), hidrogênio (10% - 14%), nitrogênio (0,1% - 2%), oxigênio (0.1% - 1.5%) e enxofre (0.5% - 6%), com outros poucos metais perfazendo uma pequena porcentagem da composição do petróleo.

2.1.1 Engenharia de Reservatórios de Petróleo

Segundo Rosa, Carvalho e Xavier (2006, p. xi):

> A engenharia de reservatórios constitui uma subárea de extrema importância na engenharia de petróleo. Os engenheiros, geólogos e geofísicos de petróleo, assim como outros profissionais que atuam na área de engenharia de reservatórios, utilizam informações sobre as propriedades e características das rochas e dos fluidos contidos nas formações portadoras de petróleo, bem como o seu comportamento passado (no caso de parte dos fluidos já ter sido produzida), para inferir o comportamento futuro desses reservatórios.

Cossé (1993, p. 3) afirma que o objetivo da engenharia de reservatórios, iniciando com a descoberta de um reservatório produtivo, é traçar um projeto de desenvolvimento que tente otimizar a recuperação dos hidrocarbonetos como parte de uma política geral de economia.

2.1.2 Rocha Reservatório

De acordo com a Universidade Federal do Ceará (2013), a rocha reservatório é uma formação rochosa com características adequadas à acumulação de petróleo, composta de grãos ligados uns aos outros por um material chamado cimento, juntamente com a matriz, um material muito fino.

Dois parâmetros são relevantes: a porosidade (Φ), que são os espaços vazios no interior da rocha que dependem da forma, arrumação e variação de tamanhos de grãos e a permeabilidade (k), que é a capacidade da rocha de transmitir o fluido, que depende, entre outros fatores, da quantidade, geometria e grau de conectividade dos poros (UNIVERSIDADE FEDERAL DO CEARÁ, 2013).

Segundo Rosa, Carvalho e Xavier (2006, p. 93), “A maioria dos depósitos comerciais de petróleo ocorrem em reservatórios formados por rochas sedimentares elásticas e não elásticas, principalmente em arenitos e calcários”.

A Figura 3 mostra em detalhes uma seção transversal de uma amostra de rocha.

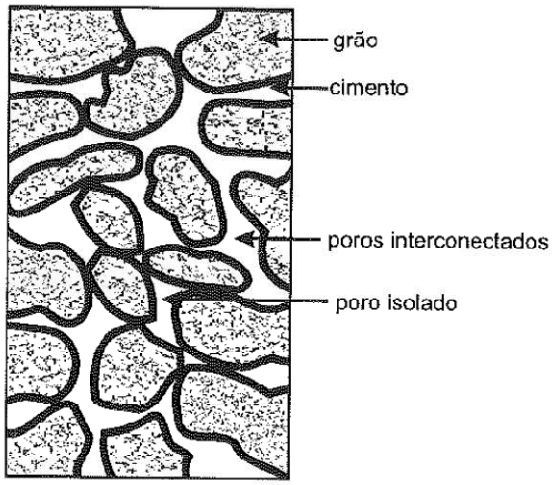


Figura 3 – Seção transversal de uma amostra de rocha
Fonte: Rosa, Carvalho e Xavier (2006, p. 93).

Os principais tipos de rochas reservatórios são os arenitos, as rochas carbonadas e outras rochas, tais como conglomerados e brechas, folhelhos fraturados, siltes, arcósios e rochas ígneas ou metamórficas fraturadas (ROSA; CARVALHO; XAVIER, 2006, p. 94).

2.1.3 Sistemas de Produção e Recuperação de Petróleo

A Figura 4 mostra um sistema completo de produção de óleo, composto de um reservatório, poço, dutos, separadores, bombas e tubulações de transporte.

> [...] o reservatório alimenta a boca do poço com óleo cru ou gás. O poço provê um caminho para o fluido de produção a ser conduzido do fundo do reservatório para a superfície e oferece um meio de controlar a taxa de produção do fluido. Os dutos conduzem o fluido produzido para os separadores. Os separadores removem o gás e a água do óleo cru. Bombas e compressores são utilizados para transportar óleo e gás até os pontos de vendas através de tubulações (GUO; GHALAMBOR, 2007, p. 1/4, tradução nossa).

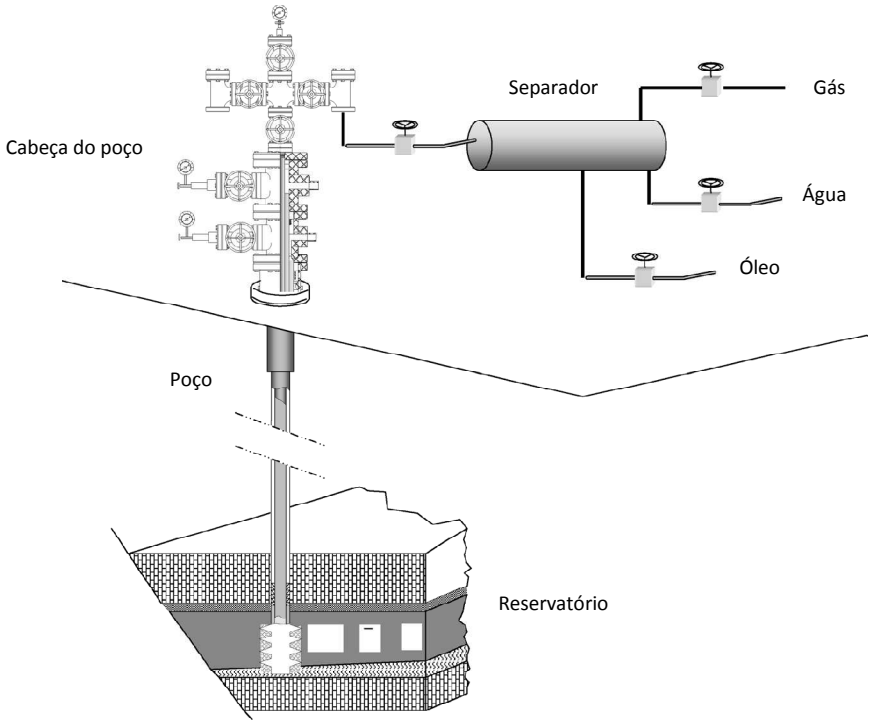


Figura 4 – Esboço de um sistema de produção de petróleo
Fonte: Guo e Ghalambor (2007, p. 1/4, tradução nossa).

Segundo Economides, Hill e Ehlig-Economides (1994, p. 2), os componentes de um sistema de produção de petróleo basicamente são os seguintes:

a) Reservatório

O reservatório é composto de uma ou mais unidades de fluxo interconectadas. Uma descrição apropriada do reservatório, incluindo a extensão das heterogeneidades, descontinuidades e anisotropias, além de ser importante, tornou-se obrigatória depois do surgimento de poços horizontais com comprimentos de vários milhares de pés.

A Figura 5 mostra um esquema apresentando dois poços, um vertical e um horizontal, inseridos dentro de um reservatório com potenciais heterogeneidades laterais ou descontinuidades (falhas de vedação), fronteiras verticais (lentes de xisto) e anisotropias (*stress* ou permeabilidade).

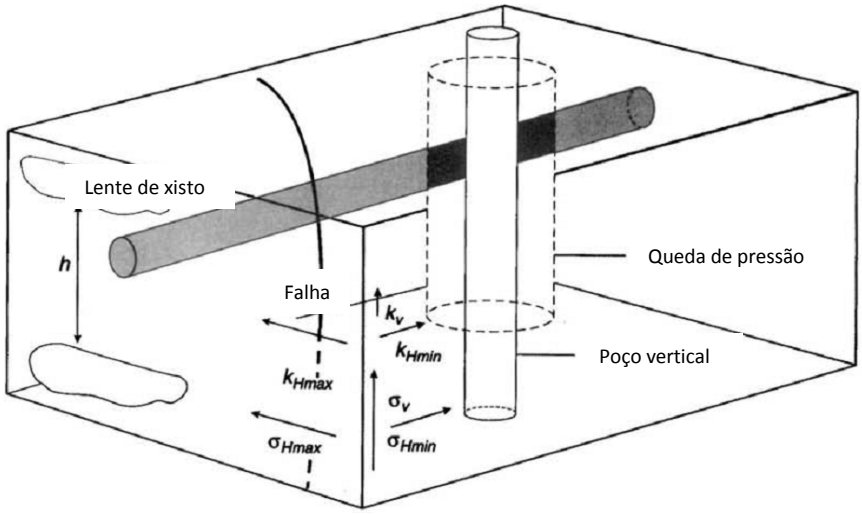


Figura 5 – Características comuns de um reservatório de petróleo
Fonte: Economides, Hill e Ehlig-Economides (1994, p. 3, tradução nossa).

b) Porosidade

A porosidade de um reservatório é a razão entre o volume dos poros V_p e o volume total do reservatório V_t e é dada pela relação abaixo.

$$\Phi = \frac{V_p}{V_t}$$

A porosidade é um dos primeiros parâmetros a serem medidos em qualquer esquema de exploração e um valor desejável é essencial para a continuação de quaisquer outras atividades de exploração do reservatório.

c) Altura do reservatório

A altura de um reservatório é a espessura de um meio poroso contido entre duas camadas geralmente consideradas impermeáveis. A presença de uma altura de reservatório satisfatória é um adicional imperativo em qualquer atividade de exploração. Na Figura 5 a altura do reservatório é representada pela letra h.

d) Permeabilidade

A permeabilidade é a propriedade que descreve a habilidade dos fluidos em fluírem em meios porosos. A presença de uma porosidade substancial na maioria dos casos implica que os poros estarão interconectados.

A correlação entre porosidade e permeabilidade deve ser utilizada com certo grau de cautela. Para efeitos de cálculos de engenharia de produção, essas correlações raramente são utilizadas.

O conceito de permeabilidade foi introduzido por Darcy em 1856 por meio de um experimento clássico, representado na Figura 6. A água flui através de uma coluna de areia e a diferença de pressão é registrada.

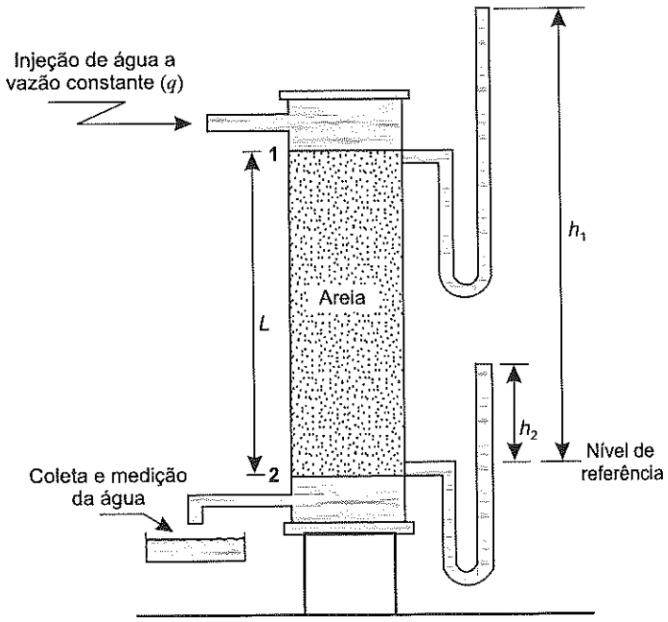


Figura 6 – O experimento de Darcy
Fonte: Rosa, Carvalho e Xavier (2006, p. 106).

Darcy observou que a razão de fluxo (ou velocidade) de um fluido através de um meio poroso específico é linearmente proporcional a diferença de pressão entre a entrada e a saída e a uma propriedade característica do meio (k). Assim,

$$u \propto k \, \Delta \, p$$

Dessa maneira, a equação da vazão conhecida como equação de Darcy pode ser escrita como (ROSA; CARVALHO; XAVIER, 2006, p. 106):

$$q = \frac{kA\Delta p}{\mu L}$$

Sendo q a vazão de fluido (cm^3/s), A a área da seção transversal (cm^2), Δp o diferencial de pressão (atm), μ a viscosidade do fluido (cp), L o comprimento do meio poroso (cm) e k a permeabilidade do meio poroso ($Darcy$). Tal equação foi estabelecida sob as seguintes condições:

- Fluxo isotérmico, laminar e permanente;
- Fluido incompressível, homogêneo e de viscosidade invariável com a pressão;
- Meio poroso homogêneo que não reage com o fluido.

e) A completação do poço

Muitos poços são cimentados e revestidos. Um dos objetivos da cimentação é dar suporte ao revestimento. Nas profundidades da formação, porém, a razão mais importante é prover uma zona de isolamento, evitando a contaminação do fluido a partir de outras formações ou a perda dos fluidos para outras formações. Um poço cimentado e revestido deve ser perfurado para restabelecer uma comunicação com o reservatório.

A Figura 7 apresenta as diversas opções para a completação de poços. São apresentados na sequência a completação a poço aberto, *liner* rasgado, *Gravel Pack* e revestimento canhoneado.

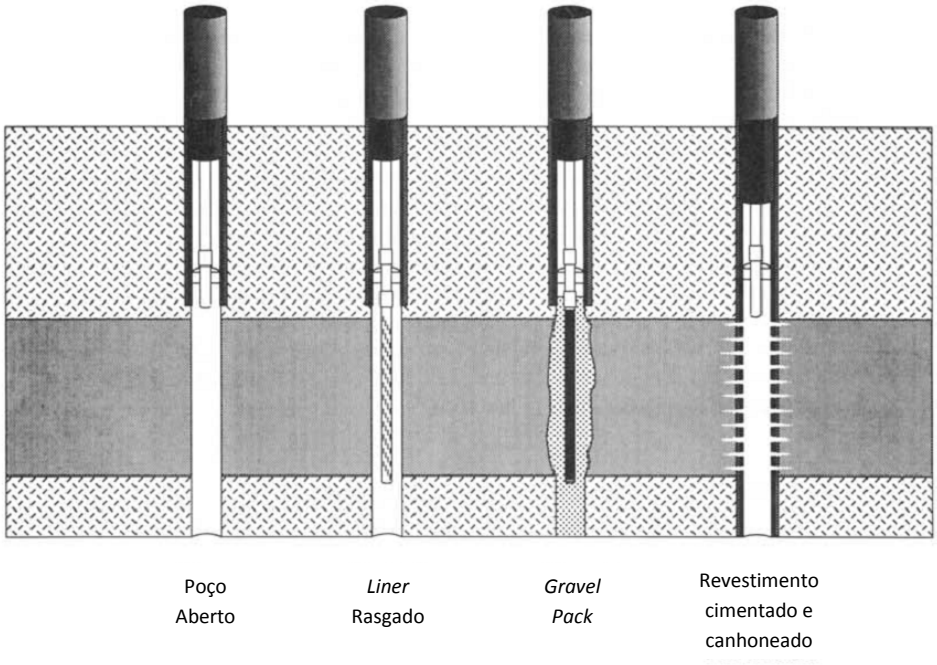


Figura 7 – Opções de completação para poços de petróleo
Fonte: Economides, Hill e Ehlig-Economides (1994, p. 3, tradução nossa).

f) O poço

A entrada de fluidos no poço – depois de passarem pelo meio poroso, a zona próxima ao poço e a completação – exige que os mesmos sejam elevados até a superfície.

Há um gradiente de pressão entre o fundo do poço e a cabeça do poço na superfície. Esse gradiente consiste da diferença de energia potencial (pressão hidrostática) e da queda de pressão de fricção.

Se a pressão do fundo de poço é suficiente para elevar os fluidos até a superfície então o poço está em elevação natural. Caso contrário, uma elevação artificial é indicada. Pode ser fornecida por uma bomba ou por meio da redução da densidade do fluido e por consequência, redução da pressão hidrostática. Isso é feito por meio da injeção de gás no poço, técnica conhecida como *gas lift*.

g) O equipamento de superfície

Assim que o fluido atinge a superfície, ele é direcionado para um *manifold* que conecta diversos poços. O fluido do reservatório é composto basicamente de óleo, gás e água.

Tradicionalmente, água, óleo e gás não são transportados grandes distâncias como essa mistura, mas são separados num complexo de processamento localizado próximo aos poços. Uma exceção é quando a exploração é feita em alto-mar.

Finalmente, os fluidos separados são transportados ou armazenados. A Figura 8 apresenta um sistema de produção de petróleo completo, incluindo o reservatório, a completação, o poço, a montagem da cabeça do poço e o equipamento de separação.

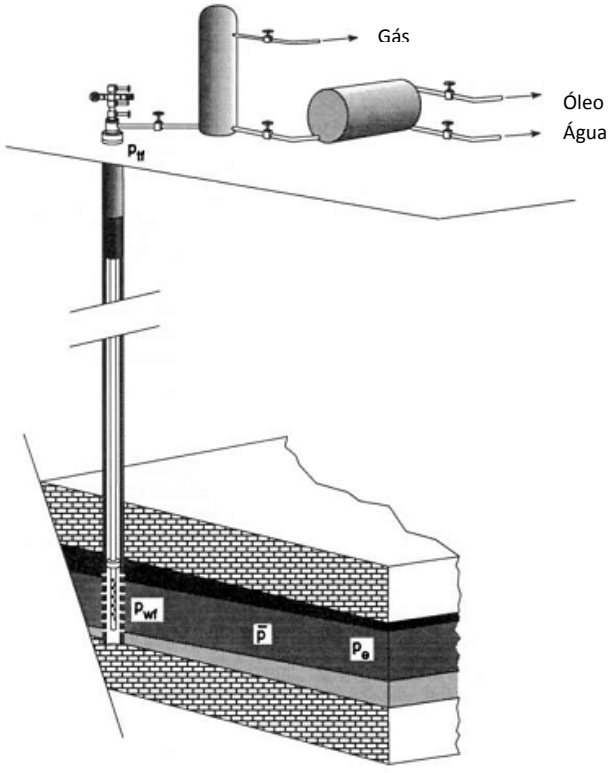


Figura 8 – Sistema de produção de petróleo
Fonte: Economides, Hill e Ehlig-Economides (1994, p. 11, tradução nossa).

2.1.4 Recuperação Primária e Secundária de Petróleo

Segundo Thomas *et al.* (2001, p. 200), os reservatórios que após a exaurição da sua energia natural (Recuperação Primária) ainda possuírem grandes quantidades de hidrocarbonetos são fortes candidatos a uma série de processos que visam à obtenção de uma recuperação adicional. Esses processos são conhecidos como Métodos de Recuperação e tentam interferir nas características do reservatório que colaboraram para a retenção exagerada do óleo.

A recuperação primária é a produção resultante da atuação da energia natural do reservatório. A um segundo esforço de produção dá-se o nome de recuperação secundária; a um terceiro, recuperação terciária e assim por diante. Talvez a única expressão que tem o mesmo significado em todas as referências seja a recuperação primária (THOMAS *et al*., 2001, p. 201).

> As acumulações de petróleo possuem, na época da sua descoberta, uma certa quantidade de energia, denominada energia primária. A grandeza dessa energia é determinada pelo volume e pela natureza dos fluidos existentes na acumulação, bem como pelos níveis de pressão e de temperatura reinantes no reservatório. No processo de produção há uma dissipação da energia primária, causada pela descompressão dos fluidos do reservatório e pelas resistências encontradas pelos mesmos ao fluírem em direção aos poços de produção (ROSA; CARVALHO; XAVIER, 2006, p. 561).

Segundo Rosa, Carvalho e Xavier (2006, p. 561), “o consumo de energia primária reflete-se principalmente no decréscimo da pressão do reservatório durante sua vida produtiva e consequente redução da produtividade dos poços”.

> A quantidade de óleo que pode ser retirada de um reservatório unicamente a expensas de suas energias naturais é chamada **recuperação primária**. Por outro lado, **recuperação secundária** é a quantidade adicional de óleo obtida por suplementação da energia primária com energia secundária, artificialmente transferida para a jazida, ou por meios que tendem a tornar a energia primária mais eficiente (ROSA; CARVALHO; XAVIER, 2006, p. 561, grifo nosso).

Ainda segundo Thomas *et al.* (2001, p. 200), os Métodos de Recuperação são classificados basicamente em duas grandes nomenclaturas: quando os processos possuem tecnologias bem conhecidas e o grau de confiança na aplicação é bastante confiável, os mesmos são conhecidos como *Métodos Convencionais de Recuperação*. Para os processos mais complexos, cujas tecnologias ainda não

estão satisfatoriamente desenvolvidas, dá-se o nome de *Métodos Especiais de Recuperação*.

2.1.4.1 Métodos Convencionais de Recuperação

Para Thomas *et al.* (2001, p. 201), os Métodos Convencionais de Recuperação são obtidos “ao se injetar um fluido em uma rocha reservatório com a finalidade única de deslocar o óleo para fora dos poros da rocha, isto é, buscando-se um comportamento puramente mecânico”.

Nos Métodos Convencionais de Recuperação são normalmente utilizados a injeção de água e o processo imiscível de injeção de gás, sendo que nesse último os fluidos não se misturam, ou seja, tanto o gás injetado quanto o óleo do reservatório permanecem como duas fases distintas durante o processo (ROSA; CARVALHO; XAVIER, 2006, p. 564).

Para Rosa, Carvalho e Xavier (2006, p. 564), já que o objetivo primordial da injeção é o aumento da recuperação de petróleo, esse volume adicional de petróleo a ser obtido com a injeção deve ser conseguido utilizando-se esquemas em que os volumes injetados sejam os menores possíveis. Alguns esquemas de injeção são apresentados a seguir.

a) Injeção Periférica, injeção no Topo e injeção na Base

Nesse grupo, tantos os poços de produção quanto os poços de injeção se concentram em determinadas áreas do reservatório.

Na Figura 9, a estrutura anticlinal do reservatório sugere o esquema de *Injeção Periférica*. A injeção da água é feita por meio de poços completados na base da estrutura do reservatório e nos mapas aparecem como se estivessem localizados na periferia do mesmo, dando origem assim ao nome desse esquema.

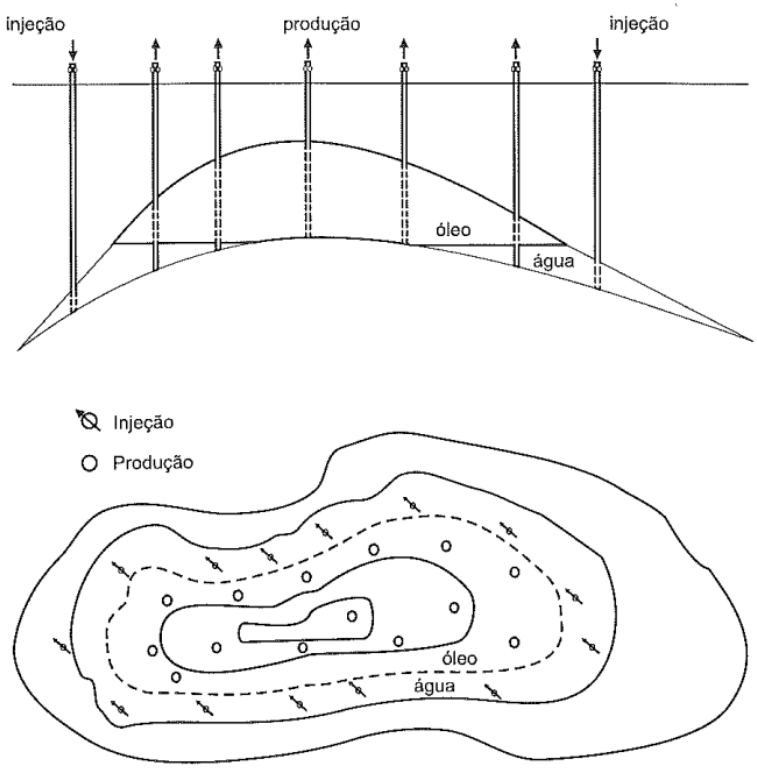


Figura 9 – Injeção periférica
Fonte: Rosa, Carvalho e Xavier (2006, p. 565).

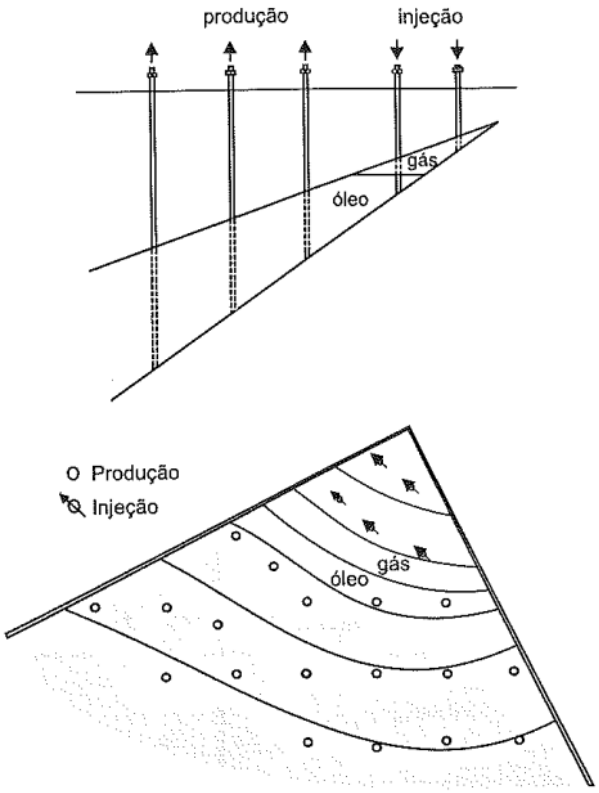


Figura 10 – Injeção no topo
Fonte: Rosa, Carvalho e Xavier (2006, p. 566).

Na Figura 10 é apresentado um exemplo de *Injeção no Topo*. A injeção de gás é feita no topo do reservatório enquanto a produção de óleo ocorre através de poços localizados na parte mais baixa.

Como a densidade do óleo é maior do que a do gás, existe a tendência do gás permanecer na parte mais alta da estrutura retardando sua chegada nos poços de produção.

Já na Figura 11 é apresentado o exemplo da *Injeção na Base*. O esquema de injeção é semelhante ao anterior, com a diferença que agora se injeta água na parte inferior da estrutura e os poços de produção são completados na parte alta da formação.

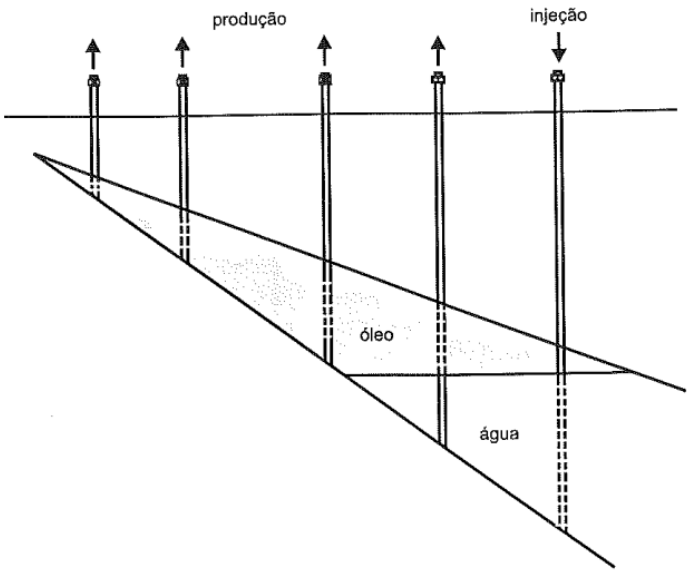


Figura 11 – Injeção na base
Fonte: Rosa, Carvalho e Xavier (2006, p. 566).

b) Injeção em Malhas

Nesse grupo de tipos de injeção tanto os poços de produção quanto os poços de injeção estão uniformemente distribuídos em toda a área do reservatório. O fluido é injetado na própria zona do óleo e são empregados em reservatórios com grandes áreas e pequenas inclinações e espessuras (ROSA; CARVALHO; XAVIER, 2006, p. 567).

O modelo em *linha direta* é composto por linhas alternadas de poços de injeção e de produção, sendo que a distância entre as linhas d e a distância entre cada um dos poços da linha a são constantes em cada projeto. A Figura 12 apresenta esse esquema de injeção em linha direta.

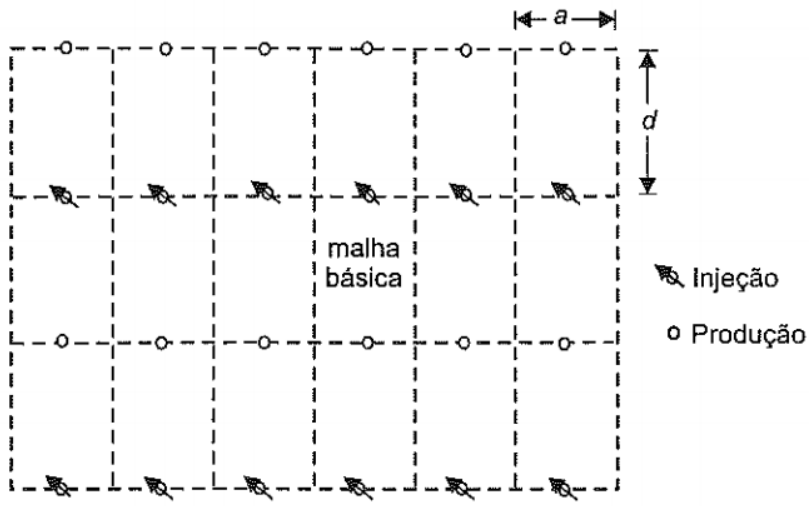


Figura 12 – Injeção em linha direta
Fonte: Rosa, Carvalho e Xavier (2006, p. 567).

Se as linhas forem defasadas em $a/2$, ou seja, meia distância dos poços de mesmo tipo, ter-se-á um novo esquema de injeção chamado de *linhas esconsas*, conforme pode ser visto na Figura 13.

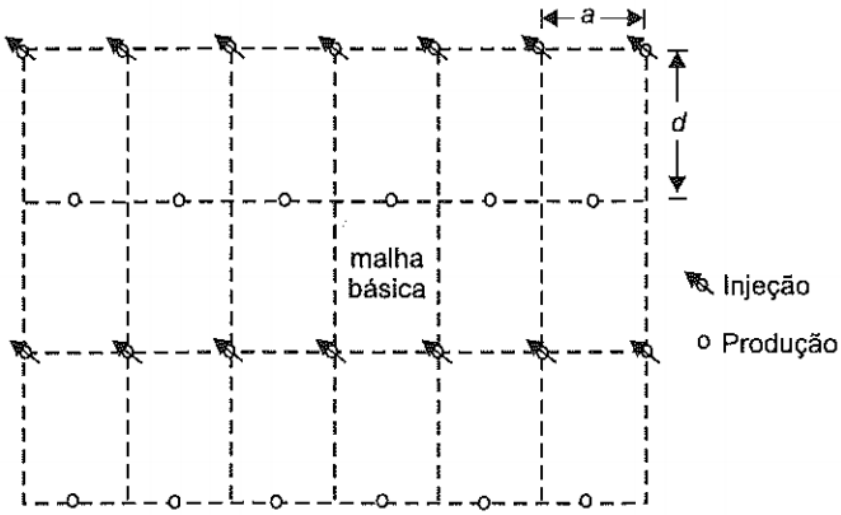


Figura 13 – Injeção em linhas esconsas
Fonte: Rosa, Carvalho e Xavier (2006, p. 567).

Para um caso particular do esquema de injeção em malhas em linhas esconsas, onde $d = a/2$, ou seja, a distância entre as linhas é igual à metade da distância entre os poços do mesmo tipo. Esse esquema é conhecido como modelo *five-spot* ou *malha de cinco pontos* e é um dos mais utilizados na recuperação secundária. O modelo *five-spot* está representado na Figura 14.

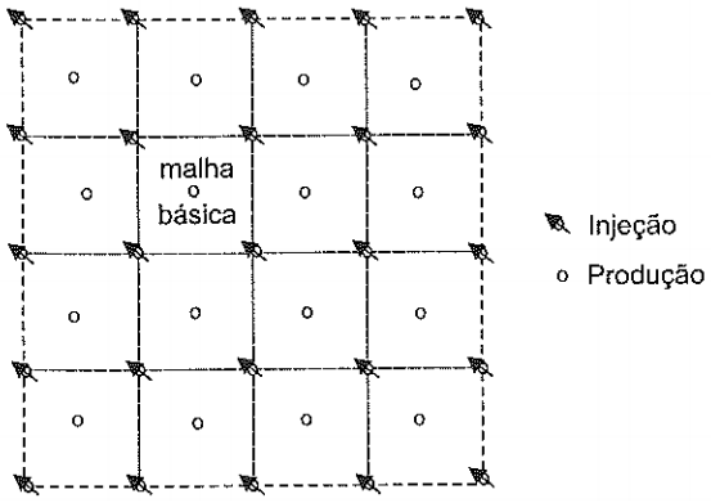


Figura 14 – Malha *five-spot*
Fonte: Rosa, Carvalho e Xavier (2006, p. 568).

As Figuras 15 e 16 representam respectivamente os modelos *seven-spot* ou malha de sete pontos e *nine-spo*t ou malha de nove pontos.

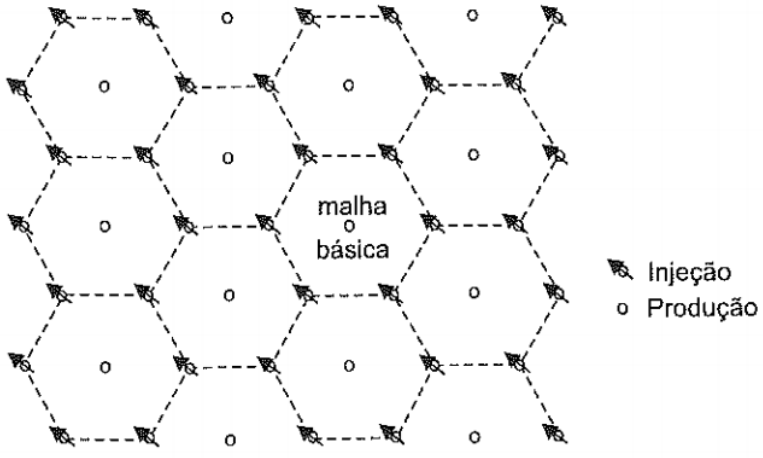


Figura 15 – Malha *seven-spot*
Fonte: Rosa, Carvalho e Xavier (2006, p. 568).

Todas as malhas aqui apresentadas são conhecidas como malhas *normais*, quando um poço de produção é cercado por vários poços de injeção. Existem modelos em

que a configuração é ao contrário, ou seja, um poço de injeção cercado por vários poços de produção. As Figuras 17 e 18 apresentam respectivamente os modelos *seven-spot* e *nine-spot* invertidos.

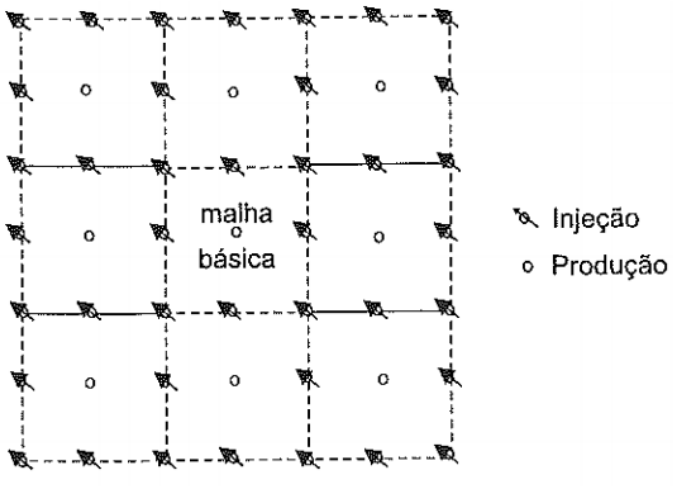


Figura 16 – Malha *nine-spot*
Fonte: Rosa, Carvalho e Xavier (2006, p. 568).

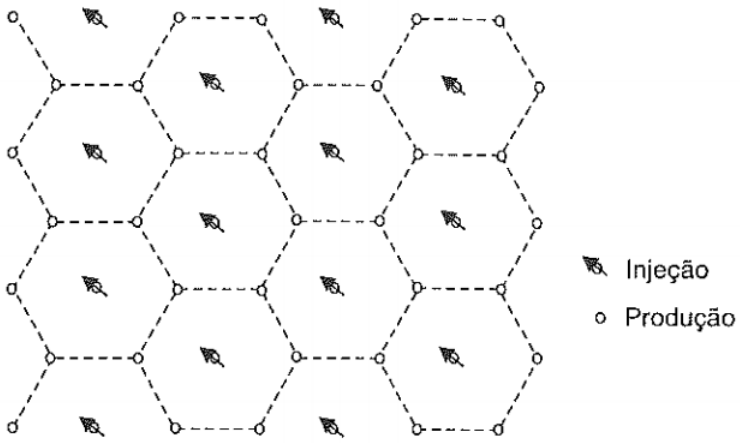


Figura 17 – Malha *seven-spot* invertido
Fonte: Rosa, Carvalho e Xavier (2006, p. 568).

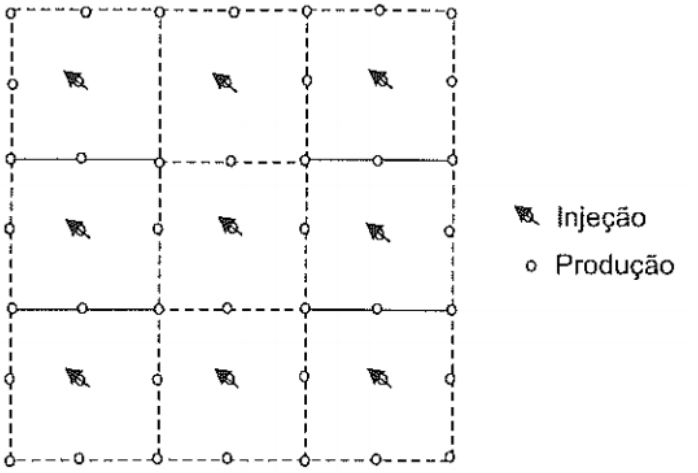


Figura 18 – Malha *nine-spot* invertido
Fonte: Rosa, Carvalho e Xavier (2006, p. 568).

2.1.4.2 Métodos Especiais de Recuperação

Os métodos especiais de recuperação são utilizados quando em determinados campos a injeção de água há algum tempo já atingiu estágios avançados de recuperação. Alguns desses reservatórios acabam ficando próximos do seu limite econômico e a aplicação desses métodos especiais de recuperação pode evitar que esses poços tenham que ser tamponados e abandonados (ROSA; CARVALHO; XAVIER, 2006, p. 676).

Dentre os principais métodos especiais de recuperação pode-se citar:

a) Métodos Miscíveis

São caracterizados por serem processos de recuperação de óleo com ausência de interface entre os fluidos deslocantes e deslocados. Esses processos reduzem as forças capilares e interfaciais que causariam a retenção do óleo no reservatório.

Classificam-se em:

- Injeção de hidrocarbonetos:
  - Injeção de banco miscível de GLP;

  - Injeção de gás enriquecido;
  - Injeção de gás pobre a alta pressão;
- Injeção de CO_2.

b) Métodos Térmicos

O objetivo da recuperação por meio dos métodos térmicos é aquecer o reservatório e o óleo nele existente para aumentar sua recuperação. Pode ser feito por meio da injeção de fluidos quentes ou da combustão *in-situ*, onde o calor é produzido dentro do próprio reservatório em vez de ser produzido na superfície e transportado para dentro do reservatório através de um fluido. Uma pequena porção do óleo do reservatório entra em ignição sustentada pela injeção de ar.

Os métodos térmicos classificam-se em:

- Injeção de fluidos quentes:
  - Injeção de água quente;
  - Injeção de vapor d'água;
- Combustão *in-situ.*

c) Métodos Químicos

Os métodos químicos têm por objetivo a diminuição da razão de mobilidade, que é a relação entre a mobilidade da água injetada (medida na saturação residual de óleo) e a mobilidade do óleo (medida na saturação da água conata).

Os métodos químicos classificam-se em:

- Injeção de polímero;
- Injeção de solução micelar;
- Injeção de solução *ASP*.

d) Outros Métodos

São os métodos que não se enquadram em nenhuma das categorias anteriores. Dentre os principais métodos, pode-se citar:

- Injeção de vapor com solvente;
- *SAGD*;
- Aquecimento eletromagnético;
- Injeção de ar;
- Injeção de Surfactante;
- Injeção de Soda Cáustica;
- *MEOR*;
- Controle da produção de água:
  - Gel bloqueador;
  - Modificador de permeabilidade relativa.

2.1.5 Simulação Numérica de Reservatórios

De acordo com Rosa, Carvalho e Xavier (2006, p. 517), “a simulação numérica é um dos métodos empregados na engenharia de petróleo para se estimar características e prever o comportamento de um reservatório de petróleo”.

Para Peaceman (1977, p. 1), a simulação de reservatórios é o processo de inferir o comportamento de um reservatório real a partir do desempenho de um modelo daquele reservatório. Esse modelo pode ser tanto *físico* (um modelo de laboratório em escala) ou *matemático*, representado por um conjunto de equações diferenciais com condições de contorno apropriadas.

Ao se resolver as equações diferenciais relacionadas ao modelo escolhido, condições de contorno apropriadas devem ser utilizadas. Apenas nos casos de modelos mais simples envolvendo reservatórios homogêneos e fronteiras muito regulares as soluções podem ser obtidas por meio dos métodos clássicos da matemática. No caso de modelos mais complexos, métodos numéricos são

utilizados em computadores de alto desempenho e têm logrado êxito ao obterem soluções para situações muito complexas de reservatórios. Um *modelo numérico* é então um programa de computador que utiliza métodos para obter soluções aproximadas ao modelo matemático (PEACEMAN, 1977, p .1).

2.1.5.1 Malhas

Uma das etapas da simulação numérica consiste na construção do modelo numérico propriamente dito. Para isso constrói-se uma malha ou *grid* para se transpor para o modelo as informações necessárias. Esta etapa consiste em dividir o reservatório em várias células, cada uma delas funcionando como um reservatório, como pode ser visto na Figura 19 (ROSA; CARVALHO; XAVIER, 2006, p. 523).

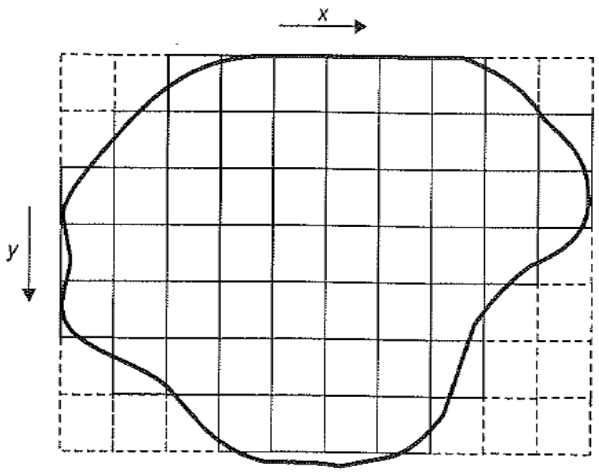


Figura 19 – Malha ou *grid* utilizado na simulação numérica de um reservatório
Fonte: Rosa, Carvalho e Xavier (2006, p. 524).

As malhas utilizadas pelos simuladores podem ser definidas de várias maneiras. A definição da orientação do sistema de coordenadas da malha varia de um simulador para outro e deve estar claramente definida para uso efetivo em um simulador específico. Tais malhas podem ser construídas em uma, duas ou três dimensões e em coordenadas cartesianas ou cilíndricas (FANCHI, 2006, p. 328).

A Figura 20 mostra uma orientação de malha escolhida para um determinado reservatório.

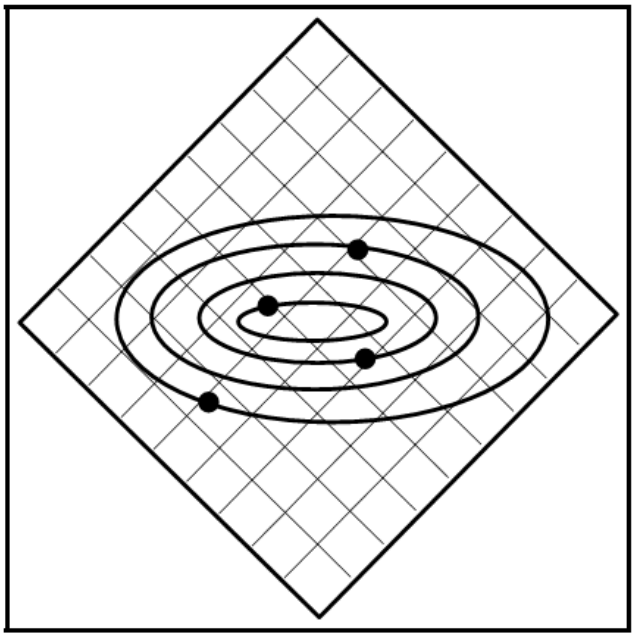


Figura 20 – Orientação da malha
Fonte: Fanchi (2006, p. 328).

A Figura 21 apresenta um exemplo de uma malha de reservatório em três dimensões.

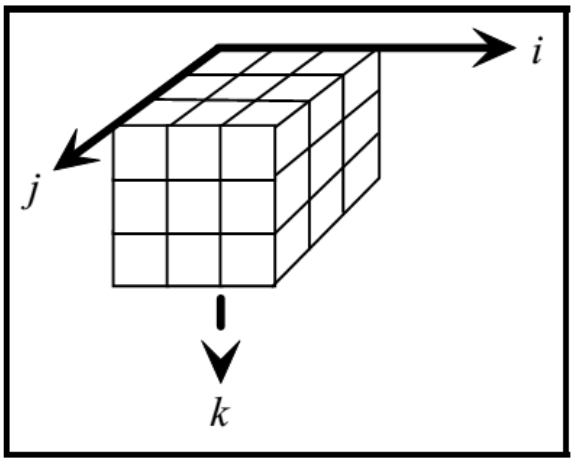


Figura 21 – Exemplo de uma malha 3-D
Fonte: Fanchi (2006, p. 329).

A Figura 22 mostra um exemplo de malha onde ocorrem o *LGR* e uma malha radial. O *LGR* é utilizado para fornecer um detalhamento adicional em alguns trechos selecionados da malha ou em uma malha muito grande.

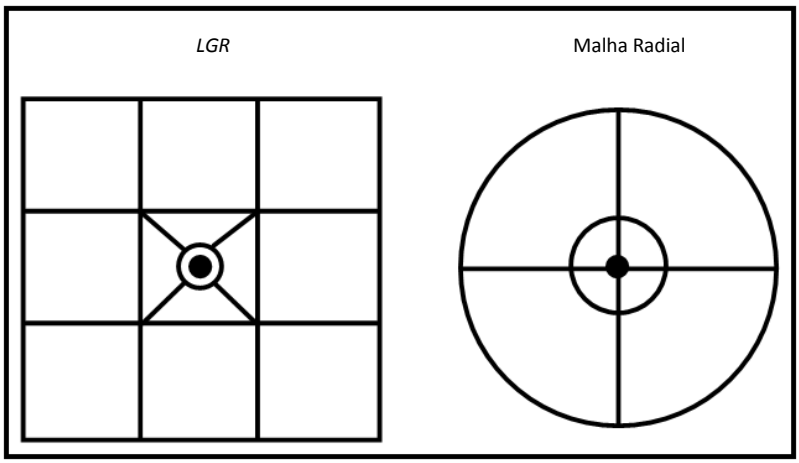


Figura 22 – Exemplos de malha não-cartesianas
Fonte: Fanchi (2006, p. 329, tradução nossa).

Para Marques (2005, p. 23), caso os centros dos volumes de controle adjacentes das malhas de discretização estejam distribuídos de forma que a “linha imaginária” que os une seja perpendicular à face comum entre eles então define-se essa malha como ortogonal ou regular, como pode ser visto na Figura 23. Tais malhas podem ser utilizadas em qualquer tipo de domínio, sendo esse complexo ou não. Caso não haja restrição de ortogonalidade entre os elementos que as compõem, as malhas são classificadas como não-ortogonais ou irregulares, conforme apresentado na Figura 24.

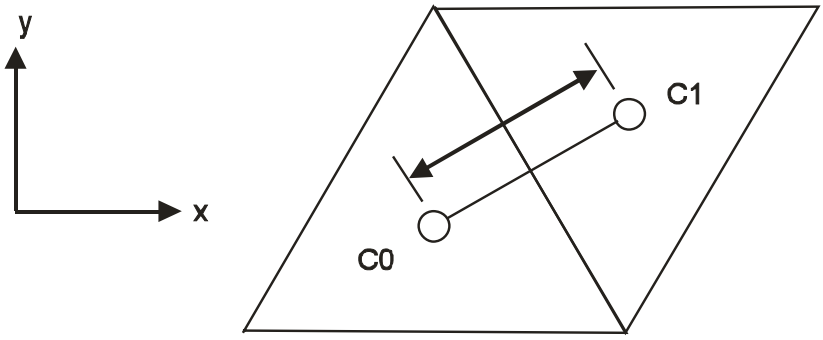


Figura 23 – Malhas ortogonais
Fonte: adaptado de Marques (2005, p. 23).

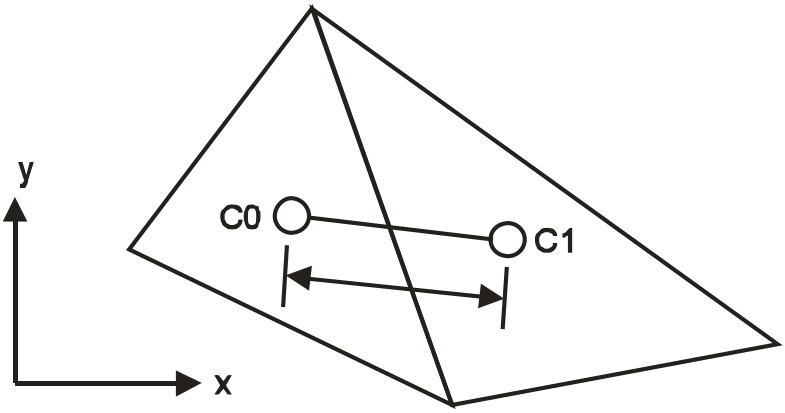


Figura 24 – Malhas não-ortogonais
Fonte: adaptado de Marques (2005, p. 24).

As malhas podem também ser classificadas em estruturadas ou não-estruturadas. Essa característica de estruturação do domínio discretizado é a existência ou não de ordem entre todos os volumes desse espaço. Essa ordem garante uma lei de formação entre os volumes e os nós do domínio, resultando assim em matrizes de coeficientes mais simples. Apesar disso ser uma vantagem, as malhas regulares podem não representar de forma adequada os locais do domínio onde a complexidade da geometria é grande (MARQUES, 2005, p. 25).

A Figura 25 apresenta uma geometria sendo representada por (a) malhas estruturadas ortogonais e (b) malhas não-estruturadas e não-ortogonais.

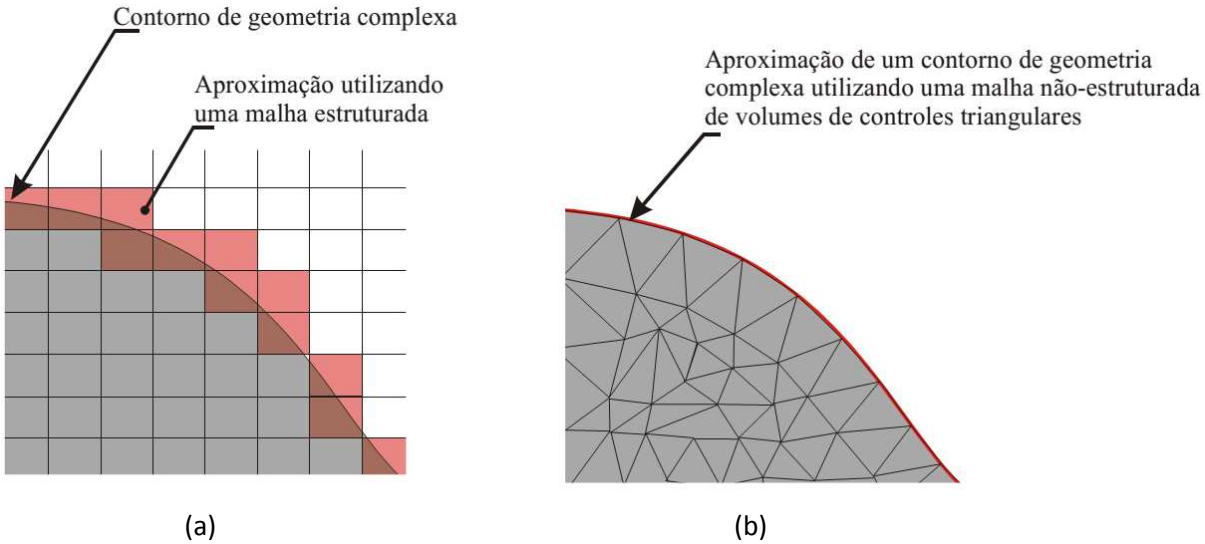


Figura 25 – Malhas regulares e não-regulares
Fonte: Marques (2005, p. 26).

Para Maliska (1995, *apud* MARQUES, 2005, p. 26):

> Diferentemente das malhas estruturadas, para as não-estruturadas não há ordem pré-definida entre os volumes no espaço discretizado. Isso implica que a matriz dos coeficientes resultante da discretização das equações terá banda variável o que impossibilita a aplicação de muitos métodos de solução de sistemas lineares. Outra característica é que o número de vizinhos pode variar de volume para volume.

Algumas vantagens podem ser citadas para que justifique a utilização de malhas não-estruturadas, como por exemplo, a adaptabilidade à geometrias mais complexas e o refinamento local do domínio que é obtido de forma mais eficiente nessas malhas (MARQUES, 2005, p. 27).

2.1.5.2 Leis básicas e princípios matemáticos

Na simulação de reservatórios, os fenômenos físicos são modelados por meio de diversas leis e princípios matemáticos.

Segundo Rosa, Carvalho e Xavier (2006, p. 520), as leis básicas que normalmente são empregadas nos simuladores, dependendo do seu tipo, são:

- Lei da conservação de massa;
- Lei da conservação de energia;
- Lei da conservação de “momentum”, dada por:

$$\sum F = \frac{\partial M}{\partial t}$$

Na qual F representa uma força e o “momentum” $M = mv$ caracteriza a dinâmica do sistema, sendo m a massa e v a velocidade.

Também dependendo do tipo de simulador numérico, um ou mais fenômenos podem ser considerados (ROSA; CARVALHO; XAVIER, 2006, p. 520):

a) Fluxo viscoso através de um meio poroso

As seguintes leis são geralmente utilizadas:

- Lei de Darcy (para fluxo “laminar”), dada pela expressão:

$$v_s = -\frac{k_s}{\mu}\frac{\partial \Phi}{\partial s}$$

Sendo k a permeabilidade efetiva do meio ao fluido considerado, μ a viscosidade do fluido, Φ o potencial de fluxo do fluido e s a trajetória de fluxo.

- Lei de Forchheimer (para fluxo “turbulento”). No caso de fluxo de um gás muitas vezes tal fluxo é turbulento, não podendo ser representado pela Lei de Darcy. A Lei de Forchheimer é dada pela expressão:

$$-\frac{dp}{ds} = \frac{\mu}{k_s} v_s - \beta \rho v_s^2$$

Sendo ρ a massa específica do fluido e β o coeficiente de resistência inercial ou de fluxo não-Darcyano.

b) Transferência de calor

Em alguns processos da simulação de reservatórios deseja-se considerar o fenômeno de transferência de calor no meio poroso. As seguintes leis são geralmente utilizadas:

- Condução, dada pela expressão conhecida como Lei de Fourier:

$$q_s = -k' \frac{\partial \mathrm{T}}{\partial s}$$

Sendo T a temperatura e k' a condutividade térmica do meio.

- Convecção, dada pela expressão:

$$q_s = c_p v_s (T - T_0)$$

Sendo c_p a capacidade calorífica do fluido à pressão constante, v a velocidade do fluido e T_0 uma temperatura de referência.

O uso de equações de estado apropriadas é fundamental na obtenção das equações diferenciais que representam o escoamento dos fluidos através do reservatório. As equações de estado podem ser divididas em (ROSA; CARVALHO; XAVIER, 2006, p. 521):

a) Fluidos

Para os fluidos normalmente é empregada como equação de estado uma lei que relacione pressão com a massa especifica, com tal relação geralmente obtida através da equação da compressibilidade.

Para os líquidos, a expressão da definição geral da compressibilidade isotérmica de um fluido é dada por:

$$c = -\frac{1}{V}\frac{\partial \mathrm{V}}{\partial p} = \frac{1}{\rho}\frac{\partial \rho}{\partial p}$$

Sendo ρ o valor da massa específica. No caso de líquidos com compressibilidade constante, o valor de ρ é dado por:

$$\rho = \rho_0 e^{c(p-p_0)}$$

No caso de líquidos com compressibilidade constante e pequena, o valor de ρ é dado por:

$$\rho = \rho_0 [1 + c(p - p_0)]$$

Para os gases, aplica-se a chamada Lei dos Gases, segundo a qual a massa específica relaciona-se com a pressão (p), a massa molecular (M), o fator de compressibilidade (Z), a constante universal dos gases (R) e a temperatura (T).

Para um gás real, a expressão é dada por:

$$\rho = \frac{pM}{ZRT}$$

E para um gás ideal, a expressão é dada por:

$$\rho = \frac{pM}{RT}$$

b) Sólidos

Para os sólidos, o comportamento da rocha é representado pela equação da chamada compressibilidade efetiva, dada pela expressão:

$$c_f = \frac{1}{\Phi}\frac{\partial \Phi}{\partial \mathrm{p}}$$

Na qual c_f é a compressibilidade efetiva da formação e Φ é a porosidade.

Ao se modelar um reservatório de óleo onde há fluxo em meios porosos geralmente são utilizadas a Lei de Darcy e o princípio da conservação da massa. Resultam dessa modelagem EDP não lineares que em sua grande maioria não podem ser resolvidas de forma analítica (SILVA, 2006, p. 16).

Além da permeabilidade da rocha, a possível presença de várias camadas sedimentares depositadas de diferentes maneiras dentro do reservatório criam direções preferenciais ao fluxo de fluidos fazendo do mesmo um meio anisotrópico. Além do exposto, a geometria do reservatório com suas falhas e camadas selantes faz com que o uso dos métodos numéricos seja a alternativa mais indicada para a

obtenção de dados que possam determinar de forma aproximada a otimização da produção em reservatórios por meio da simulação computacional dos mesmos (SILVA, 2006, p. 16).

2.1.5.3 Discretização de domínio e particionamento de malhas

Apesar dos modelos matemáticos analíticos existirem para a maioria dos fenômenos que ocorrem dentro do reservatório, na prática, suas resoluções não são triviais ou possíveis. Dessa forma, aproximações numéricas devem ser utilizadas para que sejam encontradas soluções otimais.

De acordo com Aziz e Settari (1979, *apud* SILVA, 2008, p. 16), “A ideia básica de qualquer método aproximado é substituir o problema original por outro mais fácil de ser resolvido e cuja solução é, de alguma forma, próxima da solução do problema original”.

Ao se utilizar métodos numéricos, o domínio contínuo deve ser aproximado por meio de uma discretização de pontos utilizando-se uma malha.

> [...] O processo de discretização numérica consiste em aproximar a geometria do reservatório através de uma malha e aplicar as equações do problema sobre os pontos discretos desta malha, a cada passo de tempo, resultando com isso em uma série de sistemas de equações lineares discretas, que são resolvidos empregando-se os métodos de solução de sistemas lineares adequados (SOARES, 2002, p. 2).

Quando um domínio é discretizado por meio de uma malha, quanto mais pontos forem utilizadas nessa malha, mais próximo do domínio contínuo é essa representação.

Segundo Soares (2002, p.2),

> Para representar da melhor maneira possível as heterogeneidades existentes em reservatórios reais é necessário usar malhas bastante refinadas, o que resulta geralmente em modelos de larga escala, com grande quantidade de dados a serem armazenados em memória e elevado número de equações no sistema, sendo na maioria das vezes impossível

> resolvê-los utilizando os recursos computacionais disponíveis em um único microcomputador ou estação de trabalho atuais.

Como as EDP são resolvidas sobre os pontos da malha discretizada, quanto mais pontos essa malha possuir, maior será a carga computacional para que o simulador de reservatórios a resolva. Dessa forma, limitações de processamento em computadores, especialmente em computadores seriais, podem impedir o refinamento apurado de malhas de discretização, pois isso pode tornar a execução da simulação inviável. Nesse cenário, o uso de computadores paralelos tornou-se uma opção viável.

Citando Silva (2008, p. 16), “O desenvolvimento de computadores paralelos tornou possível conduzir simulações de grande porte em reservatórios de petróleo. Na última década, o número total de blocos usados em simulações típicas de reservatório cresceu de milhares para milhões”

Os computadores paralelos de memória distribuída, ao executarem simulações que utilizem malhas discretizadas de representação de domínios, dividem os pontos da malha para cada um dos nós processadores, para que cada um deles execute seu processamento sobre esses nós.

Ao se submeter uma malha que representa um determinado domínio a uma simulação em computadores paralelos de memória distribuída, alguma medidas devem ser observadas para que o desempenho geral da simulação não seja comprometido (SILVA, 2008, p. 86; AL-NASRA; NGUYEN, 1991 *apud* GALANTE 2006, p. 27):

- Cada nó processador do computador paralelo deve receber, se possível, a mesma quantidade de nós de maneira a evitar a sobrecarga em alguns nós processadores em detrimento da ociosidade de outros;
- O número de nós que devem acessar informações em nós remotos (localizados em outros nós processadores) deve ser o menor possível, de forma a minimizar a troca de informações entre os mesmos;

- O tempo gasto no particionamento da malha que representa o domínio deve ser pequeno se comparado com o tempo total da solução do problema de simulação;
- O algoritmo de particionamento de domínios deve ser capaz de tratar geometrias irregulares e discretizações arbitrárias.

Manter o balanceamento da carga e a minimização da quantidade de nós fronteiriços da malha discretizada entre os diferentes nós processadores de um computador paralelo de memória distribuída, são condições primordiais para uma diminuição do tempo de execução de uma simulação que utilize tal configuração. Em primeiro lugar, para que não haja sobrecarga ou ociosidade nos nós processadores; em segundo lugar, os nós processadores dos computadores paralelos de memória distribuída utilizam uma rede local de computadores para a troca de informações entre seus nós vizinhos. Caso esse número de acessos seja elevado, o desempenho geral da simulação pode ser comprometido, já que acessos aos valores dos pontos da malha utilizando a rede de comunicação são muito mais lentos do que acessos a esses pontos que estejam presentes na mesma memória física do nó processador.

> A distribuição da carga de trabalho, considerando-se a arquitetura disponível e exigindo o balanceamento de carga e minimização da comunicação dos processos, durante o tempo de execução, é uma das etapas mais importantes da Computação Científica Paralela (RIZZI, 2002, *apud* GALANTE, 2006, p. 27).

Portanto, torna-se imprescindível um bom particionamento da malha de discretização entre os diversos nós processadores do computador paralelo. Tal assertiva é ratificada por Soares e Araújo (2002, p. 2972, tradução nossa, grifo nosso):

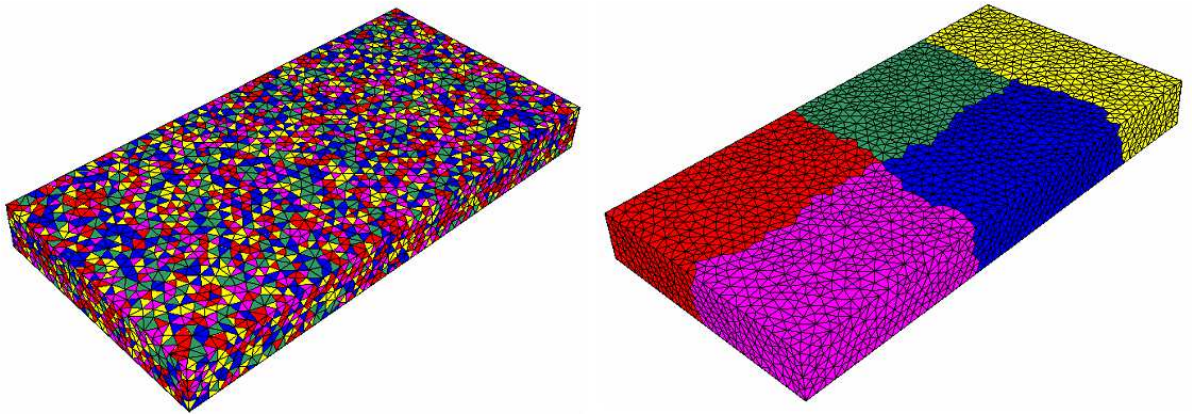
> Em determinados momentos, para realizar os cálculos, cada processador irá precisar de cópias de valores dos processadores vizinhos. Durante a simulação, essas cópias, chamados de “valores fantasmas”, precisam ser atualizados, o que requer a comunicação entre os processadores. **A minimização do tempo gasto nessas comunicações é muito importante para a eficiência paralela do simulador**.

Dentro da literatura, problemas de particionamento de malhas são tratados como problemas de particionamento de grafos. Dessa forma, ao se particionar um grafo

que representa uma malha de discretização assegurando as medidas citadas por Silva (2008, p. 86) e Al-Nasra e Nguyen (1991 *apud* GALANTE 2006, p. 27) no texto acima, os tempos de execução da simulação do reservatório podem ser diminuídos.

> O problema da divisão do domínio computacional pode ser modelado como um problema de particionamento de grafos, onde os vértices representam os pontos da malha e as arestas a relação de vizinhança entre esses. Sob essa abordagem, o particionamento da malha pode ser visto como o problema de k-particionamento de grafos, que consiste em dividir um grafo em k subgrafos, de modo que cada subgrafo contenha um número semelhante de vértices, e que o número de arestas entre os subgrafos seja o menor possível (DORNELES, 2003, *apud* GALANTE, 2006, p.27).

A Figura 26 mostra um exemplo de uma malha de discretização 3-D sendo particionada para a execução em cinco nós processadores diferentes. Cada cor representa um processo. O balanceamento de carga é assegurado, ou seja, cada nó processador recebe aproximadamente a mesma quantidade de elementos da malha para realizar o processamento da simulação.



(a) (b)

Figura 26 – Exemplos de particionamento de malha em cinco processos
Fonte: Silva (2008, p. 87).

Na Figura 26 (a), os elementos da malha foram divididos entre os nós processadores seguindo a ordem de conectividade do arquivo da malha. A quantidade de nós fronteiriços é muito alta. Já na Figura 26 (b), uma distribuição otimizada garante que a quantidade de elementos que tenham que comunicar-se com nós vizinhos seja minimizada, implicando numa diminuição do tempo gasto para envio e recebimento dos dados entre esses nós, e consequentemente, aumentando o desempenho do simulador paralelo que trabalhe sobre essa malha.

2.2 COMPUTAÇÃO PARALELA

Por muitos anos, a programação paralela tem se estabelecido na computação científica de alto desempenho. As simulações são cruciais nessa área. Simulações mais precisas ou a simulação de problemas maiores necessitam de mais e mais poder computacional e memória. Nas últimas décadas, pesquisas de alto desempenho incluíram novos desenvolvimentos em tecnologias de *hardware* e *software* paralelos. Dentre os exemplos populares de simulações podem-se citar as previsões meteorológicas, desenvolvimento de novas drogas e medicamentos, simulações de túneis de vento para a indústria automobilística e aplicações de computação gráfica para a indústria cinematográfica (RAUBER; RÜNGER, 2010, p. 1).

De acordo com Culler, Singh e Gupta (1998, p. 19), o desempenho e a capacidade dos sistemas computacionais têm experimentado um crescimento explosivo por mais de uma década. Fatores como a tecnologia *VLSI* e aumento de frequências dos *clocks* dos processadores contribuíram para esse crescimento.

Para Gebali (2011, p. 1), apesar de todo esse crescimento, a ideia de um computador com apenas um único processador está tornando-se arcaica e estranha. Algumas estratégias com relação à programação paralela têm que ser levadas em consideração:

- Não é simples melhorar o desempenho de um computador utilizando apenas um único processador. Esse processador consumiria uma energia inaceitável. É mais prático utilizar vários processadores para obter o desempenho desse único processador mais robusto;
- Ferramentas de programação que possam detectar paralelismo para um dado algoritmo têm que ser desenvolvidas;
- Otimizar o desempenho dos futuros computadores vai engrenar uma boa programação em paralelo em diversos níveis: algoritmos, desenvolvimento de programas, sistemas operacionais, compiladores e *hardware*;

- Os benefícios da computação paralela têm que levar em consideração o número de processadores implantados bem como o *overhead* de comunicação processador-processador e processador-memória;
- Os sistemas de memória ainda são muito mais lentos do que os processadores e sua largura de banda é limitada a uma palavra por ciclo de leitura/escrita.

A Taxonomia de Flynn é uma categorização das formas de arquiteturas paralelas de computador. Desenvolvida em 1966 e tendo seu uso difundido em 1972, essa é uma metodologia para classificar, de maneira geral, as operações paralelas dentro de um processador. Foi proposta como uma abordagem para esclarecer os tipos de paralelismo suportados por um *hardware* ou disponíveis em uma aplicação. A taxonomia utiliza o conceito de fluxo para categorizar a arquitetura do conjunto de instruções. Um fluxo é simplesmente uma sequência de objetos ou ações. Existem tanto fluxos de instruções quanto fluxos de dados e há quatro combinações que descrevem as arquiteturas paralelas mais familiares (PADUA, 2011, p. 690; RAUBER, T; RÜNGER, 2010, p. 11):

1) *Single Instruction, Single Data* (*SISD*) – Há uma única EP que tem que acessar um único programa e um único local de armazenamento de dados. Em cada passo, a EP carrega uma instrução e o dado correspondente e executa a instrução. O resultado é armazenado novamente no local de armazenamento de dados. É exemplo de *SISD* o computador sequencial de acordo com o modelo de Von Neumann, com o processador tradicional que inclui *pipeline*, superescalar e processadores *VLIW*. A Figura 27 mostra uma única instrução operando sobre uma única unidade de dados por meio de uma EP.

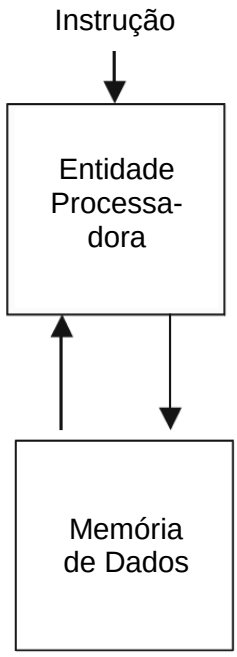


Figura 27 – *SISD*: única instrução operando sobre uma única unidade de dados
Fonte: Padua (2011, p. 690, tradução nossa).

2) *Single Instruction, Multiple Data* (*SIMD*) – Existem múltiplas EPs, cada uma delas com um acesso privado a uma memória de dados (compartilhada ou distribuída). Há apenas uma memória de programas a partir da qual um processador de controle especial busca e envia instruções. Em cada etapa, cada EP obtém do processador de controle a mesma instrução e carrega um elemento de dados separado por meio de seu acesso a dados privados em que a instrução é executada. Assim, a instrução é aplicada de forma síncrona em paralelo por todos as EPs em diferentes elementos de dados. São exemplos os processadores matriciais e os processadores vetoriais. A Figura 28 mostra um processador matricial.

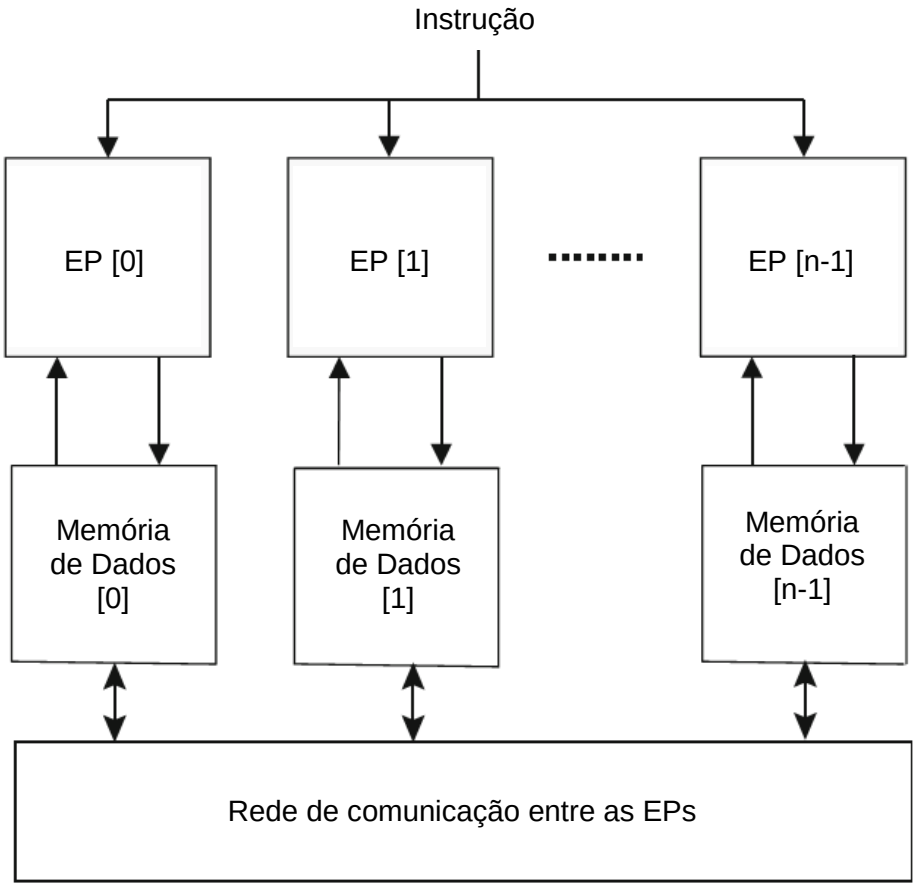


Figura 28 – *SIMD*: processador matricial utilizando EPs com memória local
Fonte: Padua (2011, p. 691, tradução nossa).

3) *Multiple Instruction, Single Data* (*MISD*) – Existem várias EPs em que cada uma tem uma memória de programa particular, mas há apenas um acesso comum para uma memória única de dados global. Em cada etapa, cada EP obtém o mesmo elemento de dado da memória de dados e carrega uma instrução a partir da sua memória de programa privada. Isso possibilita que diferentes instruções sejam então executadas em paralelo pelas EPs usando os elementos de dados (idênticos) obtidos anteriormente como operandos. Este modelo de execução é muito restritivo e nenhum computador paralelo comercial deste tipo já foi construído. Como exemplos podem-se citar os vetores sistólicos, as *GPU*s e as máquinas de fluxo de dados. A Figura 29 exemplifica um processador de fluxo.

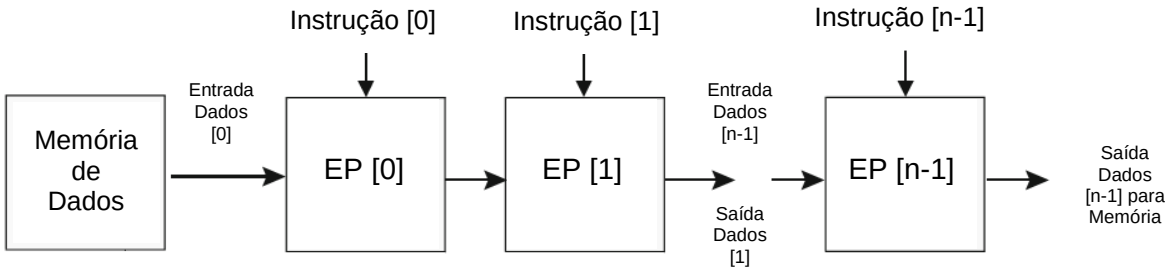


Figura 29 – *MISD*: exemplo de um processador de fluxo
Fonte: Padua (2011, p. 691, tradução nossa).

4) *Multiple Instruction, Multiple Data* (*MIMD*) – Há múltiplas EPs cada qual com uma instrução separada e acesso a dados a um programa (compartilhado ou distribuído) e dados da memória. Em cada etapa, cada EP carrega uma instrução separada e um elemento de dados separado; aplica a instrução ao elemento de dados e armazena um possível resultado de volta para o local de armazenamento de dados. As EPs funcionam de modo assíncrono uma com a outra. Processadores multinúcleo e sistemas de *cluster* são exemplos para o modelo *MIMD*. A Figura 30 mostra um processador matricial com EPs com memória local.

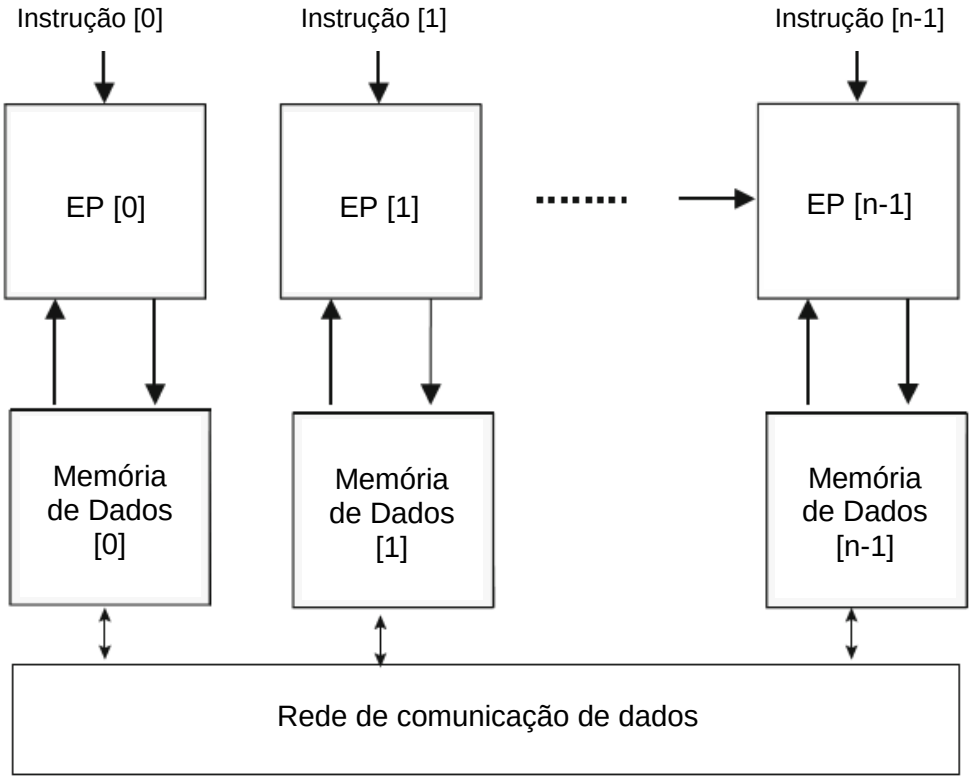


Figura 30 – *MIMD*: processador matricial utilizando EPs com memória local
Fonte: Padua (2011, p. 692, tradução nossa).

2.2.1 *Clusters*

Uma definição bem concisa de *clusters* de computadores é dada por Rauber e Rünger (2010, p. 15, tradução nossa):

> Conjuntos de computadores completos com uma rede de interconexão dedicada são muitas vezes chamado de *clusters*. *Clusters* são geralmente baseados em computadores padrão e até mesmo topologias de rede padrão. O conjunto inteiro é endereçado e programado como uma unidade. A popularidade de *clusters* como máquinas paralelas vem da disponibilidade de interconexões de alta velocidade padrão como *FCS*, *SCI*, *Switched Gigabit Ethernet*, *Myrinet* ou *InfiniBand*. Um modelo de programação natural das máquinas de memória distribuída como os *clusters* é o modelo de passagem de mensagens que é suportado pelas bibliotecas de comunicação, como *MPI* ou *PVM*. Essas bibliotecas são muitas vezes baseadas em protocolos padrão como *TCP/IP*.

A Figura 31 apresenta a arquitetura típica de um *cluster* de *PCs*.

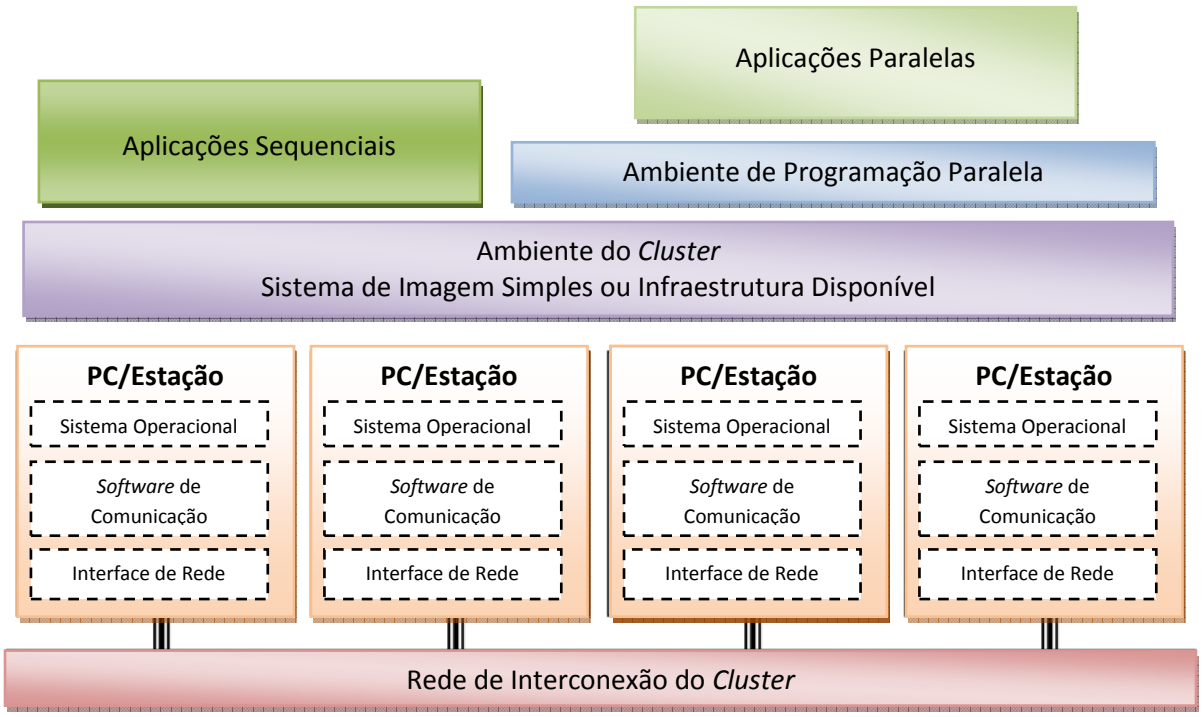


Figura 31 – Arquitetura de um *cluster* de *PCs*
Fonte: Buyya (1999, tradução nossa).

A Figura 32 mostra um exemplo de um *cluster* de computadores construído com equipamentos específicos. Esse *cluster* pertence ao Programa de Pós-Graduação em Energia do Centro Universitário Norte do Espírito Santo (PPGE-CEUNES/UFES).

Figura 32 – *Cluster* do PPGE-CEUNES/UFES

2.2.1.1 *Clusters* de Alta Disponibilidade

De acordo com Pitanga (2008, p. 26), os *Clusters* de Alta Disponibilidade (*HA - High Availability*) são sistemas onde a disponibilidade de serviços é o foco principal a ser atendido pelo mesmo. A alta disponibilidade é a garantia de continuidade da operação do sistema em serviços de rede, armazenamento de dados ou ;processamento, mesmo se houver falhas em um ou mais dispositivos de *hardware* ou *software*. Nos *clusters HA* os nós são utilizados em conjunto para manter um serviço ou equipamento sempre ativo, replicando serviços e servidores, o que evita sistemas parados, já que as demais máquinas passam a responder por aquela que teve o serviço interrompido.

De maneira geral, um servidor de boa qualidade apresenta uma disponibilidade de 99,5%, enquanto que uma solução utilizando *clusters HA* apresenta uma disponibilidade de 99,99%. São exemplos de trabalhos em *GNU/Linux open-source* sobre *clusters HA*: *Linux Virtual Server* (*LVS*), *Eddieware*, *Heartbeat*, *Mon* e *DRDB*.

As Figuras 33 e 34 mostram respectivamente exemplos dos *clusters* de balanceamento de carga e de alta disponibilidade (PITANGA, 2008, p. 26).

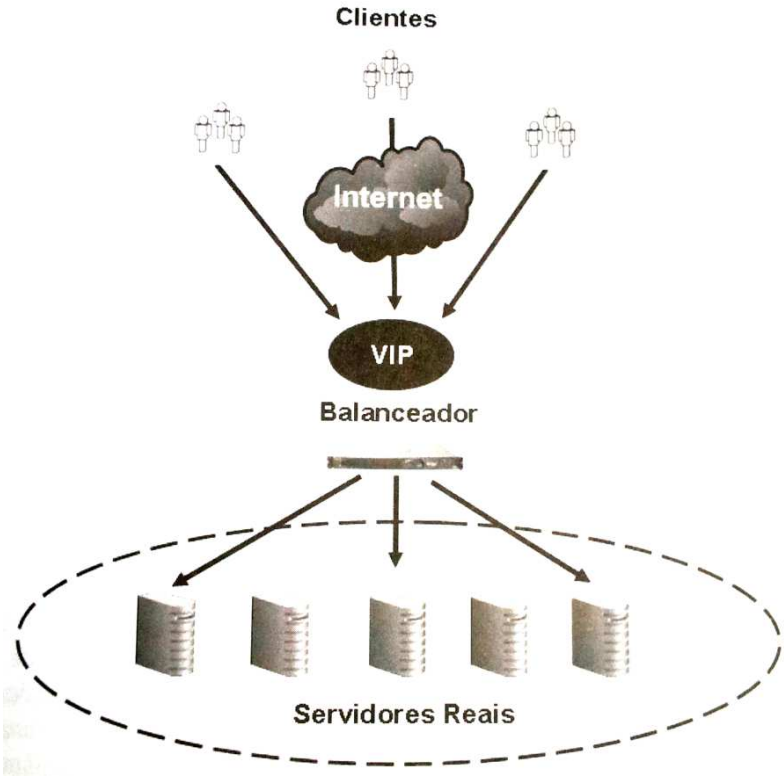


Figura 33 – *Cluster* de Balanceamento de Carga
Fonte: Pitanga (2008, p. 27).

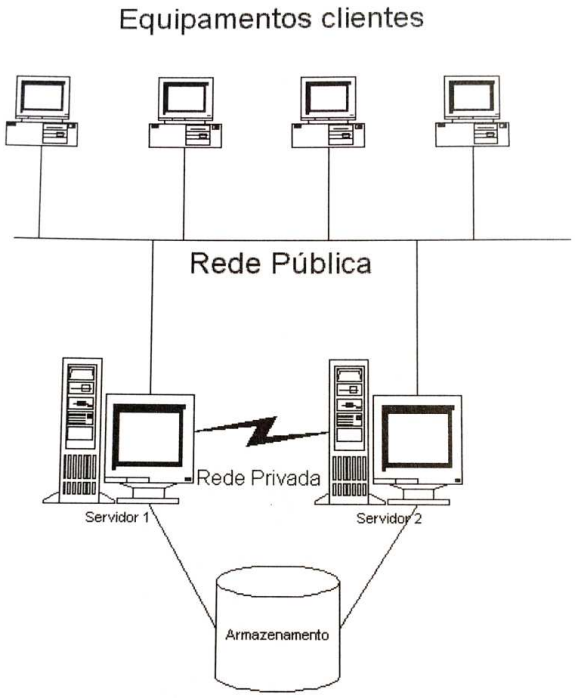


Figura 34 – Um *cluster* simples de Alta Disponibilidade
Fonte: Pitanga (2008, p. 28).

2.2.1.2 *Clusters* de Computação de Alto Desempenho

Ainda segundo Pitanga (2008, p. 28), os *Clusters* de Computação de Alto Desempenho (*HPC - High Performance Computing*) são um tipo de sistema para processamento paralelo ou distribuído, que consiste de uma coleção de computadores interconectados, trabalhando juntos como um recurso de computação simples e integrado. Um nó do *cluster* pode ser um simples sistema multiprocessador (*PCs*, estações de trabalho ou *SMPs*) com memória, dispositivos de entrada/saída de dados e um sistema operacional. No entanto, esse sistema pode fornecer características e benefícios encontrados somente em sistemas de memória compartilhada (*SMP*) como os supercomputadores.

Os *clusters HPC* são uma alternativa excelente para universidade e empresas de pequeno porte a médio porte para obterem processamento de alto desempenho na resolução de problemas por meio de aplicações paralelizáveis. Tudo isso a um custo relativamente baixo se comparado com os valores para se adquirir um supercomputador com a mesma capacidade de processamento (PITANGA, 2008, p. 28).

2.2.1.3 *Clusters PoPC* e *Beowulf*

De acordo com Pitanga (2008, p.46), um *cluster* de *PCs* aplicado na resolução de um ou mais problemas é conhecido pelo termo *PoPC*. Possui as seguintes características:

- Uso de componentes comuns, disponíveis no mercado tradicional de informática;
- Processadores dedicados na execução das tarefas;
- Rede de sistema privada para os nós computacionais.

Para ser considerado um *cluster Beowulf*, a *PoPC* precisa seguir ainda as seguintes características:

- Nenhum componente feito sob encomenda;
- Independência de fornecedores de *hardware* e *software*;
- Periféricos escaláveis;
- *Software* livre de código aberto;
- Uso de ferramentas de computação distribuída disponível livremente com alterações mínimas;
- Retorno à comunidade do projeto e melhorias.

A Figura 35 mostra um exemplo de *cluster* construído com *PCs*.

Figura 35 – *Cluster* montado com *PCs*
Fonte: Infowester (2013).

2.2.2 Análise de Desempenho

De acordo com Pitanga (2008, p. 216), uma das principais técnicas para fornecer informações sobre o desempenho do *cluster* é a análise de desempenho. Uma das métricas para se medir esse desempenho é o *speedup* (ganho de velocidade) do *cluster*. O *speedup* mede a relação entre o tempo de execução do algoritmo em um computador serial e o tempo de execução desse mesmo algoritmo em um computador paralelo.

A expressão do *speedup* é dada por:

$$Speedup = \frac{\text{Tempo de execução em Computador Serial}}{\text{Tempo de execução em Computador Paralelo}} = \frac{T_S}{T_P}$$

O ideal seria que o *speedup* fosse linearmente proporcional à quantidade de nós computacionais do *cluster* – mas na prática isso não acontece – pois, quando se escreve um programa paralelo para ser executado no *cluster* há necessidade de se fazer a sincronização e troca de informações entre os nós processadores. Dependendo dessa quantidade de informações a serem trocadas ou sincronizadas, o projeto de paralelização de um algoritmo pode ser inviabilizado. Isso é chamado *Custo de Paralelização*.

O ideal é que um aumento no número de nós processadores que compõem o *cluster* traga igual aumento de desempenho (PITANGA, 2008, p. 219).

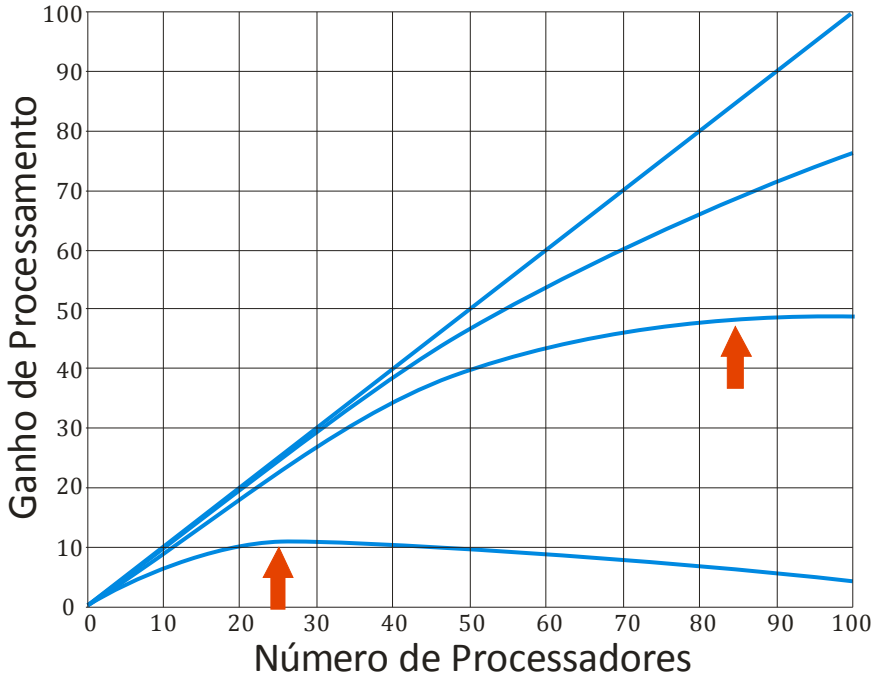


Figura 36 – Custo de Paralelização e pontos de saturação
Fonte: Pitanga (2008, p. 220).

A Figura 36 apresenta quatro situações de cálculo de *speedup*. A primeira curva mostra uma situação ideal hipotética onde o ganho de desempenho é linear ao número de processadores. A segunda curva mostra uma situação mais próxima da

prática, onde ao se aumentar a quantidade de processadores, a curva de desempenho não a acompanha na mesma proporção. As terceira e quarta curvas mostram casos onde pontos de saturação ocorrem quando os números de nós processadores do *cluster* chegam a 85 e 25, respectivamente, ou seja, a partir dessas quantidades de nós processadores torna-se inútil adicionar mais nós ao *cluster* para a realização da tarefa paralela. No caso da quarta curva, ao adicionar mais nós processadores, há uma piora de desempenho.

A *Lei de Amdahl* diz que ao se utilizar P processadores para realizar uma tarefa que tem uma fração serial de execução f, o *speedup* líquido previsto é dado por:

$$Speedup = \frac{1}{f + \frac{1 - f}{P}}$$

Ou de forma mais geral, isso mostra o *speedup* resultante ao se aplicar qualquer melhoramento de desempenho por um fator de P a apenas uma parte de uma determinada carga de trabalho, que no caso, é a parte paralela do algoritmo.

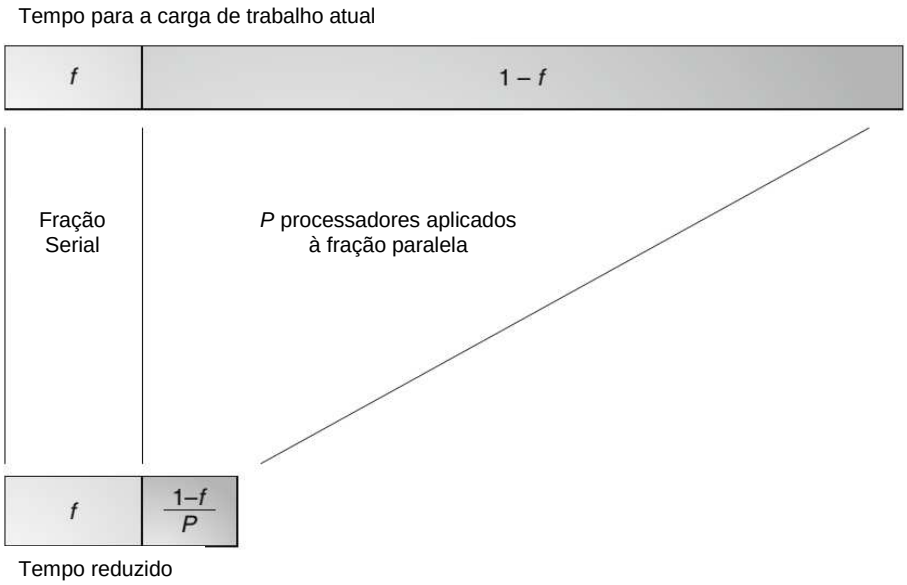


Figura 37 – Representação gráfica da Lei de Amdahl
Fonte: Padua (2011, p. 53).

A Figura 37 explica graficamente a expressão da Lei de Amdahl. O tempo necessário para resolver a carga de trabalho atual é de uma unidade (barra

superior). A parte da carga de trabalho que é serial (*f*) não é afetada pela paralelização. O modelo assume que o tempo restante $1 - f$ pode ser paralelizado perfeitamente. Então, isso leva somente $1/P$ de tempo a mais que o processador serial. A relação entre a barra superior e barra inferior é então de $1/(f + (1 - f)/P)$.

Outra medida de interesse na avaliação de desempenho é a *Eficiência Paralela*. De acordo com Mattos (2008, p. 24):

> Somente um sistema paralelo ideal contendo p processadores pode obter um fator de aceleração igual a p. Na prática, o comportamento ideal não é atingido porque enquanto executa um algoritmo paralelo, o processador não pode dedicar cem por cento do seu tempo para o processamento do algoritmo. Assim, a eficiência é uma medida da fração do tempo em que o processador é utilmente empregado; ela é definida como a razão entre o fator de aceleração e o número de processadores. Em um sistema paralelo ideal, o fator de aceleração é igual a p e a eficiência é igual a um. Na prática, o fator de aceleração é menos que p e a eficiência está entre zero e um, dependendo do grau de eficácia com que o processador é utilizado.

A Eficiência Paralela é definida matematicamente por

$$E = \frac{Speedup}{P}$$

onde P é o número de nós processadores do *cluster*. Um programa cujo *speedup* cresça linearmente com relação ao número de nós processadores ($Speedup = P$) tem uma Eficiência Paralela igual a 1 (BOSTON UNIVERSITY, 2013).

Pelo enunciado da Lei de Amdahl, f representa a fração do código que não pode ser paralelizado. A fração restante 1 - f é paralelizável. Idealmente, se o código paralelizado é linearmente proporcional ao número P de nós do *cluster*, então o tempo de execução se reduz a $(1 - f)/P$ e consequentemente

$$T_P = \left(f + \frac{1 - f}{P}\right) \times T_S$$

A razão de *Speedup* e a Eficiência Paralela podem ser usados para (BOSTON UNIVERSITY, 2013):

- Fornecer uma estimativa de quão bem o desempenho de execução de um código vai melhorar se o mesmo for paralelizado. Por exemplo, se $f = 0{,}1$ o *speedup* prevê um aumento de velocidade de no máximo 10 vezes. Por outro lado, um código que é 50% paralelizável irá na melhor das hipóteses sofrer um aumento de velocidade de fator 2. Neste último exemplo, um aumento potencial de velocidade de apenas um fator de 2 pode não ser atraente para iniciar um esforço de paralelização de código - especialmente se for necessário uma boa dose de esforço para paralelizar o código;
- Gerar um gráfico de tempo *versus* número de nós processadores para entender o comportamento do código paralelizado;
- Ver como a eficiência paralela tende para o ponto retorno decrescente. Com essas informações, é possível determinar para um problema de tamanho fixo, qual é o número ideal de nós processadores a serem utilizados.

A Figura 38 mostra exemplos de gráficos de Razão de *speedup* e Eficiência Paralela.

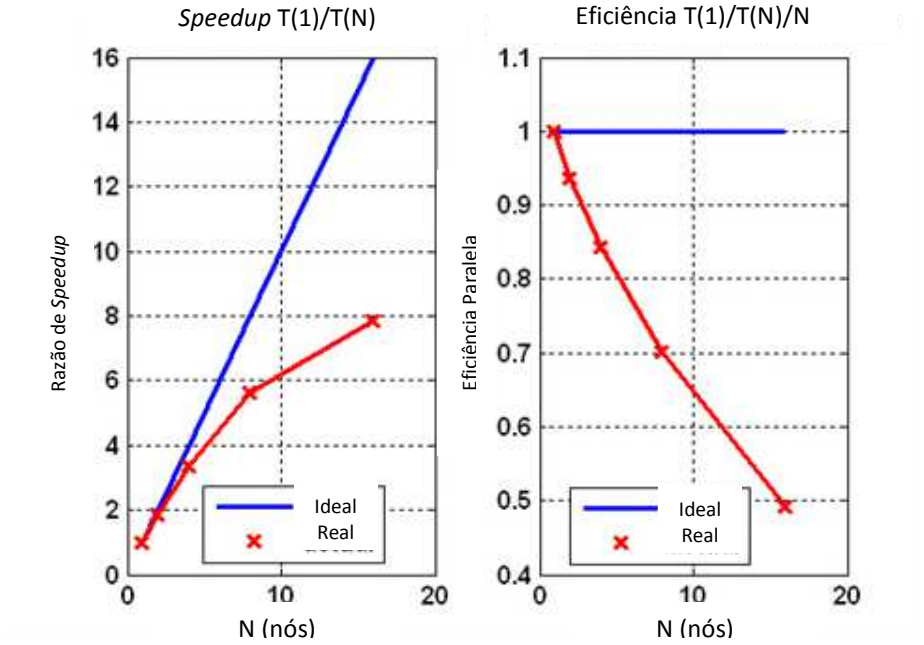


Figura 38 – Razão de *Speedup* e Eficiência Paralela
Fonte: Gebali (2011, p. 65, tradução nossa).

2.2.3 *Java*

A história do *Java* começa em 1991, quando um grupo chefiado por James Gosling e Patrick Naughton, funcionários da Sun Microsystems, desenvolveu uma linguagem chamada *Green* para ser utilizada em dispositivos de consumidores, tais como receptores inteligentes para televisão. Tal linguagem foi projetada desde o início para ser simples e neutra em relação à arquitetura, de modo que pudesse ser executada em vários *hardwares* diferentes. Nunca foi encontrado um cliente para essa tecnologia (HORSTMANN, 2004).

James Gosling batizou essa linguagem de *Oak* em homenagem a uma árvore de carvalho que ele via da sua janela na Sun. Quando a equipe da Sun visitou uma cafeteria local, o nome *Java* (cidade de origem de um tipo de café importado) foi adotado (DEITEL, 2005, p. 6).

Inicialmente o projeto *Green* passou por algumas dificuldades, já que o mercado de dispositivos inteligentes voltado para o consumidor final no início da década de 1990 não estava se desenvolvendo tão rapidamente como a Sun esperava. O projeto corria o risco de ser cancelado até que em 1993 a *World Wide Web* explodiu em popularidade e a equipe da Sun viu o potencial de utilizar o *Java* para adicionar conteúdo dinâmico (tais como interatividade e animações) às páginas da *Web* (DEITEL, 2005, p. 6).

Em 7 de Dezembro de 1995, a Microsoft assinou um acordo de intenções com a Sun para uma licença de fonte da tecnologia *Java*. Esse acordo foi muito importante, pois, ao integrar o *Java* o seu navegador *Explorer*, a Microsoft dotou o *Java* de uma enorme base de usuários do *Windows* já previamente estabelecida. Outro fato importante foi o reconhecimento da maior empresa de *software* do mundo de que a tecnologia da Sun é da maior qualidade e está muito adiantada no sentido de estabelecer o *Java* como padrão aberto *de facto* para a programação na *Internet* (NEWMAN, 1997, p. 5).

Com o *Java*, a Sun conseguiu estabelecer a primeira linguagem de programação que não estava amarrada a nenhum tipo de processador ou de sistema operacional,

já que os aplicativos escritos em *Java* são executados onde quer que seja, eliminando uma das maiores preocupações do usuário: as incompatibilidades entre sistemas operacionais e *hardwares* (NEWMAN, 1997, p. 5).

De acordo com Cornell e Horstmann (2005, p. 1 a 4), os autores da linguagem *Java* escreveram um manifesto que explica os objetivos e os sucessos na elaboração da linguagem. Publicaram também um resumo menor, organizado em torno de 11 tópicos chave, a saber:

a) Simples: a sintaxe do *Java* é uma versão melhorada da sintaxe do *C++*, porém, sem ponteiros, arquivos *header*, estruturas, uniões, sobrecargas de operadores, etc. Além disso, os programas em *Java* tendem a ficar com tamanhos pequenos;

b) Orientada a Objetos: a programação orientada a objetos tem se firmando como um paradigma de valor nos últimos 30 anos e seria inconcebível que uma linguagem de programação moderna como o *Java* não provesse tais mecanismos. As funções orientadas a objeto do *Java* são comparáveis àquelas do *C++*;

c) Distribuída: as funções de rede do *Java* são poderosas e fáceis de usar. O mecanismo de invocação remota de métodos permite a comunicação entre objetos distribuídos;

d) Robusta: *Java* foi projetada para a produção de programas que devem ser confiáveis em todos os sentidos. O compilador *Java* é capaz de detectar muitos problemas que em outras linguagens apareceriam somente em tempo de execução;

e) Segura: *Java* foi elaborada para ser utilizada em ambientes de rede/distribuídos. Sendo assim, colocou-se muita ênfase na segurança. O *Java* torna extremamente difícil vencer os seus mecanismos de segurança, tais como tomar o controle da pilha de execução, corromper a memória fora

do seu próprio espaço de processamento e ler ou escrever arquivos sem permissão;

f) Neutra em relação à arquitetura: o compilador gera um formato de arquivos neutro em relação à arquitetura, pois o código compilado pode ser executado em vários processadores ou sistemas operacionais, desde que exista o sistema em tempo de execução do *Java*. Isso é possível porque o compilador *Java* consegue gerar instruções *bytecodes* que não tem relação com alguma arquitetura computacional específica;

g) Portável: ao contrário das linguagens *C* e *C++*, não existem aspectos da especificação que sejam dependentes da implementação, ou seja, os tamanhos dos tipos de dados primitivos são especificados, bem como o comportamento da aritmética deles. Além disso, desde as primeiras versões têm sido feitos esforços no sentido de uma padronização para interfaces gráficas;

h) Interpretada: a linguagem *Java* executa os *bytecodes* diretamente em qualquer máquina para qual o interpretador em tempo de execução tenha sido escrito;

i) Alto desempenho: em muitas plataformas existe outra forma de compilação chamada de compiladores *Just-in-time*. Tais compiladores compilam os códigos nativos uma vez e armazenam os seus resultados em *cache*, chamando-os novamente depois se necessário. Isso acelera de dez a vinte vezes a execução de códigos que são frequentemente usados. Apesar de ser mais lento do que um compilador de código nativo, o *JIT* será significativamente mais rápido do que um interpretador puro;

j) *Multithreaded*: a implementação de diversas linhas de execução (*threads*) pode se beneficiar de sistemas multiprocessados se o sistema operacional de base também o fizer. Isso possibilita maior responsividade interativa e comportamento em tempo real;

k) Dinâmica: em *Java*, as bibliotecas podem adicionar novos métodos e variáveis de instância sem qualquer efeito em seus clientes. Além disso, é fácil descobrir informações de tipos em tempo de execução. Isso é muito importante para sistemas que precisem analisar objetos em tempo de execução, como por exemplo, construtores de *GUI Java*, depuradores inteligentes, componentes inseríveis e bancos de dados de objetos.

2.2.4 Mecanismo de Passagem de Mensagens

Para Rauber e Rünger (2010, p. 197), o modelo de programação de passagem de mensagens é baseado na abstração de um computador paralelo com um espaço de endereçamento distribuído onde cada processador tem uma memória local com acesso exclusivo à mesma. Como não há uma memória global, a troca de dados deve ser realizada por meio da passagem de mensagens: para transferir dados da memória local do processador A para a memória local de um outro processador B, A deve enviar uma mensagem contendo os dados para B.

Diversos tipos de comunicações podem ser estabelecidos entre os nós processadores. Para Gebali (2011, p. 64), os principais tipos que podem ser identificados são:

- Um para um (*Unicast*);
- Um para muitos (*Multicast*);
- Um para todos (*Broadcast*);
- *Gather;*
- *Reduce.*

Na Figura 39 são representados os diferentes tipos de comunicação entre os processadores: (a) Um para um, (b) Um para muitos, (c) Um para todos e (d) *Gather* e *Reduce*.

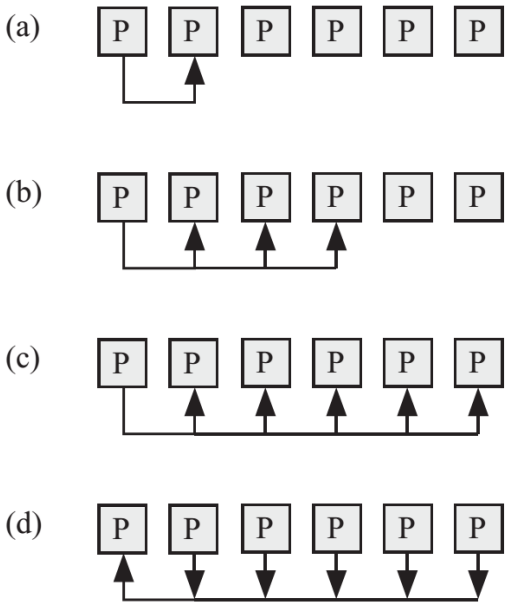


Figura 39 – Os diferentes tipos ou modos de comunicação entre processadores
Fonte: Gebali (2011, p. 65).

Segundo Gebali (2011, p. 66), o programador utiliza chamadas às funções `send()` e `recv()` de uma biblioteca. O `send(destino, mensagem)` especifica a identificação do processador ou processo destino e os dados a serem enviados. O `recv(destino, mensagem, tipo da mensagem)` especifica a identificação do processador ou processo de origem e o tipo de dado a ser recebido.

A *MPI* é a padronização de uma especificação de biblioteca de passagem de mensagens. *MPI* define a sintaxe e semântica das rotinas da biblioteca para os padrões de comunicação entre processadores. Linguagens como *C*, *C++*, *Fortran-77* e *Fortran-95* são suportadas (RAUBER; RÜNGER, 2010, p. 198).

Diversas implementações da biblioteca *MPI* foram desenvolvidas para a linguagem *Java*. Uma dessas implementações é a *MPJ Express*.

MPJ Express é uma biblioteca de passagem de mensagens de código aberto implementada em *Java* que permite que desenvolvedores de aplicações escrevam e executem aplicações paralelas para processadores multinúcleos e para computação em *clusters*/nuvem. É uma implementação de referência da *API mpiJava* 1.2, que é uma *API* que segue os padrões da especificação *MPI* (MPJ EXPRESS PROJECT, 2008).

2.3 GRAFOS

Segundo RUOHONEN (2013, p. 1, tradução nossa),

> Conceitualmente, um grafo é formado por vértices e arestas conectando os vértices. Formalmente, um grafo é um par de conjuntos (V, E), onde V é o conjunto de vértices e E é o conjunto de arestas, formado pelos pares de vértices. E é um multiconjunto, ou em outras palavras, os seus elementos podem ocorrer mais de uma vez, de modo que cada elemento tem uma multiplicidade. Frequentemente, os vértices são rotulados com letras $(a, b, c, \ldots$ ou $v_1, v_2, \ldots)$ ou números $(1, 2, \ldots)$.

A Figura 40 mostra um exemplo de grafo com os vértices identificados por números. Os conjuntos que representam o grafo abaixo são $V = \{1, 2, \ldots, 7\}$ e $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 4\}, \{5, 7\}\}$.

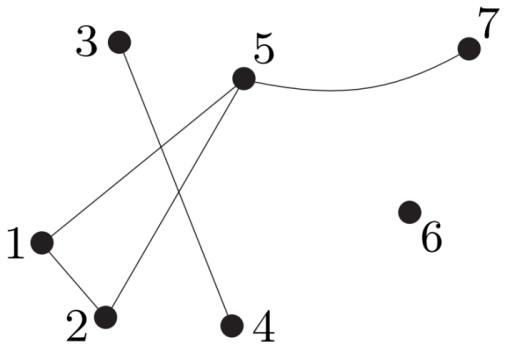


Figura 40 – Exemplo de um grafo com vértices numerados
Fonte: Diestel (2000, p. 2).

Um grafo com um conjunto de vértices V é dito um grafo em V. O número de vértices de um grafo G é conhecido como sua ordem e é denotado por $|G|$. O número de arestas é denotado por $\|G\|$. Grafos podem ser finitos ou infinitos de acordo com sua ordem. Um vértice v é incidente a uma aresta se $v \in e$, assim, e é uma aresta em v. Uma aresta $\{x, y\}$ é usualmente escrita como xy (ou yx). Dois vértices x, y de G são adjacentes ou vizinhos se xy é uma aresta de G (DIESTEL, 2000, p. 2).

De acordo com Diestel (2000, p. 5 e 148), o grau (ou valência) $d_G(v) = d(v)$ de um vértice v é o número $|E(v)|$ de arestas em v, que é igual ao número de vizinhos de v. O número $\delta(G) = \min\{d(v) | v \in V\}$ é o grau mínimo de G e o número $\Delta(G) = \max\{d(v) | v \in V\}$ é o grau máximo de G. O grau médio de G é o número dado pela seguinte equação:

$$d(G) = \frac{1}{|V|}\sum_{v \in V} d(v)$$

A densidade de um grafo G é dada pela relação abaixo:

$$densidade(G) = \frac{\|G\|}{\binom{|G|}{2}}$$

Uma das mais tradicionais aplicações de grafos é conhecida como o problema das Pontes de Königsberg. A cidade de Königsberg consistia de quatro áreas de terra separadas por ramificações do rio Pregel, sobre as quais havia sete pontes como mostra a Figura 41. O problema que os moradores da cidade propuseram para eles mesmos era andar pela cidade atravessando cada uma das sete pontes exatamente uma vez e retornar ao ponto de partida (HOPKINS; WILSON, 2004, p. 198).

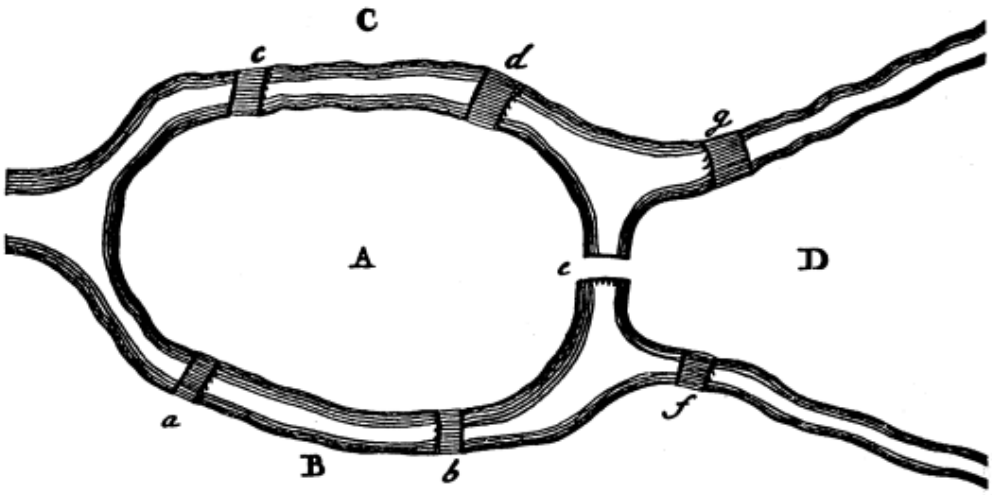


Figura 41 – Representação da cidade de Königsberg desenhado por Euler
Fonte: Hopkins e Wilson (2004, p. 200).

Uma forma proposta para se resolver o problema das Pontes de Königsberg foi representar a cidade por meio de um grafo. O problema então agora é encontrar um caminho nesse grafo que passe por cada uma das arestas somente uma vez. A Figura 42 mostra o grafo que representa o problema das Pontes de Königsberg.

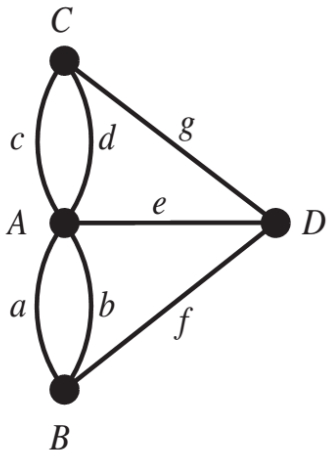
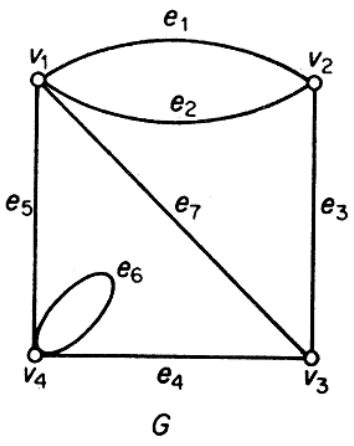


Figura 42 – O grafo de Königsberg
Fonte: Hopkins e Wilson (2004, p. 199).

2.3.1 Representação de Grafos utilizando *CSR*

Para qualquer grafo G existem duas matrizes de correspondência que o representam: a matriz de incidência e a matriz de adjacência. Ao se considerar os vértices de um grafo G como v_1, v_2, ..., v_v e suas arestas como e_1, e_2, ..., e_e, a matriz de incidência de G é a matriz $M(G) = [m_{ij}]$ de tamanho v x e, onde m_{ij} é o número de vezes (0, 1 ou 2) que o vértice v_i e a aresta e_j são incidentes. A matriz de adjacência do grafo G é a matriz $A(G) = [a_{ij}]$ de tamanho v x v, onde a_{ij} é o numero de arestas unindo v_i e v_j. A Figura 43 mostra um grafo G e suas representações utilizando as matrizes de incidência e de adjacência (BONDY; MURTY, 1982, p. 7).



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	1	0	0	1	0	1
v_2	1	1	1	0	0	0	0
v_3	0	0	1	1	0	0	1
v_4	0	0	0	1	1	2	0

M(G)

	v_1	v_2	v_3	v_4
v_1	0	2	1	1
v_2	2	0	1	0
v_3	1	1	0	1
v_4	1	0	1	1

A(G)

Figura 43 – Matrizes de representação de um grafo
Fonte: Bondy e Murty (1982, p. 7)

A matriz de adjacência do grafo G a ser particionado pode ser representada pelo formato de armazenamento de matrizes *CSR*.

O formato *CSR* é provavelmente o mais popular para armazenar matrizes esparsas. Uma matriz, ao ser representada pelo formato *CSR*, utiliza três vetores de dados com as seguintes funções (SAAD; 1995, p. 92):

- Um vetor AA que contém os valores a_{ij} não nulos armazenados linha por linha, da linha 1 ate a linha n. O comprimento do vetor AA é Nz, onde Nz é a quantidade de elementos não nulos da matriz;
- Um vetor de inteiros JA que contém os índices das colunas dos elementos a_{ij} assim como são armazenados na matriz AA. O comprimento do vetor JA também é Nz;
- Um vetor de inteiros IA que contém ponteiros para os inícios de cada linha nos vetores AA e JA. Dessa forma, o conteúdo de $IA(i)$ é a posição nos vetores AA e JA onde a i-ésima linha começa. O comprimento de IA é $n + 1$ com $IA(n + 1)$ contendo o número $IA(1) + Nz$, ou seja, o endereço na matriz A e no vetor JA do início de uma linha fictícia de número $n + 1$.

A Figura 44 mostra uma matriz A e sua representação correspondente utilizando o formato *CSR*, com seus três vetores de dados.

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Figura 44 – Uma matriz e sua representação no formato *CSR*
Fonte: Saad (1995, p. 92)

2.3.2 O Problema de Particionamento de Grafos

De acordo com Fjällström (1998, p. 1), o 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 $N_1, N_2, \ldots, N_k$, tais que:

- $\bigcup_{i=1}^{p} N_i = N$ e $N_i \cap N_j = 0$ para $i \neq j$, ou seja, todos os vértices do grafo original estejam distribuídos entre os subconjuntos criados e esses subconjuntos sejam disjuntos;
- $W(i) \cong W/p, i = 1, 2, \ldots, p$ onde $W(i)$ e W são as somas dos pesos dos nós em N_i e N respectivamente, o que significa que a soma dos pesos dos vértices dos subconjuntos encontrados seja aproximadamente igual ao peso de todos os vértices dividido pelo número de subconjuntos k;
- o *cut size*, que é a soma dos pesos das arestas entre os subconjuntos, seja minimizado.

O *cut size* de uma partição é definido como a quantidade de arestas cujos vértices estão em diferentes subconjuntos. 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 para $k > 2$ é encontrando bisseções de maneira recursiva (BUI; MOON, 1996, p. 1).

A Figura 45 mostra um exemplo de um grafo com 12 vértices particionado em 3 subconjuntos com um *cut size* total igual a 7.

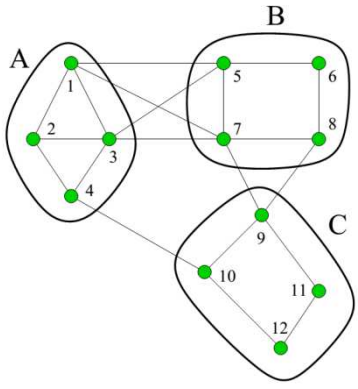


Figura 45 – Grafo particionado com *cut size* igual a 7
Fonte: Schloegel, Karypis e Kumar (2010, p. 4).

O PPG tem diversas aplicações práticas, tais como projeto de circuitos *VLSI*, fatoração de matrizes esparsas, roteamento e particionamento de malhas de

elementos finitos para aplicações de programação paralela (BUI; MOON, 1996; BONATTO; AMARAL, 2010). A Figura 46 apresenta um grafo particionado em 4 subconjuntos.

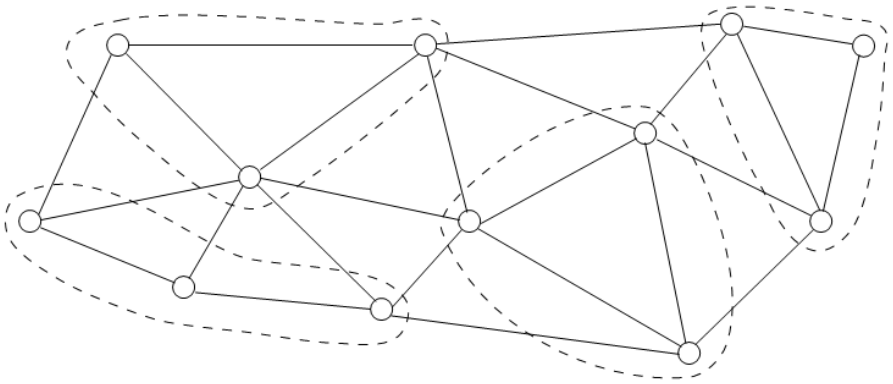


Figura 46 – Exemplo de um grafo particionado em quatro subconjuntos
Fonte: Fjällström (1998, p. 1).

2.3.3 Métodos de Particionamento de Grafos

Os PPG tendem a serem problemas *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, pois na prática, na impossibilidade de se encontrar soluções ótimas para grafos com muitos vértices, soluções boas podem ser aplicadas (BONATTO; AMARAL, 2010).

Dentre os principais métodos para a resolução do PPG, os principais são apresentados a seguir.

2.3.3.1 Métodos Geométricos

De acordo com Schloegel, Karypis e Kumar (2010, p. 5), 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 o de

minimizar métricas, como por exemplo, o número de vértices do grafo que são adjacentes a outros vértices não-locais (tamanho da fronteira).
Os algoritmos geométricos de particionamento de grafos são utilizados apenas se as coordenadas dos vértices do grafo estão disponíveis. Tais algoritmos tendem a ser mais rápidos que os métodos espectrais, mas retornam partições com piores cortes (KARYPIS; KUMAR, 1998, p. 360).

Um exemplo de método geométrico é o *Recursive Coordinate Bissection* (*RCB*). Esse método biparte o grafo ortogonalmente ao longo da maior dimensão em dois subgrafos de pesos quase iguais e aplica o mesmo princípio recursivamente aos subgrafos gerados (OU; RANKA, 1997, p. 7). Um exemplo de particionamento de grafos usando o RCB pode ser visto na Figura 47 (a) Grafo inicial; (b) partições do primeiro passo recursivo; (c) o grafo particionado após o *RCB*.

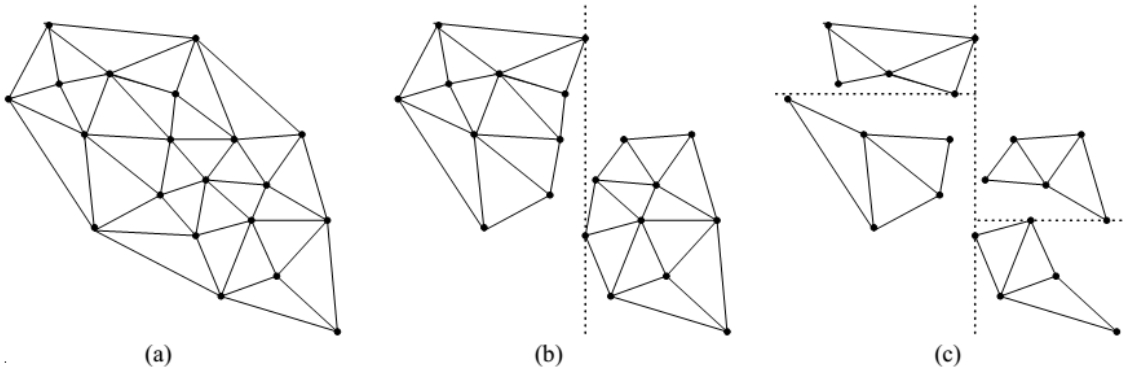


Figura 47 – Particionamento de um grafo em 4 partições utilizando *RCB*
Fonte: Ou e Ranka (1997, p. 10).

Outro método geométrico é o *Recursive Inertial Bissection* (*RIB*). Ao contrário do *RCB*, que biparte o grafo utilizando apenas eixos ortogonais, esse método biparte o grafo ao longo do eixo de maior dimensão (qualquer eixo) em dois subgrafos de pesos quase iguais e aplica o mesmo princípio recursivamente aos subgrafos gerados (OU; RANKA, 1997, p. 10). Um exemplo de particionamento de grafos usando o *RIB* pode ser visto na Figura 48: (a) Grafo inicial; (b) partições do primeiro passo recursivo; (c) o grafo particionado após o *RIB*. As linhas sólidas representam os eixos principais.

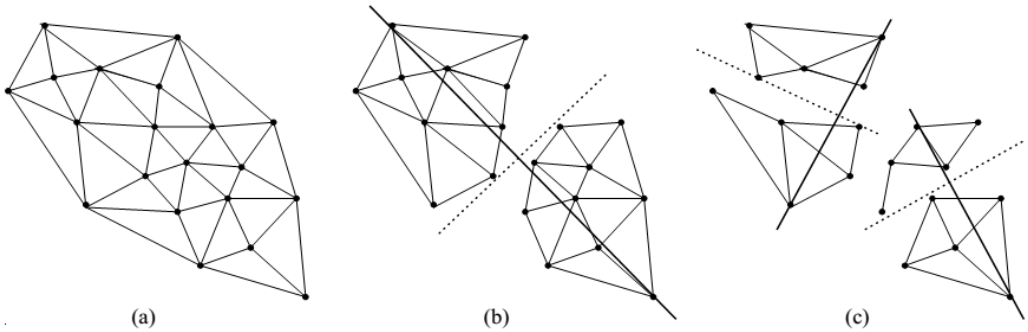


Figura 48 – Particionamento de um grafo em 4 partições utilizando *RIB*
Fonte: Ou e Ranka (1997, p. 11).

2.3.3.2 Métodos Espectrais

Métodos espectrais não particionam o grafo lidando com o mesmo propriamente dito, mas com sua representação matemática. Esses métodos resumem as propriedades estruturais dos grafos utilizando os autovetores da suas respectivas matrizes de adjacências ou matrizes Laplaciana. Um dos mais importantes atributos espectrais de um grafo é o vetor Fiedler, ou seja, o autovetor associado com o segundo menor autovalor da matriz Laplaciana (QIU; HANCOCK, 2006, p. 22).

Os grafos são formulados pela otimização de uma função quadrática discreta, porém, mesmo com essa representação, o problema é intratável. Os métodos espectrais relaxam essa representação discreta, transformando-a em uma função contínua. A minimização do problema relaxado é então resolvido pelo cálculo do segundo autovetor da matriz Laplaciana discreta do grafo (SCHLOEGEL; KARYPIS; KUMAR, 2001, p. 12).

A Figura 49 apresenta um exemplo de particionamento de um grafo utilizando o método espectral. Inicialmente, é apresentado o grafo de entrada. A matriz A é a matriz de adjacência correspondente a esse grafo. A matriz D é a matriz que contém os graus dos vértices, sendo que o elemento $D[i, i]$ contém o grau do vértice i. A matriz $L_G = A - D$. O vetor Fiedler da matriz L_G associa um valor com cada vértice, que representa a medida da distância (baseada na conectividade) entre os vértices. Os vértices são então ordenados de acordo com esses valores. Uma bisseção é obtida ao dividir essa lista pela metade.

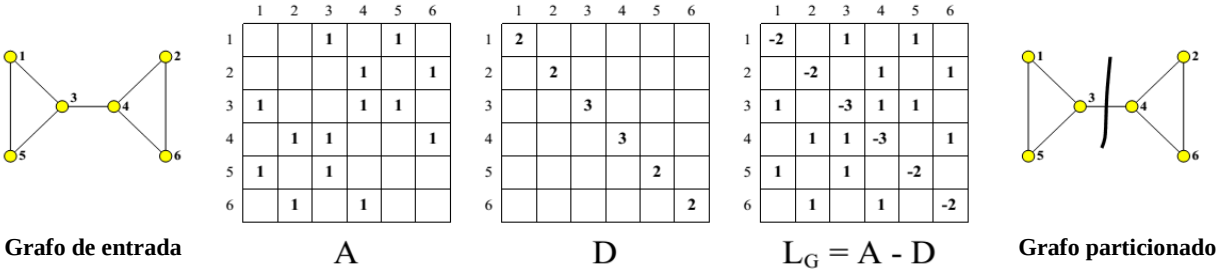


Figura 49 – Bisseção de um grafo utilizando o método espectral
Fonte: Schloegel, Karypis e Kumar (2001, p. 12, tradução nossa).

Segundo Karypis e Kumar (1998, p. 360), os métodos espectrais produzem boas partições para uma extensa classe de problemas e são utilizados de forma bem ampla. Por outro lado, demandam muito esforço computacional para o cálculo do vetor de Fiedler.

São exemplos de métodos espectrais o *Recursive Spectral Bisection* (*RSB*) e o *Multilevel Recursive Spectral Bisection* (*Multilevel RSB*).

2.3.3.3 Métodos Combinatórios

De acordo com Schloegel, Karypis e Kumar (2001, p. 9), os métodos combinatórios encontram partições baseados nas informações de adjacência do grafo e não nas coordenadas dos vértices. Tendem a produzir *cut sizes* menores, porém, são mais lentos do que os métodos geométricos e não são tão triviais de se paralelizar.

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 e o Fidducia e Mattheyeses (BENLIC; HAO, 2013).

Dada uma bisseção do grafo com dois conjuntos A e B de vértices, uma maneira de se refinar o corte da partição é encontrar dois novos conjuntos X e Y de mesmo tamanho que A e B respectivamente, tal que ao se trocar X por B e Y por A resulta na maior redução possível no *cut size*. Essas trocas podem ser realizadas um número

de vezes até que nenhuma troca mais resulte em algum melhoramento no corte (SCHLOEGEL; KARYPIS; KUMAR, 2001, p. 10).

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.

Segundo Benlic e Hao (2013) e Kenighan e Lin (1970), a heurística KL melhora uma bisseção inicial ao trocar dois conjuntos de seus vértices de iguais tamanhos. Seja (A, B) uma bisseção do grafo. Denota-se $g(a, b)$ a redução do *cut size* quando dois vértices a e b pertencentes respectivamente aos subconjuntos A e B são trocados. O ganho $g(a, b)$ é calculado por:

$$g(a, b) = g_a + g_b - 2\delta(a, b)$$

onde

$$\delta(a, b) = \begin{cases} 1 \text{ se } (a, b) \text{ são adjacentes} \\ 0, \text{ caso contrário.} \end{cases}$$

O algoritmo KL seleciona um par de vértices (a, b) para serem trocados, tais que os mesmos maximizem esse ganho $g(a, b)$. Vértices trocados entre partições cujo ganho g é máximo, diminuem o *cut size* da mesma. Uma vez que um par é selecionado, o mesmo não é mais considerado para futuras trocas. Os pares $(a_1, b_1), \cdots, (a_{n/2-1}, b_{n/2-1})$ são formados. O processo de troca de vértices é repetido até que nenhum melhoramento na bisseção seja obtido. A complexidade dessa heurística é $O(n^3)$.

Fiduccia e Mattheyses (1982) modificaram a heurística de bissecionamento KL sugerindo que se movesse apenas um vértice por vez em vez de se mover aos pares. Além disso, a heurística FM utiliza uma estrutura de dados chamada *bucket* para armazenar os ganhos dos vértices em ordem decrescente de ganho (e assim obter os vértices com maiores ganhos), o que reduz a complexidade do algoritmo para $O(|E|)$. Essa estrutura de dados também é utilizada no momento da atualização dos ganhos vértices que foram afetados pela troca de partição para outra, reduzindo em muito as buscas de ganho não necessárias.

O *bucket* é uma estrutura que guarda todos os vértices com um ganho g em uma lista na posição g do *bucket*. Para encontrar o vértice com o maior ganho, basta selecionar um vértice na posição não nula g do *bucket* com o maior ganho. A estrutura também mantém um vetor adicional de vértices onde cada elemento (vértice) aponta para o seu nó correspondente nas listas duplamente ligadas (BENLIC; HAO, 2013, p. 8).

A Figura 50 mostra um exemplo da estrutura *bucket* para duas partições, mostrando as listas duplamente encadeadas que guardam os vértices nas respectivas linhas correspondentes aos seus ganhos e logo abaixo, o vetor que aponta para cada elemento correspondente aos vértices nas listas.

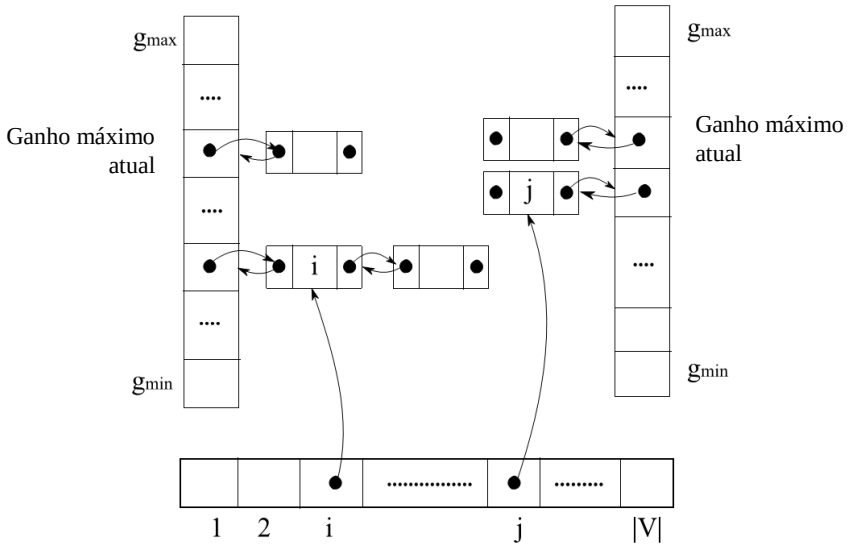


Figura 50 – Estrutura de dados *bucket* para uma bisseção de grafo
Fonte: Benlic e Hao (2013, p. 9, tradução nossa).

2.3.3.4 Métodos Multiníveis

Para Schloegel, Karypis e Kumar (2001, p. 14) e Karypis e Kumar (1998, p. 362) 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 (como por exemplo, heurística FM ou KL) é aplicado a cada nível do grafo à medida que o mesmo é expandido para que os cortes sejam melhorados a cada etapa dessa expansão.

A Figura 51 apresenta as três fases do paradigma de particionamento por meio dos métodos multiníveis. Durante a fase da contração, grafos cada vez menores são construídos. Na fase de particionamento inicial, uma bisseção é construída sobre esse grafo contraído de menor tamanho. Durante a fase de expansão, as bisseções são sucessivamente refinadas por meio de alguma heurística de melhoramento.

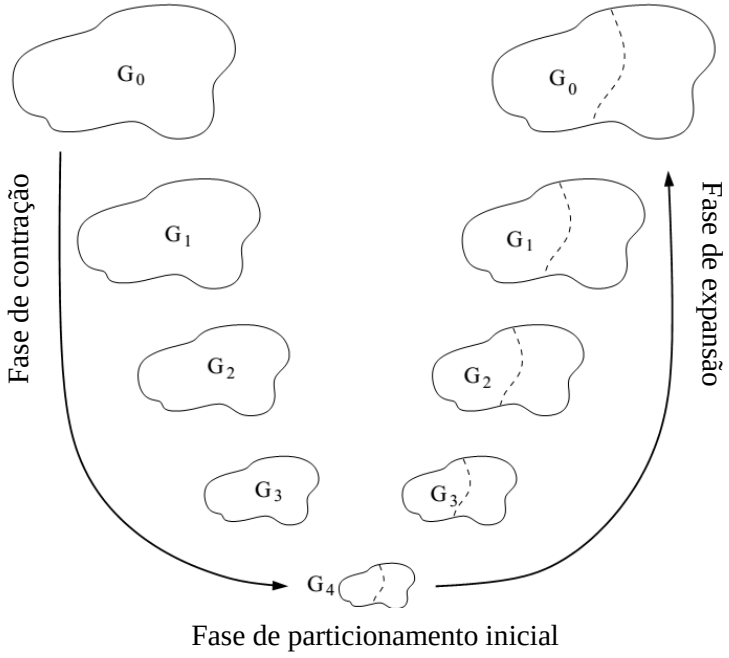


Figura 51 – As três fases do particionamento multinível
Fonte: Schloegel, Karypis e Kumar (2001, p. 13, tradução nossa).

A Figura 52 mostra como o método multínivel escapa de um mínimo local quando o mesmo particiona o grafo após a contração. Com o grafo contraído, agora é possível escapar do mínimo local ao trocar os vértices A e B por meio de alguma heurística de melhoramento.

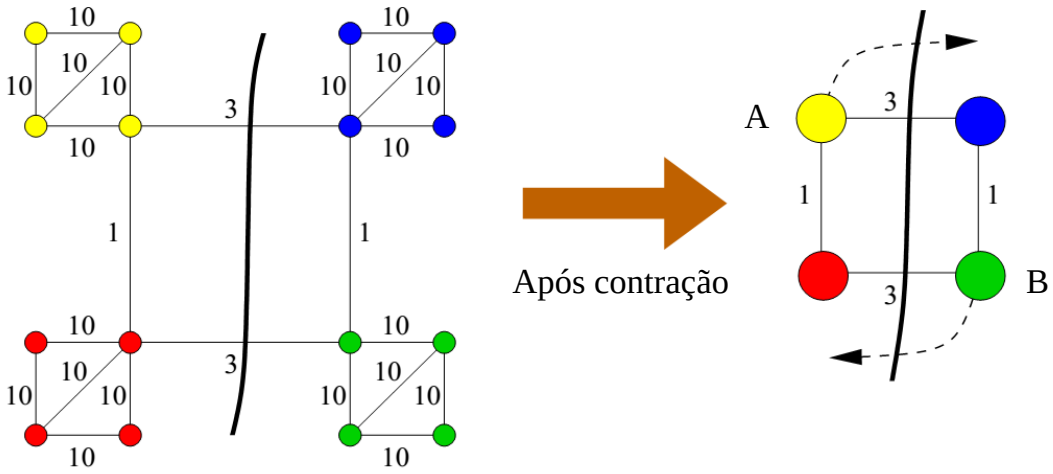


Figura 52 – Escapando de um mínimo local pelo método multinível
Fonte: Schloegel, Karypis e Kumar (2001, p. 15, tradução nossa).

2.3.3.5 Metaheurísticas

Metaheurísticas podem ser definidas como:

> [...] um conjunto de conceitos que são 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 adaptados para um problema específico (METAHEURISTICS NETWORK, 2013, tradução nossa).

Diversas metaheurísticas foram adaptadas para o particionamento de grafos, tais como Algoritmos Genéticos, *Simulated Annealing* (Recozimento Simulado), *Tabu Search* (Busca Tabu).

Segundo Fjällström (1998, p. 7), Bui e Moon (1996) propuseram um método para bisseção de grafos utilizando Algoritmos Genéticos. O algoritmo inicialmente classifica os nós seguindo a ordem de visitação da busca em largura começando em um nó aleatório do grafo. Uma população inicial de bisseções balanceadas é então gerada. O algoritmo seleciona um par delas para formar uma nova geração. Cada seleção é escolhida com uma probabilidade que depende do seu *cut size*: quanto menor o *cut size*, melhor a chance de a bisseção ser escolhida. Quando um par de pais é selecionado, uma descendência (bisseção balanceada) é criada. O algoritmo tenta então diminuir o *cut size* da prole aplicando uma variação do algoritmo KL.

Uma solução da população é escolhida para ser substituída pela descendência. Bui e Moon (1996) compararam experimentalmente seu algoritmo com os algoritmos KL e *SA*. Concluíram que o seu algoritmo produz partições de qualidade comparável ou melhor do que os outros dois algoritmos.

Bui e Moon compararam experimentalmente seu algoritmo com os algoritmos KL e *SA*. Concluíram que o algoritmo produz partições de qualidade comparável ou melhor do que os outros dois.

De acordo com Thao *et al*. (1992, p. 160, tradução nossa):

> Recozimento Simulado é uma abordagem de otimização estocástica baseada no recozimento físico. Ele tenta evitar ser preso em um ótimo local ao aceitar tanto movimentos “bons” quanto “ruins” no início das iterações e gradualmente diminuindo a probabilidade de aceitar movimentos “ruins”. Apesar de teoricamente o Recozimento Simulado poder encontrar um ótimo global, se diminuirmos lentamente a probabilidade acima em tempo exponencial, seu desempenho numa janela de tempo prática depende intimamente de um parâmetro conhecido com “cronograma de refrigeração”.

Johnson *et al*. (1989) propuseram uma maneira de bissecionar um grafo utilizando o Recozimento Simulado. Apesar dessa proposta obter bons resultados – até melhores do que a heurística KL em alguns grafos – ela não se mostrou a melhor solução para grafos esparsos ou que tenham alguma estrutura local (FJÄLLSTRÖM, 1998, p. 6).

Rolland, Pirkul e Glover (1996) adaptaram com sucesso a técnica combinatória Busca Tabu para o bissecionamento de grafos. De acordo com Fjällström (1998, p. 6, tradução nossa):

> [...] Começando com uma bisseção balanceada selecionada aleatoriamente, eles [os autores] iterativamente procuram melhores bisseções. Durante cada iteração, um nó é selecionado e movido da parte a qual ele pertence atualmente para outra parte. Após ser movido, o nó é mantido numa chamada *lista tabu* por um determinado número de iterações. Se o movimento resulta em uma bisseção balanceada com um *cut size* menor do que qualquer outra bisseção balanceada previamente encontrada, a bisseção é marcada como a melhor bisseção atual. Se uma bisseção melhor não for encontrada depois de um número fixado de movimentos consecutivos, o algoritmo incrementa um *fator de desequilíbrio*. Esse fator limita a diferença de cardinalidade entre as duas partes na bisseção. Inicialmente, este fator é zero e é zerado em determinados intervalos. A escolha de qual nó será movido depende de vários fatores. Movimentos que resultariam em uma diferença de cardinalidade maior do que é permitida

> pelo fator de desequilíbrio atual são proibidos. Além disso, movimentos que envolvam os nós da lista tabu são permitidos apenas se isso conduzir a uma melhoria sobre a bisseção atual. De todos os movimentos permitidos, o algoritmo faz o movimento que leva à maior redução no *cut size*.

Rolland, Pirkul e Glover (1996) compararam experimentalmente seu algoritmo com os algoritmo KL e SA e comprovaram que o mesmo é melhor tanto em qualidade de soluções quanto em tempo de execução (FJÄLLSTRÖM, 1998, p. 6).

2.3.3.6 Heurísticas Propostas

O objetivo dessa seção é explicar sucintamente as heurísticas combinatórias de particionamento de grafos propostas por Bonatto (2010), já que a proposta principal do trabalho é a paralelização das mesmas. Para mais detalhes, recomenda-se a leitura do trabalho de Bonatto (2010).

O referido autor propôs quatro heurísticas para PPG-*k*, dentre as quais as três primeiras heurísticas 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 ideia básica é uma heurística de crescimento: construir uma partição do grafo acumulando vértices em subconjuntos, vértice a vértice, exceto para a heurística que implementa a bisseção recursiva. Inicialmente existe apenas um subconjunto contendo todos os vértices do grafo. Um outro subconjunto p qualquer é construído selecionando aleatoriamente um vértice do grafo que não pertença a nenhum outro subconjunto já formado. A escolha dos vértices que comporão o próximo subconjunto é feita a partir dos vértices que fazem parte da fronteira desse novo subconjunto que está sendo construído (BONATTO, 2010, p. 47).

A Figura 53 ilustra a construção de um subconjunto. Inicialmente, existe apenas um subconjunto de índice $p = 1$. Um vértice v inicial é selecionado aleatoriamente para

compor o subconjunto de índice $p = 2$. O crescimento segue pela fronteira do vértice v. Após o subconjunto ser construído, uma heurística de melhoramento mostrada em linha tracejada refina o corte parcial do grafo. Por fim, o subconjunto $p = 2$ é formado (BONATTO, 2010, p. 48).

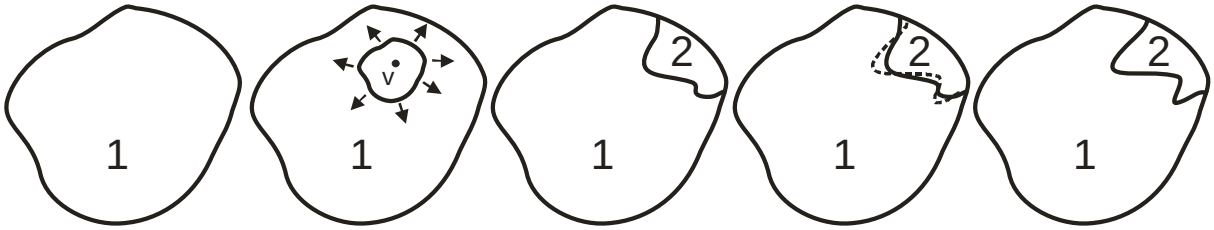


Figura 53 – Exemplo de construção de um subconjunto
Fonte: Bonatto (2010, p. 48).

O algortimo *k-BalancedPartition* é responsável por manter o balanceamento das partições do grafo após o particionamento. Ele também utilizada os procedimentos *CreateSubset_p*, responsável pela construção dos subconjuntos e *Improvement*, responsável pelo refinamento da partição sendo formada. O algoritmo *k-BalancedPartition* é mostrado na Figura 54.

```
Algoritmo k-BalancedPartition
1.   n ← |V|
2.   LISTA: subsetp ← ∅
3.   LISTA: frontier ← ∅
4.   edgecut ← 0
5.   p ← 2
6.   resto ← n mod k
7.   enquanto p ≤ k faca
8.        se resto > 0 então
9.             maxCard ← ⌈n / k⌉
10.            resto ← resto - 1
11.       senão
12.            maxCard ← n / k
13.       fim-se
14.       edgecut ← edgecut + CreateSubset_p(G, p, maxCard, subsetp, frontier)
15.       mincut ← edgecut
16.       Improvement(G, subsetp, frontier, p)
17.       edgecut ← mincut
18.  fim-enquanto
```

Figura 54 – Algoritmo *k-BalancedPartition*
Fonte: Bonatto (2010, p. 49).

a) Heurística 1

De acordo com Bonatto (2010, p. 49), cada subconjunto é construído adicionando-se vértices até que o tamanho do conjunto p seja atingido. A cada iteração do método *CreateSubset_p* 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 v no subconjunto p, o corte de arestas cut do grafo é atualizado pela expressão $cut = cut - g(v)$, onde cut é o corte de arestas atual do grafo e $g(v)$ é o ganho obtido ao inserir o vértice v no subconjunto em crescimento, ou seja, representa o quanto o corte do grafo diminuirá com o movimento do vértice. O ganho $g(v)$ é dado pela equação $g(v) = E(v) - I(v)$, onde $E(v)$ é a quantidade de vértices adjacentes à v que estão no subconjunto em expansão e $I(v)$ a quantidade de vértices adjacentes que estão no mesmo subconjunto que v, ou seja, o subconjunto de índice 1. Assim, se $g(v) > 0$, pela expressão $cut = cut - g(v)$, temos que o corte do grafo irá diminuir, portanto, o ganho de inserir o vértice é positivo. Contudo, se $g(v) < 0$, então o corte do grafo irá aumentar (BONATTO, 2010, p. 50).

O pseudocódigo do algoritmo *CreateSubset_p* utilizando a Heurística 1 é apresentado na Figura 55.

```
Algoritmo CreateSubset_p
1.    seed ← random vertex ∈ subset1
2.    Inserir seed em subsetp
3.    Remover seed de subset1
4.    edgecut ← valor absoluto de g(seed)
5.    frontier ← { v ∈ subset1 / E(seed, v) ∈ E }
6.    enquanto | subsetp | < maxCard faça
7.        se frontier = ∅ então
8.            vertex ← random vertex ∈ subset1
9.            Inserir vertex em frontier
10.       fim-se
11.       vertex ← random vertex ∈ frontier
12.       Remover vertex de frontier
13.       Inserir vertex em subsetp
14.       Remover vertex de subset1
15.       edgecut ← edgecut – g(vertex)
16.       se Adjacent(vertex) ≠ ∅ então
17.           para todo v ∈ Adjacent(vertex) faça
18.               se v ∈ subset1 então
19.                   se v ∉ frontier então
20.                       Inserir v em frontier
21.                   fim-se
22.           fim-para todo
23.       fim-se
24.   fim-enquanto
25.   retorna edgecut
```

Figura 55 – Algoritmo *CreateSubset_p* utilizando a Heurística 1
Fonte: Bonatto (2010, p. 50).

b) Heurística 2

Na descrição de Bonatto (2010, p. 49), na Heurística 1 os vértices são adicionados ao novo conjunto que está sendo construído sem critério algum, apenas de forma aleatória. Na Heurística 2, a principal diferença é que agora cada vértice pertencente à fronteira tem seu ganho $g(v)$ calculado e armazenado em ordem decrescente em função dos ganhos dos vértices da mesma. A cada passo de execução, o vértice com maior ganho será inserido no conjunto em expansão. Quando esse vértice é inserido no subconjunto, os ganhos de todos os vértices adjacentes que estão na fronteira precisam ser atualizados e os ganhos que ainda não estão na fronteira precisam ser inseridos.

A Figura 56 mostra esse procedimento sendo executado e a Figura 57 apresenta o pseudocódigo da rotina *CreateSubset_p* executando a Heurística 2.

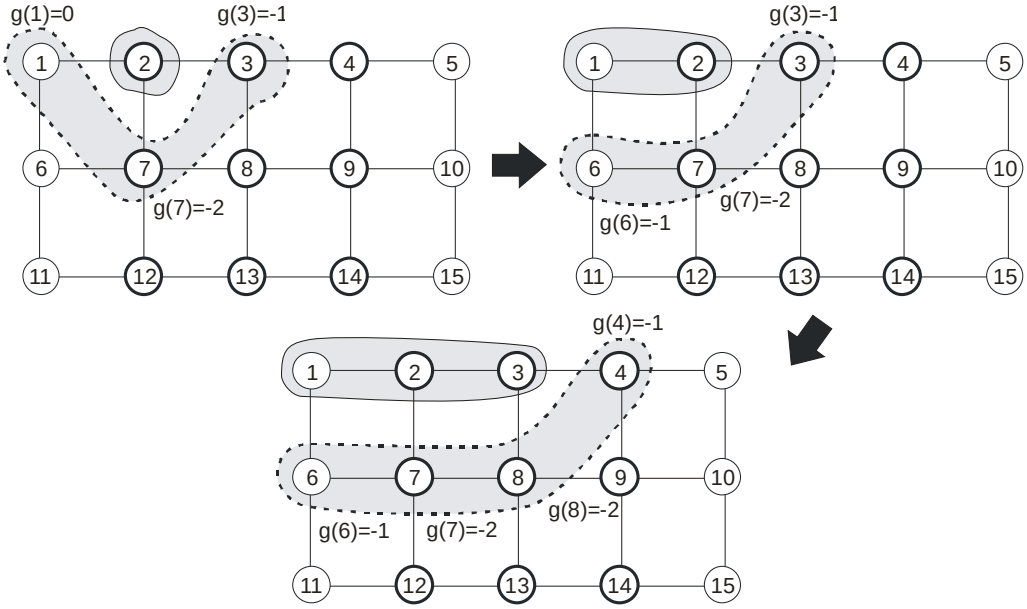


Figura 56 – Exemplo da construção de um subconjunto com três vértices
Fonte: Bonatto (2010, p. 52).

```
Algoritmo CreateSubset_p
1.    seed ← random vertex ∈ subset1
2.    Inserir seed em subsetp
3.    Remover seed de subset1
4.    edgecut ← valor absoluto de g(seed)
5.    frontier ← { v ∈ subset1 / E(seed, v) ∈ E }
6.    enquanto | subsetp | < maxCard faça
7.        se frontier = ∅ então
8.            vertex ← random vertex ∈ subset1
9.            Inserir ordenado vertex em frontier
10.       fim-se
11.       vertex ← vertex ∈ frontier com maior ganho
12.       Remover vertex de frontier
13.       Inserir vertex em subsetp
14.       Remover vertex de subset1
15.       edgecut ← edgecut – g(vertex)
16.       se Adjacent(vertex) ≠ ∅ então
17.           para todo v ∈ Adjacent(vertex) faça
18.               se v ∈ subset1 então
19.                   se v ∈ frontier então
20.                       Atualizar frontier
21.                   senão
22.                       Inserir ordenado v em frontier
23.                   fim-se
24.               senão
25.                   se v ∈ subsetp entao
26.                       Atualizar subsetp
27.               fim-se
28.           fim-para todo
29.       fim-se
30.   fim-enquanto
31.   retorna edgecut
```

Figura 57 – Algoritmo *CreateSubset_p* utilizando a Heurística 2
Fonte: Bonatto (2010, p. 52).

c) Heurística 3

Assim como na Heurística 2, a Heurística 3 calcula e armazena os ganhos $g(v)$ de cada vértice em ordem decrescente. A diferença entre elas é que a Heurística 3, em vez de tomar 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). O responsável por definir o tamanho da LCR é o parâmetro *alfa*, definido no intervalo $[0, 1]$ (BONATTO, 2010, p. 53).

> Para um problema de minimização, como o de minimizar o corte de arestas do grafo, seja v_{min} o vértice correspondente à seleção gulosa, isto é, com maior ganho, e v_{max} o vértice que implica no maior aumento no tamanho do corte de arestas. A LCR então compreenderá todos os vértices contidos no intervalo $[v_{min}, \alpha(v_{max} - v_{min}) + v_{min}]$ que ainda não fazem parte da solução, isto é, os vértices que estão na fronteira e são candidatos a serem adicionados ao subconjunto em expansão. Observe que $\alpha = 0$ implica em uma escolha puramente gulosa e, portanto, o algoritmo se comporta exatamente igual à Heurística 2. Por outro lado, $\alpha = 1$ implica em uma escolha aleatória, já que a LCR conterá todos os vértices possíveis, definidos pelo intervalo $[v_{min}, v_{max}]$, fazendo com que o algoritmo se comporte igual à Heurística 1. De fato, o parâmetro α controla a qualidade dos vértices da LCR (BONATTO, 2010, p. 54).

A Figura 58 apresenta o pseudocódigo da rotina *CreateSubset_p* executando a Heurística 3.

```
Algoritmo CreateSubset_p
26.   Ler valor de α
27.   subset1 ← { v ∈ V / Subset(v) = 1}
28.   seed ← random vertex ∈ subset1
29.   Inserir seed em subsetp
30.   Remover seed de subset1
31.   edgecut ← g(seed)
32.   frontier ← { v ∈ subset1 / E(seed, v) ∈ E }
33.   enquanto | subsetp | < maxCard faça
34.       se frontier = ∅ então
35.           vertex ← random vertex ∈ subset1
36.           Inserir vertex em frontier
37.       fim-se
38.       LCR = [ 0, α * size(frontier) ]
39.       vertex ← random vertex ∈ frontier in range LCR
40.       Remover vertex de frontier
41.       Inserir vertex em subsetp
42.       Remover vertex de subset1
43.       edgecut ← edgecut – g(vertex)
44.       se Adjacent(vertex) ≠ ∅ então
45.           para todo v ∈ Adjacent(vertex) faça
46.               se v ∈ subset1 então
47.                   se v ∈ frontier então
48.                       Atualizar frontier
49.                   senão
50.                       Inserir v em frontier
51.                   fim-se
52.               senão
53.                   se v ∈ subsetp entao
54.                       Atualizar subsetp
55.               fim-se
56.           fim-para todo
57.       fim-se
58.   fim-enquanto
59.   retorna edgecut
```

Figura 58 – Algoritmo *CreateSubset_p* utilizando a Heurística 3
Fonte: Bonatto (2010, p. 52)

d) Heurística 4

A Heurística 4 é uma aplicação da Heurística 3 de maneira recursiva. Essa quarta heurística forma uma *k*-partição do grafo por meio da aplicação de bisseções recursivas. O grafo inicialmente é bi-partido, depois o método de melhoramento é chamado para refinar a bisseção formada. 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 (BONATTO, 2010, p. 55).

A Figura 59 mostra a construção da bisseção utilizando a Heurística 4. Primeiro, um vértice aleatório é escolhido. Uma fronteira é formada (linha tracejada). O algoritmo segue adicionando vértices ao subconjunto enquanto não atingiu metade dos

vértices do grafo. Crescido o subconjunto até a metade dos vértices livres, a fronteira é usada para refinar a partição formada, melhorando assim o seu corte.

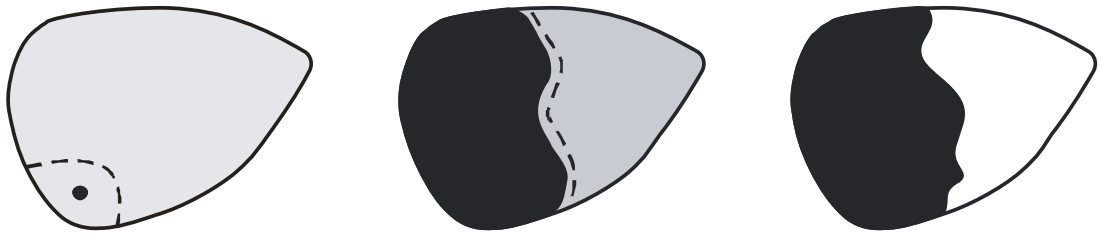


Figura 59 – Construção de uma bisseção utilizando a Heurística 4
Fonte: Bonatto (2010, p. 55)

O pseudocódigo principal da Heurística 4, responsável por conduzir o particionamento do grafo pode ser visto na Figura 60. O procedimento *Bisection* é o responsável por construir uma bisseção.

```
Algoritmo RecursiveBisection (graph, subSetL, h)
19.  h ←h+1
20.  Bisection(graph, subSetL, subSetR);
21.  se subSetL.size() > ⌈n/k⌉ então
22.        RecursiveBisection(graph, subSetL);
23.        RecursiveBisection(graph, subSetR);
24.  fim-se
```

Figura 60 – Pseudocódigo do procedimento *RecursiveBisection*
Fonte: Bonatto (2010, p. 56)

O procedimento recebe como entrada um subconjunto chamado *subSetL* que inicialmente contém todos os vértices do grafo (|*subSetL*| = |*V*|), o qual será dividido, e um subconjunto *subSetR* vazio, que posteriormente conterá a metade dos vértices do subconjunto inicial *subSetL*. O procedimento *Bisection* funciona de maneira similar ao procedimento *CreateSubSet_p* descrito nas heurísticas anteriores, com a diferença que o procedimento *CreateSubSet_p* produz uma lista contendo os vértices do subconjunto formado e uma outra lista contendo os vértices da fronteira com esse subconjunto gerado e que fazem parte do outro subconjunto, sendo ambas usadas pela rotina de melhoramento. Já o algoritmo *Bisection* produz duas listas (*subSetL* e *subSetR*) contendo os vértices da bisseção formada e já refinada, ou seja, a rotina de melhoramento é executada internamente (BONATTO, 2010, p. 56).

A Figura 61 apresenta um exemplo de particionamento por bisseção recursiva.

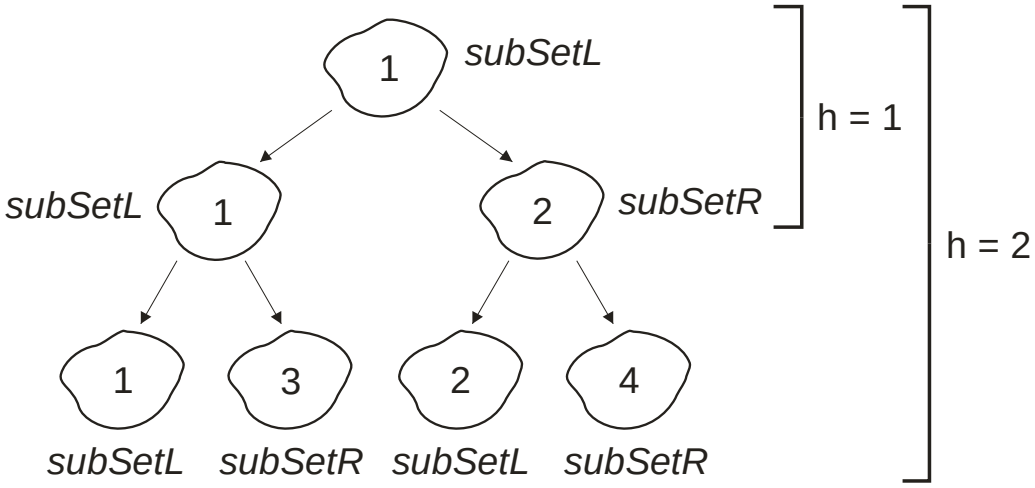


Figura 61 – Uma 4-partição via bisseção recursiva
Fonte: Bonatto (2010, p. 56).

Bonatto (2010, p. 56) explica que:

> O método utiliza a altura da árvore para definir os índices dos subconjuntos formados. Por exemplo, um subconjunto de índice *p*, após sofrer uma bisseção, gera dois subconjuntos, o subconjunto *subSetL*, de índice *p* e o subconjunto *subSetR*, de índice $p + 2^{h-1}$, sendo *h* a altura corrente do *subSetR* na árvore.

A Heurística 4 também é *multistart* e o seu pseudocódigo é dado pela Figura 62.

```
Algoritmo heurística 4
10.  alfa ← {αi , i = 1,...,10}
11.  para i ← 1 até MAX_ITERATIONS passo 1 faça
60.    subsetL ← { v ∈ V }
12.    α ← random α ∈ alfa
13.    RecursiveBisection ( graph, subSetL, α)
14.    se mincut < bestcut então
15.          bestcut ← mincut
16.          bestsolution ← partition
17.    fim-se
       mincut ←bestcut
       partition ← bestsolution
18.  fim-para
```

Figura 62 – Pseudocódigo da Heurística 4
Fonte: Bonatto (2010, p. 57).

e) Método de Melhoramento

O método de melhoramento é utilizado por todas as heurísticas propostas para refinar o corte parcial do grafo. A sub-rotina *Improvement* 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 (BONATTO, 2010, p. 57).

> [...] Inicialmente, todos os vértices podem ser movidos (os do subconjunto formado e os da fronteira). Então, o vértice com maior ganho de um subconjunto é movido para outro subconjunto e marcado. Assim um vértice trocado não poderá mais ser movido na mesma iteração. De forma análoga, um vértice de maior ganho é selecionado no outro subconjunto e movido. Cada movimento é seguido por outro em direção oposta. O algoritmo segue estes passos sucessivamente até que não existam mais vértices a serem movidos ou até que todos os vértices do subconjunto de menor cardinalidade já tenham sido movidos. Se alguma troca melhorou o corte de arestas do grafo, então o algoritmo mantém todas as trocas realizadas até a que resultou em menor corte e desfaz todas as trocas posteriores. A nova partição gerada é usada como partição inicial para o próximo passo do algoritmo e todos os vértices são novamente desmarcados. O algoritmo termina quando, durante um passo, nenhuma troca diminui o corte do grafo. (BONATTO, 2010, p. 57).

O pseudocódigo da sub-rotina *Improvement* é apresentado na Figura 63.

```
Algoritmo Improvement
1.    melhorou ← verdadeiro
2.    bestcut  ← ∞
3.    t ← 0
4.    enquanto melhorou faça
5.        melhorou ← falso
6.        para todo vertex ∈ subSetp faça
7.            Lock[vertex] ← unlock
8.        fim-para-todo
9.        para todo vertex ∈ frontier faça
10.           Lock[vertex] ← unlock
11.       fim-para-todo
12.       bestswap ← -1
13.       enquanto Size(subSetp) > 0 e Size(frontier) > 0 faça
14.           v_s ← vertex ∈{subSetp / Lock[vertex] = unlock e g(vertex) é máximo
15.           Lock[v_s] ← lock
16.           v_t ← vertex ∈{frontier / Lock[vertex] = unlock e g(vertex) é máximo.
17.           Lock[v_t] ← lock
18.           cut ← cut – g(v_s) – g(v_t)
19.           Pair[t] ← (v_s, v_t)
20.           Cut[t] ← cut
21.           se cut < bestcut então
22.               bestcut ←cut
23.               bestswap ← t
24.               melhorou = verdadeiro
25.           fim-se
26.       fim-enquanto
27.       se melhorou então
28.           para i←0 até bestswap passo 1 faça
29.               Troca Pair[i]
30.           fim-para
31.       fim-se
32.   fim-enquanto
```

Figura 63 – Pseudocódigo da sub-rotina de melhoramento *Improvement*
Fonte: Bonatto (2010, p. 58).

3 METODOLOGIA UTILIZADA

Neste capítulo é apresentada a metodologia de paralelização das heurísticas propostas por Bonatto (2010).

Os algoritmos de paralelização são apresentados, assim como as melhorias propostas sobre os algoritmos originais de Bonatto (2010).

Na parte final do capítulo são apresentados os tipos abstratos de dados que foram propostos para realizar a implementação das diversas estruturas de dados que as heurísticas necessitam para sua execução.

3.1 PARALELIZAÇÃO DAS HEURÍSTICAS

Na concepção deste trabalho, todas as heurísticas combinatórias para particionamento de grafos 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.

A Figura 64 mostra um exemplo de execução de um algoritmo em configuração *multistart*. O algoritmo encontra quatro partições diferentes para o mesmo grafo, cada uma delas também com um *cut size* distinto. A partição de menor *cut size* é escolhida e as demais são descartadas.

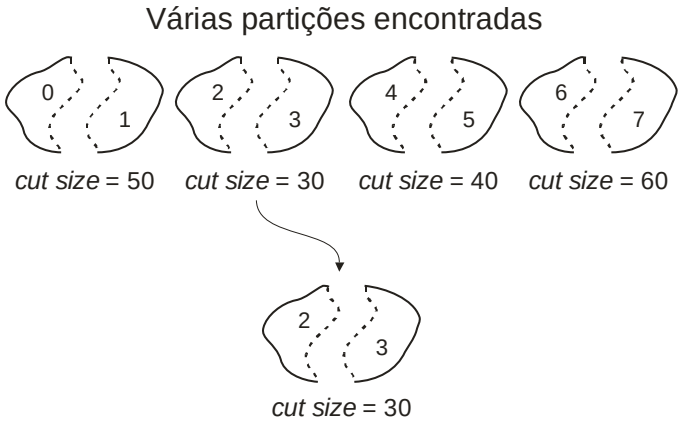


Figura 64 – Exemplo de execução de uma configuração *multistart*

Em uma implementação serial, para se obter um número elevado de repetições é necessário fazer com que o algoritmo principal de particionamento seja executado diversas vezes.

A proposta de paralelização das heurísticas combinatórias de Bonatto (2010), foco principal deste trabalho, baseia-se nessa ideia: ao invés de um processador serial realizar várias iterações do algoritmo de particionamento, diversos nós processadores de um *cluster* de computadores realizarão essas repetições, porém com a carga de trabalho dividida pelo número de nós processadores.

As Heurísticas 1, 2 e 3 de Bonnato (2010) 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*.

Considere que originalmente a implementação serial de particionamento seja executada com i iterações. Considere agora um *cluster* com p nós processadores, com $p \geq 2$. A implementação paralela proposta por este trabalho possibilita que cada um desses nós realize no máximo i/p iterações. O algoritmo paralelo proposto não prevê o balanceamento de carga com relação ao número de iterações de cada nó, portanto, considera-se que a quantidade de iterações i é múltipla do número de nós processadores p.

Caso a configuração dos nós processadores seja multinúcleo (mais de um *core* ou núcleo de processamento em cada encapsulamento de um processador), multiprocessada (mais de um processador por nó) ou a combinação de ambos, a proposta de paralelização desse trabalho também contempla a execução de várias *threads* (um processo dividindo-se em duas ou mais linhas concorrentes de execução). Isso faz com que a quantidade de iterações a ser executada por cada nó processador de forma paralela do *cluster* seja ainda maior.

Considere o exemplo apresentado na Figura 65. Em (a), um computador serial mononúcleo e monoprocessado executa i iterações de um determinado algoritmo de particionamento. Em um *cluster* com p nós processadores onde cada nó processador possui j *CPUs* com k núcleos por *CPU*, cada nó processador do *cluster* executará $\frac{i}{p \times j \times k}$ iterações.

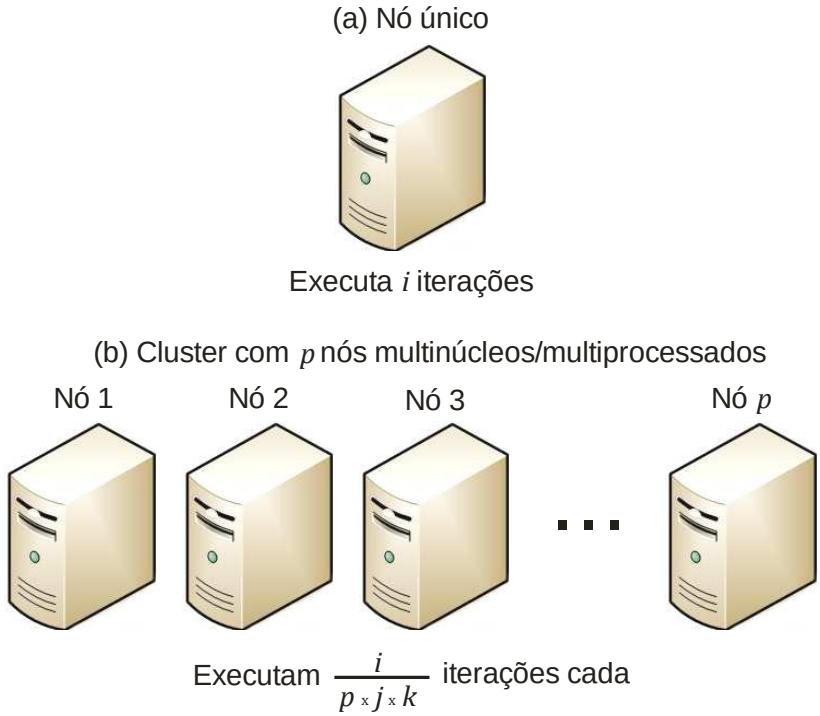


Figura 65 – Iterações executadas por computadores seriais e paralelos

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 pseudocódigo do algoritmo principal é apresentado na Figura 66.

```
1.   Algoritmo Particionador
2.
3.   g ← grafo lido do disco
4.   n ← quantidade de vértices do grafo
5.   k ← número de partições
6.   size ← quantidade de nós do cluster
7.   rank ← id do nó do cluster
8.   subsets ← new Subset[k]
9.   maxCard ← n
10.  hmax ← log2 k
11.  α ← valor de α de acordo com a distribuição escolhida
12.
13.  se (size = 1) então // caso a execução seja serial
14.     Executa Algoritmo Particionador Serial
15.  senão // caso a execução seja paralela
16.     Executa Algoritmo Particionador Paralelo
17.  fim-se
18.
19.  se (rank = root) então
20.     imprime os resultados finais obtidos
21.  fim-se
```

Figura 66 – Pseudocódigo do algoritmo principal do Particionador

No algoritmo principal, o grafo a ser particionado é lido do disco. O algoritmo também inicializa outras variáveis, tais como a quantidade de vértices do grafo, o número de partições no qual o grafo será particionado, a quantidade de nós do *cluster* e o *rank* de cada nó. Apesar de não ser mostrada no pseudocódigo, a escolha de qual heurística será utilizada para construir as bisseções recursivas também é informada, ou seja, dentro do mesmo algoritmo pode-se escolher qual das três heurísticas de criação dos subconjuntos será utilizada.

Outras variáveis de controle também são inicializadas, tais como o vetor de $subsets$, que conterá as partições com os vértices do grafo antes e depois de serem particionados e a variável $maxCard$ que a cada rodada de particionamento determinará a quantidade máxima de vértices que a nova partição possuirá.

A variável $hmax$ determinará a quantidade de rodadas que o particionador executará, que é uma função logarítmica de base dois da quantidade de partições que se deseja obter com o particionador. Por exemplo, caso o grafo seja dividido em 2 partições, o algoritmo será executado 1 vez. Caso o grafo seja dividido em 4 partições, o algoritmo será executado 2 vezes e assim por diante. Tal característica é decorrente do fato de o particionador construir as partições a partir de bisseções recursivas.

Outra variável de controle é o α, que é utilizado apenas pela Heurística 3 como um fator de qualidade do corte.

Caso a quantidade de nós que serão utilizados no *cluster* (*size*) seja igual a 1, o algoritmo particionador serial será executado.

O algoritmo particionador serial é apresentado na Figura 67. Esse algoritmo tem basicamente dois laços de execução que dependem da quantidade de partições na qual o grafo será particionado.

```
1.   Algoritmo Particionador Serial
2.
3.   para h ← 0 até menor hmax passo 1 faça
4.       maxCard ← calculaNovoMaxCard(maxCard)
5.
6.       para j ← 0 até menor 2^h passo 1 faça
7.           menorCorteNo ← ∞
8.           particaoInicial ← j
9.           particaoFinal ← j + 2^h
10.
11.          para i ← 0 até menor numeroThreadsNo passo 1 faça
1.              inicia uma nova ThreadGrasp(g, maxCard,
                subsets[particaoInicial], subsets[particaoFinal], frontier)
12.          fim-para
13.
14.          para i ← 0 até menor numeroThreadsNo passo 1 faça
15.              se (corte da thread i < menorCorteNo) então
16.                  menorCorteNo ← corte da thread i
17.                  subsets[particaoInicial] ← partição inicial da thread i
18.                 subsets[particaoFinal] ← partição final da thread i
19.                 frontier ← fronteira da thread i
20.             fim-se
21.          fim-para
22.
23.          corteTotal ← corteTotal + menorCorteNo
24.
25.      fim-para
26.  fim-para
```

Figura 67 – Pseudocódigo do algoritmo Particionador Serial

O laço exterior (variável h – linha 3) determina quantas vezes o algoritmo deve ser executado (rodadas) em função do logaritmo de base dois do número de partições, pois $hmax = \log_2 k$, com k sendo o número final de partições do grafo.

Para cada rodada h, o laço mais interior (variável j – linha 6) executa 2^h vezes. A variável j é utilizada para fazer o controle dos índices das partições que serão criadas (variáveis $particaoInicial$ e $particaoFinal$).

O laço inicializa a quantidade de *threads* determinada para a execução, como pode ser visto na Figura 68. Dentro de cada *thread* serão executadas as iterações de particionamento determinadas para a execução. Também dentro das *threads* serão executadas as chamadas à rotina de melhoramento do corte, caso isso tenha sido configurado antes da execução. Após a execução das iterações de cada *thread*, cada uma delas terá obtido o menor corte daquela rodada.

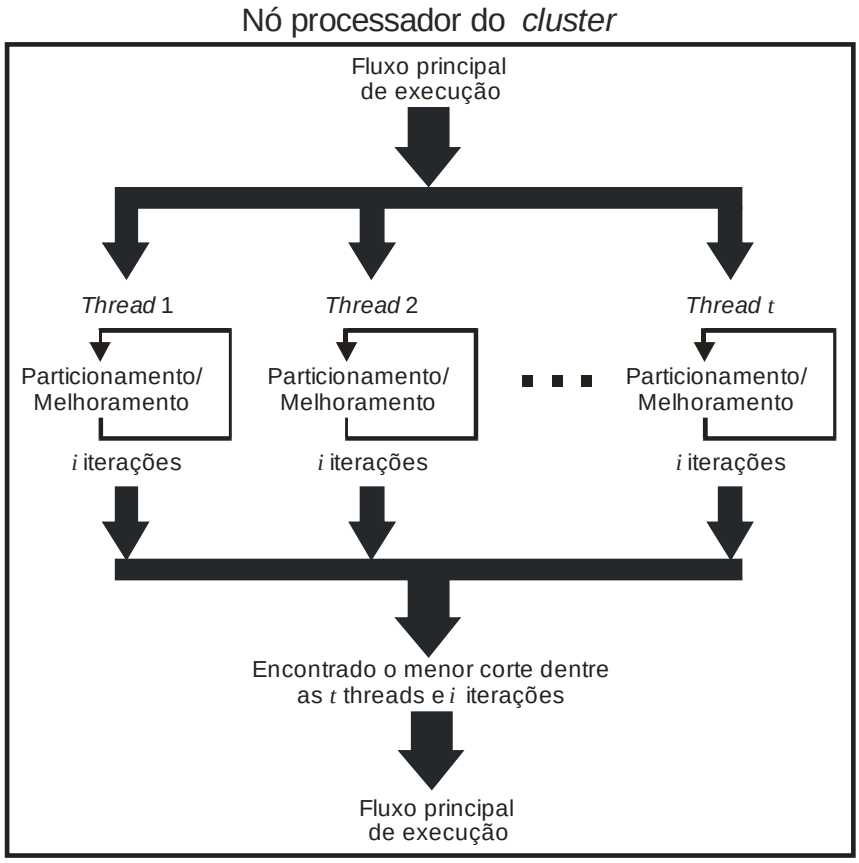


Figura 68 – *Threads* e iterações sendo executadas em um nó processador

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 na cardinalidade dos dois subconjuntos.

Ao valor total do corte 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 $k > 2$.

Um exemplo do particionador serial sendo executado com 8 partições é mostrado na Figura 69.

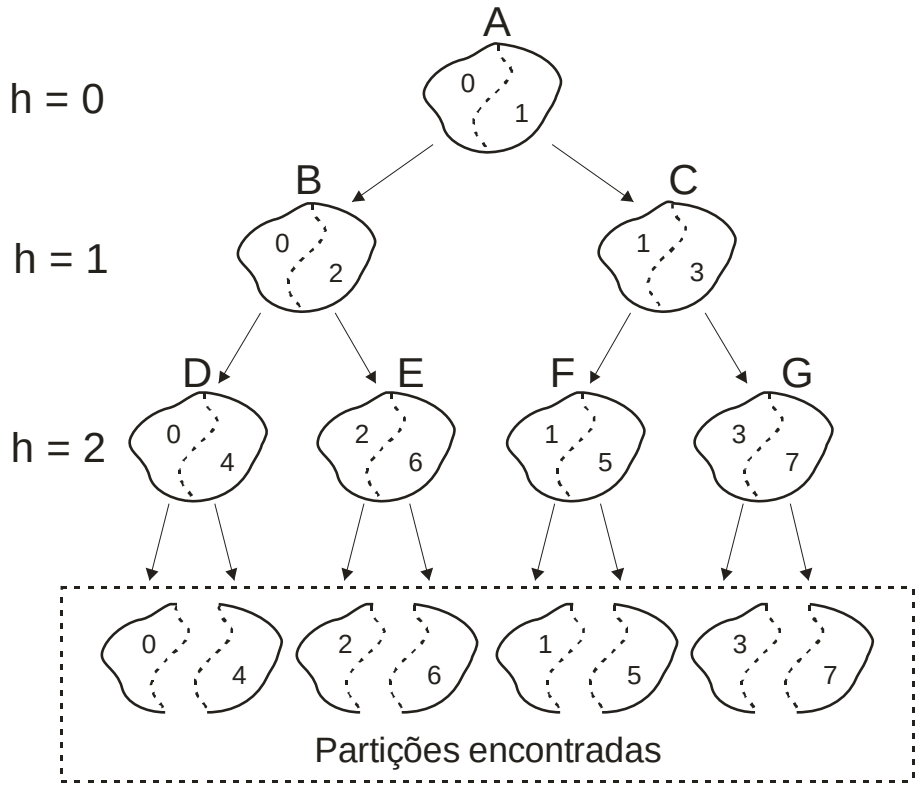


Figura 69 – Exemplo de um particionamento serial em 8 partições

Todos os grafos são representados como duas partições (*subsets*), uma contendo todos os vértices antes do particionamento e uma vazia. Para o exemplo acima, os valores das variáveis e partições são apresentados no Quadro 2.

	h=0	h=1		h=2			
	j=0 Grafo A	j=0 Grafo B	j=1 Grafo C	j=0 Grafo D	j=1 Grafo E	j=2 Grafo F	j=3 Grafo G
Part. Inicial	0	0	1	0	1	2	3
Part. Final	1*	2*	3*	4*	5*	6*	7*

Quadro 2 – Valores das variáveis durante a execução do particionamento serial

As partições assinaladas com asterisco (*) estão vazias antes do início do particionamento daquela rodada h. Após o término de cada rodada de particionamento, elas possuem $maxCard$ vértices oriundos da partição de origem para aquela rodada.

Caso a quantidade de nós que serão utilizados no *cluster* (*size*) seja maior do que um, o particionador paralelo será executado.

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

Em seguida, os nós do *cluster* retornam ao nó principal (*root*) os valores dos seus menores cortes e o *root* requisita as partições que resultaram nesses menores cortes aos nós que obtiveram tais cortes.

Dentro de cada nó processador do *cluster* as *threads* também serão inicializadas dependendo do número de núcleos e processadores daquele nó, possibilitando ainda mais iterações na tentativa de se obter o menor corte. Na execução das iterações dentro de cada nó processador do *cluster*, o algoritmo comporta-se exatamente como na versão serial.

Na implementação paralela, o particionador paralelo também executa $hmax$ rodadas no laço principal de execução, com $hmax = \log_2 k$, 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 $\frac{p}{2} - 1$ 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 $\frac{p}{2}$ 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 de particionamento paralelo executado em um *cluster* com $p = 8$ nós para se obter $k = 8$ partições é mostrado na Figura 70. Na rodada $h = 0$, o *root* envia para todos os nós as partições 0 e 1, sendo que a partição 0 possui inicialmente todos os vértices do grafo e a partição 1 ainda está vazia. Todos os nós em paralelo executam *t threads* e *i* iterações para encontrar o menor corte. Cada nó, ao final das iterações, vai enviar para o *root* o valor do seu menor corte obtido. O *root* vai determinar de qual nó veio o menor corte e vai solicitar somente para aquele nó que ele envie ao *root* suas partições 0 e 1 que resultaram nesse menor corte. No exemplo da Figura 70, foi o nó 2.

Na rodada seguinte ($h = 1$), o *root* envia para os nós com os *ranks* entre 0 e $\frac{p}{2} - 1$ (no caso, os nós de *rank* entre 0 e 3) as partições 0 (com os vértices obtidos do primeiro particionamento) e 2 (vazia). Da mesma forma, o *root* envia para os nós com os *ranks* entre $\frac{p}{2}$ e $p - 1$ as partições 1 (com os vértices obtidos do primeiro particionamento) e 3 (vazia). Agora metade dos nós do *cluster* calculam o menor corte entre as partições 0 e 2 e a outra metade dos nós calcula o corte entre as partições 1 e 3. Ao término da execução das iterações, todos os nós enviam seus respectivos menores cortes para o *root*. O *root* verifica o menor deles entre cada par de partições e solicita aos dois nós com os menores cortes que enviem sua partições (no exemplo, foram os nós 1 e 4) para que a rodada seguinte inicie-se até que o número de partições seja atingido.

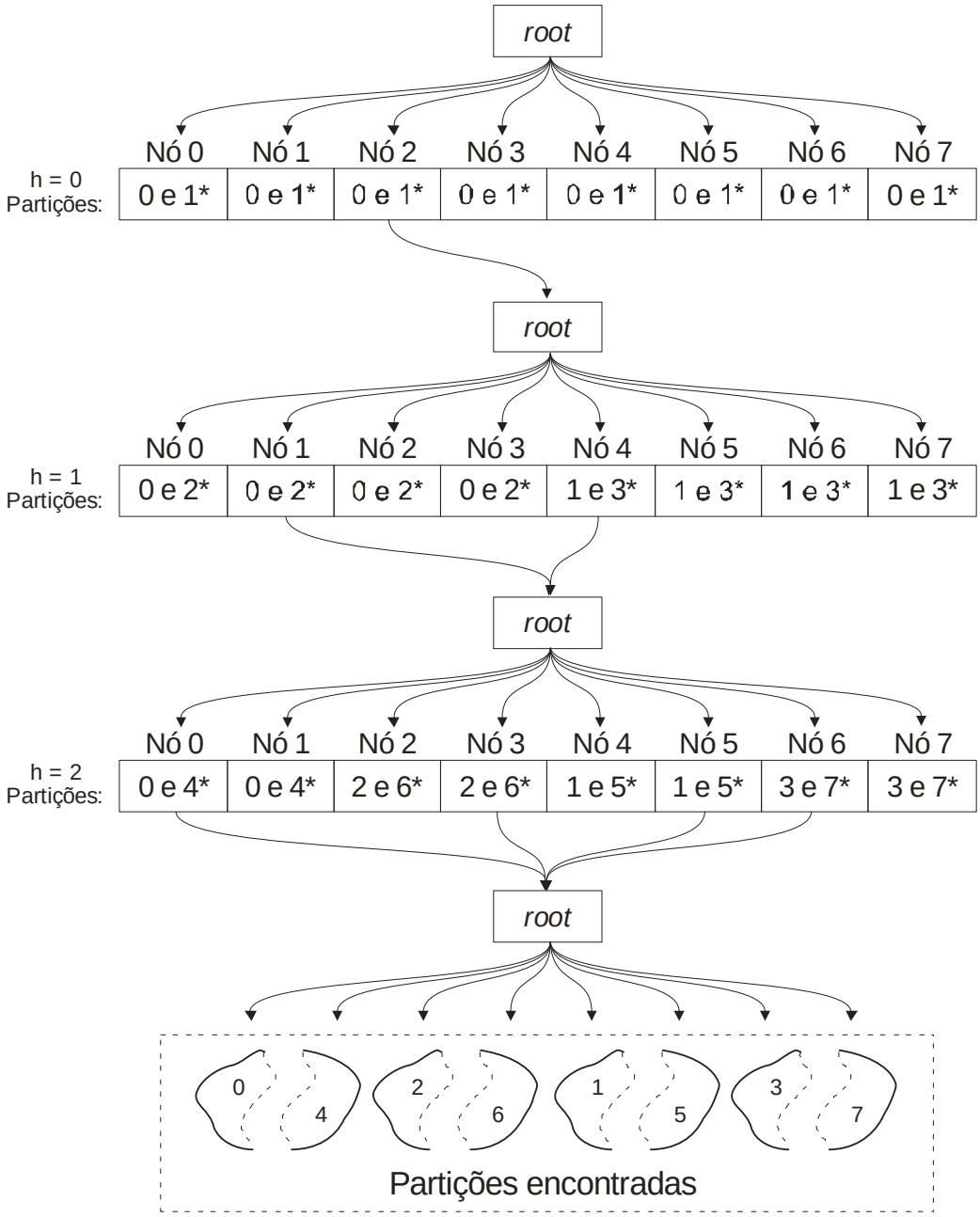


Figura 70 – Exemplo de um particionamento paralelo em 8 partições e 8 nós
As partições assinaladas com (*) estão vazias antes de iniciar-se a respectiva rodada de particionamento.

O algoritmo particionador paralelo é apresentado na Figura 71. Esse algoritmo tem basicamente dois laços de execução que dependem da quantidade de partições na qual o grafo será particionado.

```
1.   Algoritmo Particionador Paralelo
2.
3.   // Envio das partições iniciais para todos os nós
4.   para h ← 0 até menor hmax passo 1 faça
5.      se (rank = root) então
6.         para j ← 0 até menor 2^h passo 1 faça
7.            noInicial ← calculaNoInicial(j, size, 2^h)
8.            noFinal ← calculaNoFinal(j, size, 2^h)
9.
10.           para i ← noInicial até noFinal passo 1 faça
11.              root envia para nó i a id (j) e a partição inicial (subsets[j])
12.           fim-para
13.        fim-para
14.     senão
15.        demais nós recebem a id e a partição inicial
16.     fim-se
17.
18.     //Cálculo local do menor corte
19.     maxCard ← calculaNovoMaxCard(maxCard)
20.
21.     para j ← 0 até menor 2^h passo 1 faça
22.        noInicial ← calculaNoInicial(j, size, 2^h)
23.        noFinal ← calculaNoFinal(j, size, 2^h)
24.        particaoInicial ← j
25.        particaoFinal ← j + 2^h
26.
27.        se (rank >= noInicial e rank <= noFinal) então
28.              menorCorteNo ← ∞
29.
30.              para i ← 0 até menor numeroThreadsNo passo 1 faça
31.                 inicia uma nova ThreadGrasp(g, maxCard, subsets[particaoInicial],
                    subsets[particaoFinal], frontier)
32.              fim-para
33.
34.              para i ← 0 até menor numeroThreadsNo passo 1 faça
35.                 se (corte da thread i < menorCorteNo) então
36.                    menorCorteNo ← corte da thread i;
37.                    subsets[particaoInicial] ← partição inicial da thread i
38.                    subsets[particaoFinal] ← partição final da thread i
39.                    frontier ← frontier da thread i
40.                 fim-se
41.              fim-para
42.        fim-se
43.     fim-para
44.
45.     //O root recebe os menores cortes de todos os nós e verifica qual é o menor
46.     se (rank = root) então
47.        para j ← 0 até menor 2^h passo 1 faça
48.           noInicial ← calculaNoInicial(j, size, 2^h)
49.           noFinal ← calculaNoFinal(j, size, 2^h)
50.           particaoInicial ← j
51.           particaoFinal ← j + 2^h
52.
53.           para i ← noInicial até noFinal passo 1 faça
54.              root recebe os menores cortes de cada nó
55.
56.              se (menorCorteRecebido < menorCorteRodada[j]) então
57.                 menorCorteRodada[j] ← menorCorteRecebido
58.                 noMenorCorteRodada[j] ← i
59.              fim-se
60.           fim-para
61.        fim-para
62.     senão
63.        demais nós enviam seus menores cortes
64.     fim-se
65.
66.     // O root envia para todos os nós quem é o nó de menor corte
67.     se (rank = root) então
68.        para j ← 0 até menor 2^h passo 1 faça
69.           noInicial ← calculaNoInicial(j, size, 2^h)
70.           noFinal ← calculaNoFinal(j, size, 2^h)
71.           particaoInicial ← j
72.           particaoFinal ← j + 2^h
73.
74.           para i ← noInicial até noFinal passo 1 faça
75.              root envia para todos os nós a id do nó de menor
76.           fim-para
77.        fim-para
78.     senão
79.        demais nós recebem a id do nó de menor corte
80.     fim-se
81.
82.     // O root envia as ids das partições inicial e final para o nó de menor corte, que irá devolver ao
        root essas partições
```

```
83.      se (rank = root) então
84.         para j ← 0 até menor 2^h passo 1 faça
85.            se (noMenorCorteRodada[j] != root) então
86.               particaoInicial ← j
87.               particaoFinal ← j + 2^h
88.
89.               root envia para o nó de menor corte as ids das partições inicial e final que devem ser
                  devolvidas
90.            fim-se
91.         fim-para
92.      senão
93.         apenas o nó de menor corte recebe as ids das partições inicial e final que devem ser devolvidas
            ao root
94.      fim-se
95.
96.      // Os nós de menor corte enviam ao root agora suas partições
97.      se (rank = root) então
98.         root recebe as partições inicial e final do nó de menor corte
99.      senão se (rank = noMenorCorte)
100.        apenas o nó de menor corte envia as partições inicial e final ao root
101.     fim-se
102.
103. fim-para
```

Figura 71 – Pseudocódigo do algoritmo Particionador Paralelo

3.2 MELHORIAS IMPLEMENTADAS COM A PARALELIZAÇÃO DAS HEURÍSTICAS

Além da paralelização das heurísticas combinatórias de Bonatto (2010), que foi o principal foco desse trabalho, outras melhorias foram implementadas no decorrer do desenvolvimento do algoritmo paralelo de particionamento.

3.2.1 Divisão exclusiva das *seeds* entre os nós processadores

Todas as três heurísticas implementadas por Bonnato (2010) escolhem um vértice aleatório (*seed*) para começar a formar o novo subconjunto a ser criado. Diferentes escolhas de *seeds* podem conduzir a diferentes resultados nos cortes das partições. Como o algoritmo de particionamento é uma implementação paralela, mais de um nó pode escolher aleatoriamente a mesma *seed* para iniciar o particionamento. Para evitar que isso aconteça, aumentando a probabilidade de diferentes nós processadores obterem os mesmos cortes finais, uma divisão exclusiva dos vértices candidatos a *seeds* é feita entre os nós.

Para uma partição com n vértices com os vértices numerados de 1 a n e um *cluster* com $size$ nós processadores, sendo que cada nó processador é identificado por um número único chamado $rank$, com $0 \leq rank < size$, a divisão dos intervalos para o sorteio das *seeds* é feita conforme a seguir:

- Caso o resto da divisão de n por $size$ seja zero, os intervalos serão todos de tamanho igual a $\frac{n}{size}$;
- Caso contrário, os primeiros nós do *cluster* terão intervalos de tamanho $\left\lceil \frac{n}{size} \right\rceil$ e os $n - (n \; mod \; size)$ últimos nós terão intervalos de tamanhos $\left\lfloor \frac{n}{size} \right\rfloor$.

A Figura 72 apresenta o pseudocódigo para a função *getVerticeAleatorio* que obtém o valor inicial e final do intervalo de sorteio das *seeds* para cada um dos nós do *cluster*. São passados como parâmetros o *rank* que identifica o nó e o *size* do *cluster*.

```
1.   Função getVerticeAleatorio(rank, size)
2.
3.   n ← quantidade de vértices do grafo (v)
4.   valorInicial ← 0
5.   resto ← n mod size
6.   i ← 0
7.   enquanto (i < rank) faça
8.      se (resto > 0) então
9.         valorInicial ← valorInicial + ceil(n / size)
10.        resto ← resto - 1
11.     senão
12.        valorInicial ← valorInicial + n / size
13.     fim-se
14.     i ← i + 1
15.  fim-enquanto
16.
17.  se (resto > 0) então
18.     intervalo ← ceil(n / size)
19.  senão
20.     intervalo ←  n / size;
21.  fim-se
22.
23.  seed ← random(valorInicial, valorInicial + intervalo)
24.
25.  retorna (seed)
26.  fim-função
```

Figura 72 – Pseudocódigo da função *getVerticeAleatorio*

A Figura 73 mostra um exemplo da divisão das *seeds* de um grafo com 100 vértices em um *cluster* com 8 nós processadores. O nó 0 do *cluster* só pode obter uma *seed* entre os vértices 1 e 13, o nó 1 entre os vértices 14 e 26 e assim por diante.

Nó 0	Nó 1	Nó 2	Nó 3	Nó 4	Nó 5	Nó 6	Nó 7
1 a 13	14 a 26	27 a 39	40 a 52	53 a 64	65 a 76	77 a 88	89 a 100

Figura 73 – Exemplo da divisão das *seeds* em um *cluster*

3.2.2 Rotina de melhoramento do corte na iteração *versus* execução

De acordo com Bonnato (2010), após cada iteração do algoritmo de particionamento o método de melhoramento do corte é executado. Em situações nas quais as partições encontradas após a execução da rotina de particionamento resultam em cortes altos, a rotina de melhoramento pode ser um processo demorado para ser executado. Nesses casos, uma melhoria proposta foi a de poder selecionar quando o método de melhoramento deve ser executado – se a cada iteração ou após a execução de todas as iterações do particionamento.

Na primeira situação, os algoritmos implementados funcionam como a proposta de Bonnato (2010): a cada iteração que o particionamento é executado, o método de melhoramento é executado também. Na segunda situação, somente após a execução de todas as iterações é que o método de particionamento é executado. Quando o menor corte for encontrado, o método de melhoramento é executado apenas sobre as partições que resultaram nesse menor corte.

Durante os testes computacionais, notou-se que em algumas situações não necessariamente as partições que resultaram no menor corte na fase de particionamento resultaram também nas partições de menor corte na fase de melhoramento. Por isso, a opção de se escolher a aplicação da rotina de melhoramento na execução após terem sido executadas todas as iterações apenas com a rotina de particionamento pode não ser uma escolha ideal.

Tal melhoria foi proposta para resolver o problema de se esperar longos períodos de tempo quando a quantidade de iterações por nó era muito elevada e a rotina de melhoramento era executada após cada iteração. Para se encontrar um equilíbrio na relação tempo *versus* menor corte, a próxima melhoria foi proposta.

3.2.3 Cortes máximos para execução do melhoramento na iteração

Uma melhoria proposta para equilibrar a relação tempo *versus* menor corte ao se aplicar a rotina de melhoramento logo após a rotina de particionamento em cada

iteração foi a de estipular cortes máximos acima dos quais a rotina de melhoramento não será executada.

Para cada rodada de execuções, um corte máximo é estipulado. Cortes obtidos após a execução da rotina de particionamento que forem menores do que esses cortes estipulados terão a rotina de melhoramento executada logo após a iteração daquela. Caso os cortes obtidos pela rotina de particionamento sejam maiores do que os cortes máximos, não terão a rotina de melhoramento executada após cada iteração.

Durante os testes computacionais, verificou-se que partições que resultaram em cortes muito altos após a execução da rotina de particionamento não conseguiam resultar em menores cortes mesmo após a aplicação da rotina de melhoramento do mesmo. A ideia foi então estipular um valor considerado ideal para que os cortes obtidos no particionamento que fossem maiores do que esses valores fossem descartados pela rotina de melhoramento, já que ao executar a mesma sobre partições com cortes muito altos o tempo de execução seria excessivo.

Experimentações mostraram que cortes obtidos após a execução da rotina de particionamento que eram maiores do que a média dos cortes obtidos em todas as iterações da rotina de particionamento, mesmo sendo descartados pela rotina de melhoramento, produziam cortes baixos e não demandavam muito tempo de execução. Os cortes obtidos que eram maiores do que esse valor médio eram descartados pela rotina de melhoramento.

3.2.4 Modificação da rotina de melhoramento do corte

De acordo com pseudocódigo da rotina de melhoramento do corte proposta por Bonatto (2010), a mesma recebe duas listas de vértices, uma sendo o conjunto recém-formado de vértices e a outra a fronteira desse subconjunto com os demais. Inicialmente, todos os vértices podem ser movidos.

A rotina obtém o vértice de maior ganho $g(i)$ de um subconjunto e move-o para o outro conjunto. Esse vértice é marcado e não pode ser mais movido nessa iteração.

Da mesma forma, um vértice do outro subconjunto, também de maior ganho, é movido e marcado. Cada troca é seguida por outra troca em sentido oposto. Isso acontece até que não existam mais vértices a serem movidos ou até que todos os vértices do subconjunto com o menor número de vértices tenham sido movidos. Se alguma das trocas melhorou o *cut size* do grafo, a rotina mantém todas as trocas até essa que resultou no menor corte e desfaz as trocas posteriores.

Ao mover um vértice de um subconjunto para o outro, os ganhos dos vértices adjacentes a esse vértice movido alterar-se-ão. O pseudocódigo de Bonatto (2010) não contempla essa funcionalidade, visto que a lista ordenada pelos ganhos dos vértices de forma decrescente é utilizada como parâmetro para a escolha dos próximos vértices a serem movidos, porém, ela não tem atualizados os ganhos dos vértices que ainda não foram movidos.

Uma melhoria proposta foi a de que, assim que um vértice é movido para o outro subconjunto, os ganhos dos vértices adjacentes a esse vértice que ainda não foram movidos tenham os seus ganhos atualizados e a lista de vértices organizada pela ordem decrescente dos ganhos seja também atualizada. Dessa forma, a rotina de melhoramento escolherá o próximo vértice a ser movido para o outro conjunto baseada em um ganho que reflete a nova situação do conjunto de vértices. Na rotina de melhoramento original, isso não era considerado no momento de escolher o próximo vértice a ser movido, gerando melhorias inferiores nos cortes das partições.

3.2.5 Seleção da heurística

O algoritmo paralelo de particionamento foi implementado com uma série de configurações que podem ser definidas antes da execução do mesmo por meio de um arquivo de configurações (Apêndice B).

Em todas as situações de particionamento, as partições são obtidas por meio de bisseções recursivas do grafo. Durante a criação dessas bisseções, a heurística que será utilizada para construí-las (1, 2 ou 3) pode ser selecionada via arquivo de

configuração, permitindo uma comparação imediata da aplicação de mais de uma heurística de criação de partições em um mesmo grafo.

3.2.6 Estratégia de variação do alfa

De acordo com Bonatto (2010), o parâmetro α é utilizado para definir a qualidade dos ganhos dos vértices que fazem parte da LCR na Heurística 3. Foram propostas três estratégias de variação do parâmetro α, a saber: Fixo Próximo da escolha Gulosa (FPG), Equiprovável (E) e Híbrido (H). Os valores adotados para α ou as suas probabilidades são os seguintes:

- FPG: $\alpha = 0{,}2$
- E: $p(\alpha 1) = 0{,}1$; $p(\alpha 2) = 0{,}1$; $p(\alpha 3) = 0{,}1$; $p(\alpha 4) = 0{,}1$; $p(\alpha 5) = 0{,}1$; $p(\alpha 6) = 0{,}1$; $p(\alpha 7) = 0{,}1$; $p(\alpha 8) = 0{,}1$; $p(\alpha 9) = 0{,}1$; $p(\alpha 10) = 0{,}1$
- H: $p(\alpha 1) = 0{,}500$; $p(\alpha 2) = 0{,}250$; $p(\alpha 3) = 0{,}125$; $p(\alpha 4) = 0{,}030$; $p(\alpha 5) = 0{,}030$; $p(\alpha 6) = 0{,}030$; $p(\alpha 7) = 0{,}010$; $p(\alpha 8) = 0{,}010$; $p(\alpha 9) = 0{,}010$; $p(\alpha 10) = 0{,}005$

onde $p(\alpha_i)$ é a probabilidade do valor α_i ser selecionado, com os valores $0{,}1$; $0{,}2$; $0{,}3$; $0{,}4$; $0{,}5$; $0{,}6$;$0{,}7$; $0{,}8$; $0{,}9$ e $1{,}0$ sendo atribuídos respectivamente para $\alpha 1$, $\alpha 2$, ..., $\alpha 10$.

A Heurística 2 não utiliza o conceito de LCR, tomando sempre o vértice de maior ganho para ser adicionado ao novo subconjunto que está sendo formado. Porém, essa heurística tende a estacionar em algum ótimo local, não produzindo os menores cortes possíveis.

A proposta da Heurística 3 é exatamente a de tirar a otimalidade local que em algumas ocasiões a Heurística 2 incorre. A utilização da estratégia de variação do α FPG pode gerar situações de otimalidade local como na Heurística 2, apesar de gerar cortes menores após o particionamento do que quando utilizada a estratégia E, fazendo com que a execução da rotina de melhoramento sobre esses cortes menores seja rápida.

Durante a execução da rotina de particionamento, valores de α próximos de 1 produzem cortes mais altos do que os valores de α próximos de 0. Nesse caso, a escolha da estratégia de variação do α como E tende a fazer com que o algoritmo descarte a maioria desses cortes obtidos quando o α for selecionado com algum valor próximo a 1. Além disso, cortes maiores vindos da rotina de particionamento tendem a fazer com que a rotina de melhoramento seja executada mais demoradamente.

Diante dessa situação, Bonatto (2010) propôs a estratégia de variação do α chamada H, onde as possibilidades dos valores de α mais próximos da escolha gulosa serem selecionados são maiores, gerando assim cortes menores. Porém, como valores próximos de 1 para α ainda podem ser selecionados, mesmo que em uma probabilidade menor, alguns cortes altos ainda podem ser gerados.

Uma proposta de melhoria apresentada é uma estratégia de variação do α Equiprovável Gulosa (EG), com as probabilidades de α como a seguir:

- EG: $p(\alpha 1) = 0{,}3333$; $p(\alpha 2) = 0{,}3333$; $p(\alpha 3) = 0{,}3333$

Com essa estratégia de variação do α, as vantagens obtidas são a menor possibilidade de se encontrar otimalidades locais, como na escolha da estratégia FPG, juntamente com rápida execução da rotina de melhoramento, visto que a tendência dos cortes após o particionamento é de serem menores, ao contrário do que ocorre na estratégia E e em menor escala, na estratégia H.

3.3 TIPOS ABSTRATOS DE DADOS UTILIZADOS

Na implementação do algoritmo paralelo de particionamento, diversos objetos foram utilizados como Tipos Abstratos de Dados. Levando-se em conta fatores como o desempenho de execução e ausência de algumas estruturas básicas em *Java*, como por exemplo, os ponteiros, alguns objetos foram modelados como a seguir:

3.3.1 *CSRG (Compressed Sparse Row Graph)*

O grafo é o objeto principal a ser manipulado pelos algoritmos particionadores. Apesar da existência de diversas implementações de grafos tais como as listas e matrizes de adjacência e listas e matrizes de incidência, a representação escolhida para a implementação do algoritmo foi a *CSRG*.

A classe *CSRG* é uma classe de grafos que utiliza o formato compacto de representação de matrizes chamado *CSR* (*Compressed Sparse Row*). Além do fato de os *CSRG* terem menos sobrecarga do que muitos outros formatos de grafos (por exemplo, lista de adjacências), eles não fornecem qualquer mutabilidade, ou seja, não se pode adicionar ou remover vértices ou arestas de um grafo *CSRG*. Essa mutabilidade é utilizada em aplicações de alto desempenho ou para grafos muito grandes nos quais não há necessidade de se mudar nada nos mesmos (BOOST, 2013).

No caso do particionador paralelo, o grafo é utilizado apenas no formato de leitura, pois sua principal função é fornecer a relação entre os vértices e seus vizinhos.

Nos grafos utilizados pelo particionador paralelo deste trabalho não há situações em que nas respectivas matrizes de adjacência existam valores diferentes de 0 ou 1, ou seja, nenhum dos grafos a serem particionados são multigrafos (grafos com laços ou arestas múltiplas/paralelas unindo os vértices) (DIESTEL, 2000, p. 25).

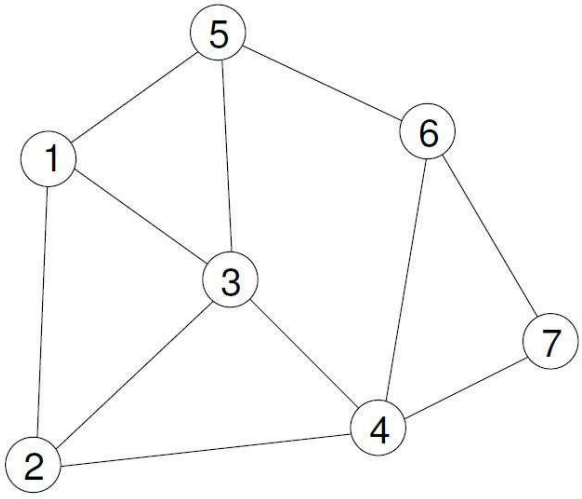
Além disso, tantos os pesos dos vértices quanto os pesos das arestas são unitários. Dessa maneira, o formato de armazenamento da matriz de adjacência que representa o grafo foi modificado a partir dessas pré-condições, sendo utilizado um formato *CSR* modificado para representação do *CSRG* implementado.

A partir da matriz de adjacência do grafo, são criados dois novos vetores chamados de *Vértices* e *Arestas*. Os tamanhos desses vetores são $2 \times |V|$ e $2 \times |E|$, respectivamente.

O vetor Vértices armazena nas posições de índice $2 \times (v_i - 1)$ e $2 \times (v_i - 1) + 1$ os valores dos índices inicial e final do vetor Arestas que contêm os vértices adjacentes a cada vértice v_i.

O vetor Arestas contém todos os números das colunas da matriz de adjacência (que representam os vértices adjacentes ao vértice v_i na linha i) cujo valor para aquela linha seja igual a 1.

A Figura 74 mostra um grafo com 7 vértices e 11 arestas. Logo à direita está sua representação por meio de uma matriz de adjacência.



Grafo

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	1	0	1	0	0
2	1	0	1	1	0	0	0
3	1	1	0	1	1	0	0
4	0	1	1	0	0	1	1
5	1	0	1	0	0	1	0
6	0	0	0	1	1	0	1
7	0	0	0	1	0	1	0

Matriz de Adjacência do Grafo

Figura 74 – Exemplo de Grafo e sua matriz de adjacência

Na Figura 75 são mostrados os vetores de vértices e de arestas que representam essa mesma matriz de adjacência no formato *CSR* modificado. Os tamanhos desses vetores são respectivamente $2 \times 7 = 14$ e $2 \times 11 = 22$.

Id dos Vértices	1		2		3		4		5		6		7	
Ponteiros	0	2	3	5	6	9	10	13	14	16	17	19	20	21
Índices	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Vetor de Vértices

Adjacentes	2	3	5	1	3	4	1	2	4	5	2	3	6	7	1	3	6	4	5	7	4	6
Índices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Vetor de Arestas

Figura 75 – Vetores do formato de representação *CSR* modificado

3.3.2 *Subset*

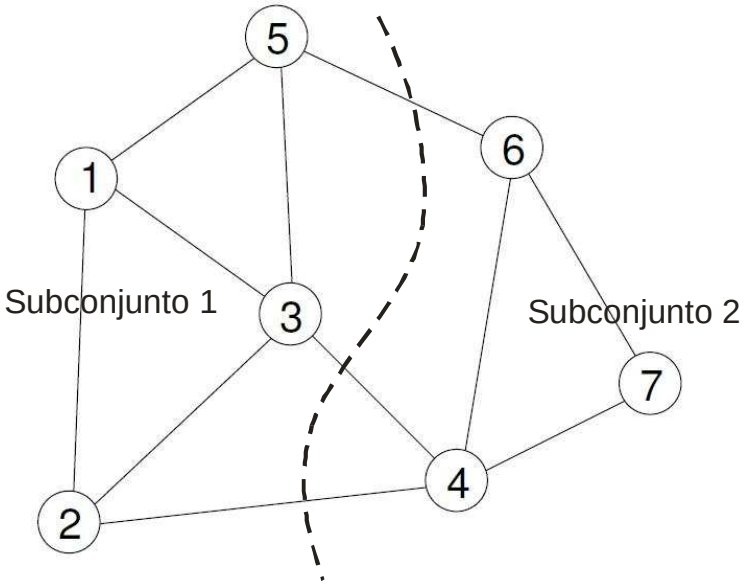
Os *subsets* (subconjuntos) são as estruturas que contêm efetivamente os vértices que pertencem a cada subconjunto nos quais o grafo está sendo particionado.

Na implementação do particionador paralelo os *subsets* são representados com vetores de booleanos de tamanho $|V|$. Como cada vértice do grafo é representado por um número entre 1 e $|V|$, quando o vértice v_i está presente em algum *subset*, a posição de índice $v_i - 1$ desse *subset* possui o valor *true* (verdadeiro). Caso o referido vértice v_i não faça parte desse *subset*, sua posição de índice $v_i - 1$ possui valor *false* (falso).

Apesar de ser esparso em situações em que o número de partições finais do grafo seja muito grande (na bisseção de um grafo, a esparsidade de cada *subset* é aproximadamente de 50%) e ocupar uma quantidade de memória relativamente grande no caso de grafos com muitos vértices, o acesso direto aos vértices do *subset* permite operações de inclusão, exclusão e verificação de pertinência de um vértice em tempo constante.

O número de *subsets* que são instanciados depende do número de partições em que o grafo será particionado. Para um particionamento em k partições são criados k *subsets*. Na implementação do particionador paralelo durante a execução do programa principal é instanciado um vetor de *subsets* de tamanho k.

A Figura 76 apresenta uma bisseção de um grafo e as suas respectivas representações dos *subsets*.



Vértices	1	2	3	4	5	6	7
Subconjunto 1	T	T	T	F	T	F	F
Subconjunto 2	F	F	F	T	F	T	T
Índices do vetor	0	1	2	3	4	5	6

Figura 76 – Grafo bissecionado e a representação dos seus *subsets*

3.3.3 *Bucket*

Os *buckets* são estruturas propostas por Fiduccia e Mattheyeses (1982) que tornam o acesso aos ganhos dos vértices do grafo muito eficiente. Quando o algoritmo particionador utiliza a Heurística 1 para compor o novo subconjunto que está sendo formado, a escolha do próximo vértice que fará parte desse subconjunto em formação é totalmente aleatória.

Porém, ao se utilizar as Heurísticas 2 ou 3, os vértices da fronteira devem ser ordenados por ordem decrescente de ganho para que o algoritmo escolha o maior deles (Heurística 2) ou escolha aleatoriamente dentre os vértices da LCR (Heurística 3).

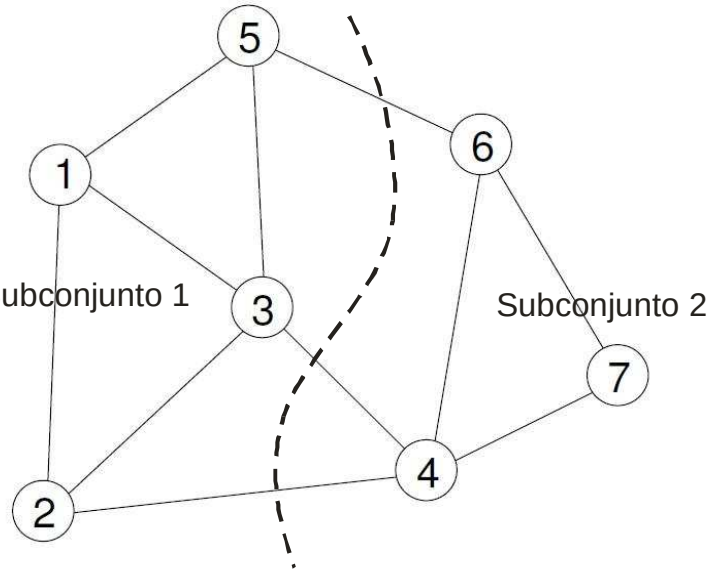
Em ambas as situações, o algoritmo particionador precisa ter acesso de forma rápida aos vértices ordenados decrescentemente pelos seus ganhos.

Como na linguagem de programação *Java* não existe a implementação de ponteiros como nas linguagens *C*/*C*++ para que o TAD Lista Duplamente Encadeada fosse definido e utilizado, alternativamente o algoritmo particionador paralelo utiliza a classe *ArrayList* do *Java*.

A classe *Bucket* que é implementada no algoritmo particionador paralelo é composta de um vetor de *ArrayList*, que são as listas contendo os vértices nas respectivas linhas que representam os seus ganhos e um vetor de inteiros que guarda o índice da linha em que cada vértice se encontra.

Como não é possível manter um ponteiro no vetor de inteiros diretamente para cada elemento das listas, a classe *bucket* guarda um ponteiro apenas para o índice do vetor de *ArrayList* onde o ganho do vértice se encontra. Uma vez encontrado o *ArrayList* no qual o vértice se encontra, o método `get()` da classe *ArrayList* retorna o elemento procurado.

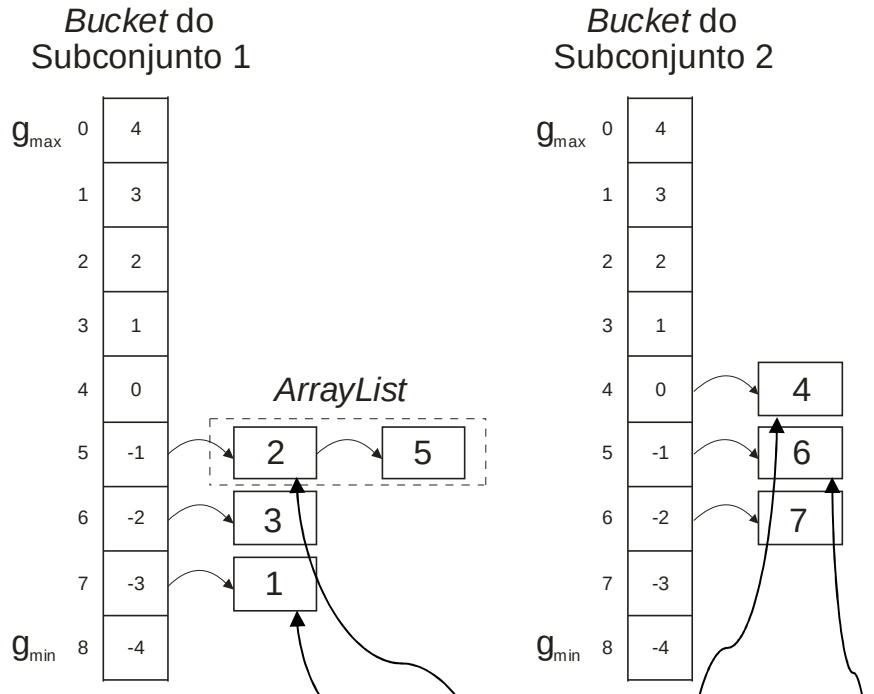
a)



b)

Vértice	1	2	3	4	5	6	7
Ganho	0 – 3 = –3	1 – 2 = –1	1 – 3 = –2	2 – 2 = 0	1 – 2 = –1	1 – 2 = –1	0 – 2 = –2

c)



d)

Vértices	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Ponteiro para Linha do *Bucket* 1	7	5	6	-∞	5	-∞	-∞
Ponteiro para Linha do *Bucket* 2	-∞	-∞	-∞	4	-∞	5	6
Índices do vetor de ponteiros de linhas	0	1	2	3	4	5	6

Figura 77 – Grafo bissecionado e a representação dos seus *buckets*

A Figura 77 ilustra um exemplo do uso dos *buckets* implementados pelo algoritmo particionador. Em (a) está o grafo bissecionado. Em (b) está a tabela dos ganhos $g(i)$ dos vértices, com $g(i) = E(i) - I(i)$. Em (c) estão as representações dos respectivos *buckets*, sendo instanciado um *bucket* para cada partição do grafo. Por último, em (d) estão os vetores com os ponteiros para as linhas dos vetores de *ArrayList* onde os vértices encontram-se armazenados.

Caso o algoritmo particionador esteja utilizando a Heurística 2 para particionar o grafo, o vértice de maior ganho que será utilizado para compor o novo subconjunto que está sendo formado localiza-se na primeira posição não vazia do vetor de *ArrayLists*. Por exemplo, no caso da Figura 77 (c), para o *Bucket* 1, seria o vértice 2.

Se o algoritmo particionador estiver utilizando a Heurística 3, uma Lista de Candidatos Restrita deve ser formada com os n vértices de maiores ganho. Um vértice aleatório da LCR será escolhido para compor o novo subconjunto a ser formado. Isso é feito da seguinte forma: começando do primeiro *ArrayList* não nulo do vetor de *ArrayList*, os elementos são lidos e armazenados em um novo vetor de tamanho n, até que esse seja preenchido. Depois de criado esse vetor, um vértice aleatório será escolhido a partir desse vetor.

Para a figura 77 (c) e uma LCR de tamanho igual a 3, os elementos que serão adicionados ao vetor para que depois seja feita a escolha aleatória do vértice a partir do *Bucket* 1 são na respectiva ordem 2, 5 e 3.

3.4 MÉTODOS, TÉCNICAS E EQUIPAMENTOS UTILIZADOS

Nessa seção são apresentadas as metodologias para as obtenções dos cortes em diversos grafos, além das técnicas utilizadas para análise das medidas de desempenho e uma descrição dos equipamentos utilizados nos testes computacionais.

3.4.1 Cortes obtidos nos particionamentos dos grafos

Em seu trabalho, Bonatto (2010) constatou que a execução da Heurística 3 gerava os melhores resultados nos cortes dos grafos. Em 10 execuções de 100 iterações cada uma, o algoritmo serial conseguiu, na maioria dos grafos analisados, uma diminuição do corte se comparado com o algoritmo multinível *METIS* e com o algoritmo multinível de bisseção espectral *CHACO*.

Como as heurísticas propostas são *multistart*, ou seja, várias partições são geradas e as de menores cortes são aproveitadas, o aumento da quantidade de iterações para geração das várias partições pode proporcionar melhorias nos cortes das mesmas.

Nos testes realizados, assim como Bonatto (2010), que realizou 10 execuções de 100 iterações da Heurística 3 Serial para obtenção do menor corte totalizando assim $10 \times 100 = 1.000$ iterações, também foram realizadas 10 execuções de 100 iterações cada uma, com a diferença que agora cada trabalhador de um *cluster* de 16 nós vai realizar essa tarefa, sendo que cada nó processador pode executar simultaneamente 16 *threads*, totalizando assim $10 \times 100 \times 16 \times 16 = 256.000$ iterações da Heurística H3 Paralela.

Diversos grafos foram utilizados nos experimentos. A Tabela 1 apresenta os grafos que foram particionados e suas características, tais como o número de vértices $|V|$, o número de arestas $|E|$, o grau mínimo d_{min}, o grau máximo d_{max}, o grau médio $d_{médio}$ e a densidade.

Tabela 1 – Grafos utilizados nos experimentos e suas características

(continua)

Grafo	$\|V\|$	$\|E\|$	d_{min}	d_{max}	$d_{médio}$	Densidade
144	144649	1074393	4	26	14,8552	0,01027
3elt	4720	13722	3	9	5,8144	0,12321
598a	110971	741934	5	26	13,3717	0,01205
add20	2395	7462	1	123	6,2313	0,26029
add32	4960	9462	1	31	3,8153	0,07694
airfoil1	4253	12289	3	9	5,7790	0,13591
bcsstk29	13992	302748	4	70	43,274	0,3093

Tabela 1 – Grafos utilizados nos experimentos e suas características

(conclusão)

Grafo	$\|V\|$	$\|E\|$	d_{min}	d_{max}	$d_{médio}$	Densidade
bcsstk33	8738	291583	19	140	66,7391	0,76387
big	15606	45878	3	10	5,8795	0,03768
CCC5	160	240	3	3	3,0000	1,88679
crack	10240	30380	3	9	5,9336	0,05795
data	2851	15093	3	17	10,5879	0,37150
fe_rotor	99617	662431	5	125	13,2996	0,01335
fe_tooth	78136	452591	3	39	11,5847	0,01483
G124.04	124	318	1	11	5,1290	4,16994
G124.16	124	1271	12	30	20,5000	16,66670
G250.08	250	2421	10	30	19,3680	7,77831
G500.02	500	2355	2	18	9,4200	1,88778
Grid32x32	1024	1984	2	4	3,8750	0,37879
memplus	17758	54196	1	573	6,1038	0,03437
nasa1824	1824	18692	5	41	20,4956	1,12428
nasa2146	2146	35052	13	35	32,6673	1,52295
stufe	1036	1868	2	4	3,6062	0,34842
U500.20	500	4549	4	30	18,1960	3,64649
U500.40	500	8793	8	56	35,1720	7,04850
U1000.20	1000	9339	4	39	18,6780	1,86967
U1000.40	1000	18015	9	58	36,0300	3,60661
vibrobox	12328	165250	8	120	26,8089	0,21748
whitaker	9800	28989	3	8	5,9161	0,06037
wing_nodal	10937	75488	5	28	13,8042	0,12623

Fonte: Bonatto (2010)

Os grafos utilizados nos experimentos estão disponíveis na *Internet* nos seguintes endereços eletrônicos:

- http://www2.cs.uni-paderborn.de/cs/ag-monien/RESEARCH/PART/graphs.html
- http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/#graphs

Bonatto (2010) utilizou o *software p-METIS*, que é indicado para até 8 subconjuntos e produz partições balanceadas e o *software CHACO* com o número de vértices do grafo reduzido igual a 50.

Além dos menores cortes para cada grafo, foram apresentados também os desvios padrões referentes aos cortes obtidos nas 10 execuções. O desvio padrão representa a homogeneidade dos cortes obtidos durante as execuções, sendo que nos casos em que o mesmo obteve valor zero ou próximo dele, todas ou a maioria

das execuções conseguiu atingir o menor corte apresentado ou um valor próximo dele respectivamente. Nos casos em que o desvio padrão apresenta um valor muito alto significa que durante as execuções do algoritmo de particionamento os cortes encontrados foram bem diferentes do corte mínimo encontrado.

3.4.2 Comparativo entre os Métodos de Melhoramento

Bonatto (2010) propôs um Método de Melhoramento a ser aplicado nas partições obtidas e em suas fronteiras após o particionamento. Uma das melhorias propostas por este trabalho foi uma modificação nesse Método de Melhoramento.

Os Métodos de Melhoramento Original e Modificado foram executados em 10 execuções de apenas 10 iterações cada da Heurística 3. O pequeno número de iterações em cada rodada de execução justifica-se pelo fato de que se esse número for muito elevado o algoritmo tenderia a levar os dois métodos de melhoramento aos menores cortes, não sendo possível uma comparação entre os mesmos.

3.4.3 Análise de *Speedup* e Eficiência Paralela

Um dos objetivos desse trabalho é a melhoria do tempo de execução a partir da paralelização dos algoritmos de particionamento. Em computação paralela, dois índices são muito utilizados para mensurar essa melhoria no desempenho: o *speedup* e a eficiência paralela.

O *speedup* mede o quanto a execução paralela de um determinado número de nós do *cluster* foi mais rápida do que a execução serial e a eficiência paralela, que mostra a real utilização do processador durante as execuções paralelas.

Para avaliar o desempenho da execução paralela, foram propostas diversas configurações de execução, cada uma delas com uma respectiva quantidade de nós, *threads* e iterações, sendo que em todas elas 6.400 iterações são executadas. A Tabela 2 apresenta cada uma dessas configurações.

Tabela 2 – Configurações de execuções serial e paralelas

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
16n16t	16	16	25	6.400

Como a paralelização do algoritmo foi realizada tanto em relação ao número de *cores* do nó processador (e em consequência, com relação ao número de *threads* que o mesmo pode executar) quanto com relação ao número de nós do *cluster*, a configuração 01n01t foi proposta. Ela representa a situação máxima de serialização, ou seja, a execução do algoritmo de particionamento utilizando uma *thread* e apenas um nó processador do *cluster*.

Por outro lado, para a configuração do *cluster* utilizada no experimento, a configuração 16n16t representa a situação de máxima paralelização, ou seja, 16 nós processadores com a execução de 16 *threads* em cada um deles.

Foram medidos os tempos para o particionamento dos grafos em $k = 2$ subconjuntos e feito um comparativo entre as diversas configurações de execuções serial e paralelas.

Para a análise de *speedup*, foram feitas comparações das razões de ganho entre as configurações paralelas e a configuração 01n01t e também entre as configurações paralelas e a configuração 01n16t. Elas demonstram os ganhos ao se comparar um primeiro nível de paralelização por meio apenas do aumento de *threads* sobre uma configuração serial e depois os ganhos obtidos com relação ao aumento do número de *threads* e nós do *cluster*.

A análise de eficiência paralela mostra o quanto o processador é realmente utilizado na execução paralela. Para sistemas ideais, a eficiência paralela é igual a 1. Em

situações práticas, seus valores ficam entre 0 e 1, mostrando efetivamente o quanto o processador foi utilizado em cada configuração.

A eficiência paralela é analisada apenas da configuração 02n16t em diante, já que é necessário pelo menos dois nós processadores para se calcular a mesma e que todos os nós processadores do *cluster* estejam executando as mesmas quantidades de *threads*. Ao se comparar a eficiência paralela com a configuração serial máxima (01n01t), os valores encontrados são maiores que 1, estando dessa forma fora da faixa de valores que a eficiência paralela trabalha que é entre 0 e 1.

3.4.4 Equipamentos utilizados

Para os experimentos de particionamento e análises de desempenho foi utilizado um *cluster* com 16 nós com a configuração apresentada no Quadro 3, com suas respectivas quantidades de nós e descrição.

Qtde	Descrição
01	• 02 Processadores Quad Core Intel X5570 Xeon, 2.93 GHz, 8 MB Cache, 6.4 GT/s QuickPath Interconnect, Tecnologia Turbo HyperThreading; • Índice SPECint_rate2006 (baseline) de 232 op/s; • 32 GB de memória RAM com taxa de transferência de 10.667 MB/s; • 02 unidades de disco rígido SATA hot swapping de 1 TB, 7.200 RPM, RAID 1; • 01 unidade gravadora de CD/DVD. • 04 interfaces de rede com padrão 10BaseT/100BaseTX/ 1000BaseTX e suporte a balanceamento de carga; • Controladora de vídeo integrada de 8 MB de memória não compartilhada.
15	• 02 Processadores Quad Core Intel E5530 Xeon, 2.4 GHz, 8 MB Cache, 5.86 GT/s QuickPath Interconnect, Tecnologia Turbo HyperThreading; • Índice SPECint_rate2006 (baseline) de 194 op/s; • 32 GB de memória RAM com taxa de transferência de 10.667 MB/s; • 02 unidades de disco rígido SATA hot swapping de 1 TB, 7.200 RPM, RAID 1; • 01 unidade gravadora de CD/DVD. • 04 interfaces de rede com padrão 10BaseT/100BaseTX/ 1000BaseTX e suporte a balanceamento de carga; • Controladora de vídeo integrada de 8 MB de memória não compartilhada.

Quadro 3 – Configurações dos equipamentos utilizados

Como Sistema Operacional foi utilizada a distribuição *Ubuntu Server* 12.04.4 *LTS* 64 *bits* tanto no processador *root* como nos demais nós.

A versão da máquina virtual e do compilador *Java* instalados no *cluster* é a 8 e a versão do *MPJ Express* é a 0.38. Para a implementação do algoritmo particionador foi utilizada a *IDE Netbeans* 7.4 com a mesma versão do *Java* do *cluster* instalada no computador de desenvolvimento.

No capítulo seguinte são apresentados os resultados obtidos com os experimentos e a discussão dos mesmos.

4 RESULTADOS E DISCUSSÕES

Nesse capítulo são apresentados os resultados obtidos com a execução do particionador paralelo em vários grafos, obtendo cortes para diversos valores de k subconjuntos.

Além disso, são apresentados os dados obtidos com a comparação entre o Método de Melhoramento Original e o Método de Melhoramento Modificado proposto.

Uma análise sobre o *speedup* obtido com o uso do particionador paralelo comparado com o particionador serial e uma análise de eficiência paralela são apresentadas em seguida.

4.1 CORTES OBTIDOS NOS PARTICIONAMENTOS DOS GRAFOS

Nas Tabelas a seguir, a Heurística 3 Serial de Bonatto (2010) foi executada com $10 \times 100 = 1.000$ iterações e a Heurística 3 Paralela foi executada com $10 \times 100 \times 16 \times 16 = 256.000$ iterações.

Para uma bisseção ($k = 2$ subconjuntos), os valores dos menores cortes encontrados e os seus respectivos desvios padrões com relação ao menor corte médio obtido pelas 10 execuções – tanto do algoritmo serial quanto do algoritmo paralelo – são apresentados na Tabela 3. Os menores cortes obtidos estão em destaque. Caso os menores cortes apareçam como resultado da execução de mais de um algoritmo, todos estão destacados.

Tabela 3 – Comparativo de cortes e desvios padrões para $k = 2$ subconjuntos

(continua)

			H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**CHACO**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
144	**6871**	6994	7248	211,6890	7575	80,5412
3elt	98	103	93	6,7790	**90**	0,0000
598a	2470	2484	2476	49,7030	**2463**	50,7346
add20	741	742	715	10,2250	**646**	10,9659
add32	19	**11**	**11**	0,7270	**11**	0,0000
airfoil1	85	82	77	4,7250	**74**	1,6248

Tabela 3 – Comparativo de cortes e desvios padrões para $k = 2$ subconjuntos

(conclusão)

			H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**CHACO**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
bcsstk29	2923	3140	2885	66,3920	**2851**	0,9798
bcsstk33	12620	10172	**10171**	0,7380	**10171**	0,0000
big	165	150	160	15,3930	**146**	1,6155
CCC5	28	29	**16**	2,7968	**16**	0,0000
crack	206	266	194	6,0190	**184**	1,2207
data	203	234	**195**	2,3940	**195**	1,6733
fe_rotor	2146	2230	2161	16,0910	**2107**	7,7743
fe_tooth	4198	4642	4113	152,8240	**3984**	26,5586
G124.04	65	65	**63**	0,4830	**63**	0,0000
G124.16	462	465	**449**	0,0000	**449**	0,0000
G250.08	860	845	**828**	0,8433	**828**	0,0000
G500.02	661	636	630	3,2740	**628**	0,9220
Grid32x32	**32**	34	**32**	1,2650	**32**	0,0000
memplus	6671	7549	6534	3,3270	**6524**	1,9596
nasa1824	757	824	**739**	30,0040	**739**	0,0000
nasa2146	882	**870**	**870**	0,0000	**870**	0,0000
stufe	19	20	**16**	0,6320	**16**	0,0000
U500.20	**178**	182	**178**	0,4220	**178**	0,0000
U500.40	**412**	**412**	**412**	0,0000	**412**	0,0000
U1000.20	**222**	286	**222**	6,8028	**222**	0,0000
U1000.40	812	895	**737**	0,0000	**737**	0,0000
vibrobox	11746	11196	10489	659,6330	**10343**	1,7321
whitaker	128	133	128	1,4340	**127**	0,0000
wing_nodal	1848	1850	1746	10,3180	**1727**	4,8332

Fonte: Bonatto (2010)

Em 46,67% dos grafos o menor corte foi melhorado pela aplicação do algoritmo paralelo de particionamento com relação aos outros algoritmos particionadores e com relação à execução serial da Heurística 3 de Bonatto (2010). Em 50,00% dos grafos o menor corte foi mantido. Em 3,33% dos grafos o menor corte foi piorado. Esses resultados são exibidos no Gráfico 1.

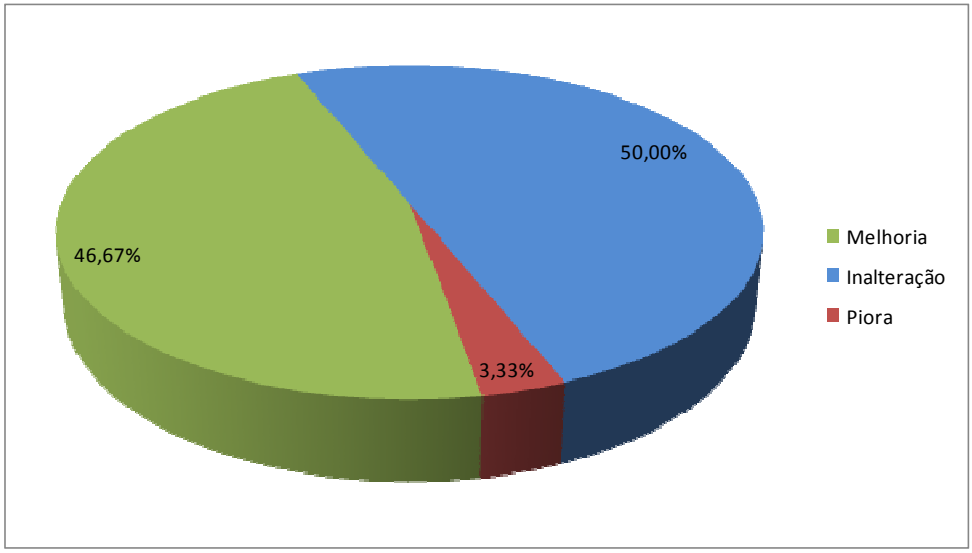


Gráfico 1 – Situação dos cortes para $k = 2$ subconjuntos

Para um particionamento dos grafos em $k = 4$ subconjuntos, os valores dos menores cortes encontrados e os seus respectivos desvios padrões com relação ao menor corte médio obtido pelas 10 execuções – tanto do algoritmo serial quanto do algoritmo paralelo – são apresentados na Tabela 4. Os menores cortes obtidos estão em destaque. Caso os menores cortes apareçam como resultado da execução de mais de um algoritmo, todos estão destacados.

Tabela 4 – Comparativo de cortes e desvios padrões para $k = 4$ subconjuntos

(continua)

			H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**CHACO**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
144	17649	17996	17996	677,6831	**17622**	168,7602
3elt	252	230	230	16,7252	**208**	0,9165
598a	**8300**	9013	9013	322,7912	8394	71,7694
add20	1335	1236	1236	15,6663	**1182**	12,7138
add32	56	**34**	**34**	1,2293	**34**	0,0000
airfoil1	179	178	178	10,2654	**162**	0,8000
bcsstk29	8820	**8477**	**8477**	240,7043	8524	1,0000
bcsstk33	22635	**22121**	**22121**	114,2837	22764	0,0000
big	405	451	451	25,5389	**364**	7,1056
crack	458	411	411	21,4707	**371**	23,3749
data	416	437	437	17,7266	**402**	8,6626
fe_rotor	8070	8664	8664	227,6441	**7779**	10,0020
fe_tooth	8063	9618	9618	309,8448	**8033**	34,4029
G500.02	1069	1071	1071	3,5839	**1026**	2,9257
Grid32x32	**64**	66	66	8,7490	**64**	0,0000
memplus	10412	10075	10075	29,3629	**9879**	6,0374
nasa1824	1627	1619	1619	32,7963	**1547**	16,7380
stufe	55	56	56	3,7476	**50**	0,0000

Tabela 4 – Comparativo de cortes e desvios padrões para $k = 4$ subconjuntos

(conclusão)

Grafo	p-METIS	CHACO	H3 Serial		H3 Paralela	
			Corte	Desv. Pad.	Corte	Desv. Pad.
U500.20	394	**351**	**351**	7,3371	384	0,0000
U500.40	1040	**1020**	**1020**	0,0000	1057	0,0000
vibrobox	20576	22311	22311	1030,3080	**20402**	3,7148
whitaker	424	437	437	11,3117	**382**	0,6633
wing_nodal	4215	3739	3739	53,1477	**3681**	29,0730

Fonte: Bonatto (2010)

Em 69,57% dos grafos o menor corte foi melhorado pela aplicação do algoritmo paralelo de particionamento com relação aos outros algoritmos particionadores e com relação à execução serial da Heurística 3 de Bonatto (2010). Em 8,70% dos grafos o menor corte foi mantido. Em 21,74% dos grafos o menor corte foi piorado. Esses resultados são exibidos no Gráfico 2.

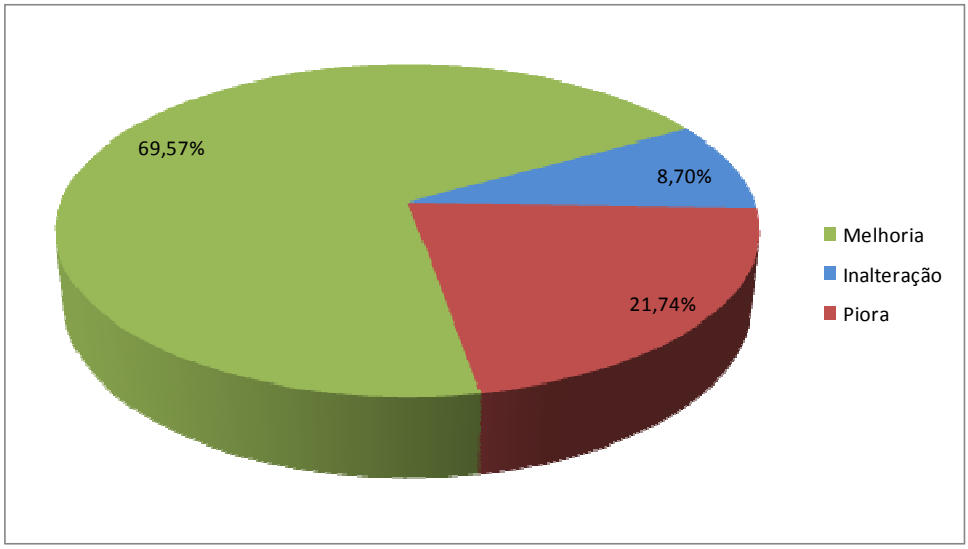


Gráfico 2 – Situação dos cortes para $k = 4$ subconjuntos

Para um particionamento dos grafos em $k = 8$ subconjuntos, os valores dos menores cortes encontrados e os seus respectivos desvios padrões com relação ao menor corte médio obtido pelas 10 execuções – tanto do algoritmo serial quanto do algoritmo paralelo – são apresentados na Tabela 5. Os menores cortes obtidos estão em destaque. Caso os menores cortes apareçam como resultado da execução de mais de um algoritmo, todos estão destacados.

Tabela 5 – Comparativo de cortes e desvios padrões para $k = 8$ subconjuntos

		H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
144	28121	30958	476,4652	**28092**	316,1472
3elt	416	413	12,5649	**371**	2,6096
598a	**17440**	19268	374,3474	17492	78,5138
add20	1961	1760	24,2441	**1748**	51,3740
add32	75	73	8,8575	**68**	1,2000
airfoil1	310	349	9,9381	**303**	3,5440
bcsstk29	18273	15576	535,9155	**14837**	77,6003
bcsstk33	37424	**36090**	906,3033	38918	7,0541
big	668	741	24,3687	**583**	12,6842
crack	747	823	23,2226	**736**	3,4073
data	745	786	18,1402	**722**	6,9347
fe_rotor	14449	16804	375,0653	**13540**	59,0393
fe_tooth	13034	15694	432,1426	**12324**	73,7692
G500.02	1345	1365	4,8259	**1292**	3,9560
Grid32x32	133	147	4,3729	**128**	0,0000
memplus	12676	12979	68,0800	**12214**	13,1833
nasa1824	2587	2577	21,3604	**2431**	81,9119
stufe	144	121	4,3716	**105**	0,7746
U500.20	686	**647**	18,7983	690	0,3000
U500.40	2117	**1885**	6,0332	2085	2,9394
vibrobox	**27622**	33369	302,3077	31665	97,7620
whitaker	710	809	16,0748	**683**	0,9434
wing_nodal	6179	5904	145,7359	**5850**	102,2272

Fonte: Bonatto (2010)

Em 78,26% dos grafos o menor corte foi melhorado pela aplicação do algoritmo paralelo de particionamento com relação aos outros algoritmos particionadores e com relação à execução serial da Heurística 3 de Bonatto (2010). Em 21,74% dos grafos o menor corte foi piorado. Esses resultados são exibidos no Gráfico 3.

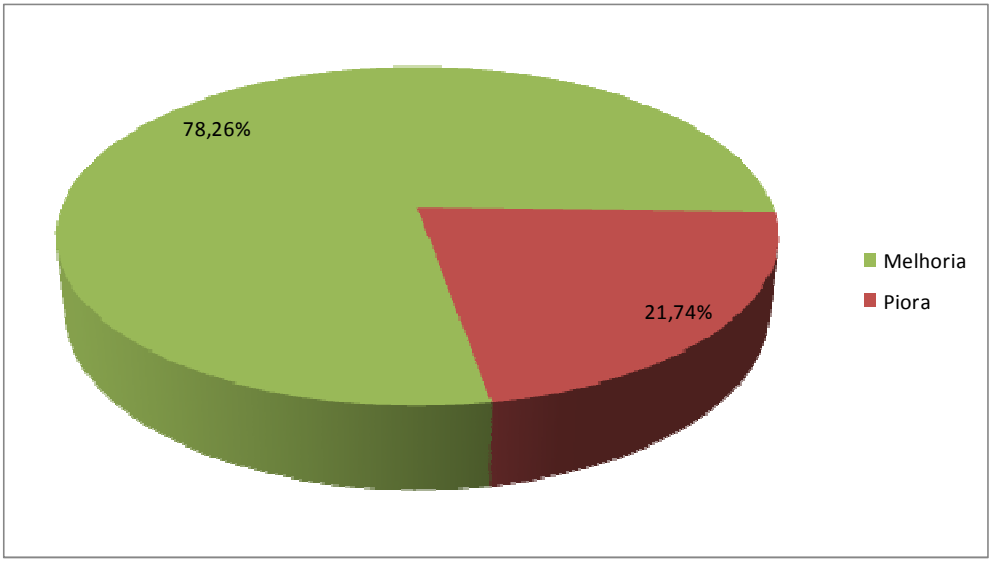


Gráfico 3 – Situação dos cortes para $k = 8$ subconjuntos

Para um particionamento dos grafos em $k = 16$ subconjuntos, os valores dos menores cortes encontrados e os seus respectivos desvios padrões com relação ao menor corte médio obtido pelas 10 execuções – tanto do algoritmo serial quanto do algoritmo paralelo – são apresentados na Tabela 6. Os menores cortes obtidos estão em destaque. Caso os menores cortes apareçam como resultado da execução de mais de um algoritmos, todos estão destacados.

Tabela 6 – Comparativo de cortes e desvios padrões para $k = 16$ subconjuntos

(continua)

		H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
144	43955	44665	618,1515	**43101**	243,3820
3elt	676	682	19,6822	**621**	3,8223
598a	30016	30005	608,4144	**28975**	427,9122
add20	2497	2263	23,8942	**2175**	49,3806
add32	129	155	12,9632	**122**	2,3664
airfoil1	558	625	16,8394	**547**	3,0017
bcsstk29	27292	**24046**	845,6151	24231	102,2184
bcsstk33	**58760**	59554	454,5342	59751	78,1704
big	1122	1255	28,0296	**1024**	15,3261
crack	1285	1344	36,8572	**1167**	7,4626
data	1330	1378	18,6098	**1222**	18,7339
fe_rotor	23448	26630	498,9420	**22196**	80,1805
fe_tooth	20106	23682	205,6665	**18988**	135,2805
G500.02	1537	1577	3,3149	**1491**	4,1485
Grid32x32	211	246	6,6131	**192**	0,0000
memplus	14824	16314	176,2932	**14535**	21,1388
nasa1824	3758	3768	40,5989	**3609**	31,5381
stufe	193	210	4,2896	**186**	1,4832

Tabela 6 – Comparativo de cortes e desvios padrões para $k = 16$ subconjuntos

(conclusão)

		H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
U500.20	1164	**1104**	12,0504	1150	13,6121
U500.40	3541	3534	31,34911	**3530**	3,7094
vibrobox	**36350**	38707	348,1268	41964	90,2114
whitaker	1198	1281	31,6349	**1167**	9,2309
wing_nodal	9317	9452	66,4096	**8952**	39,7442

Fonte: Bonatto (2010)

Em 82,61% dos grafos o menor corte foi melhorado pela aplicação do algoritmo paralelo de particionamento com relação aos outros algoritmos particionadores e com relação à execução serial da Heurística 3 de Bonatto (2010). Em 17,39% dos grafos o menor corte foi piorado. Esses resultados são exibidos no Gráfico 4.

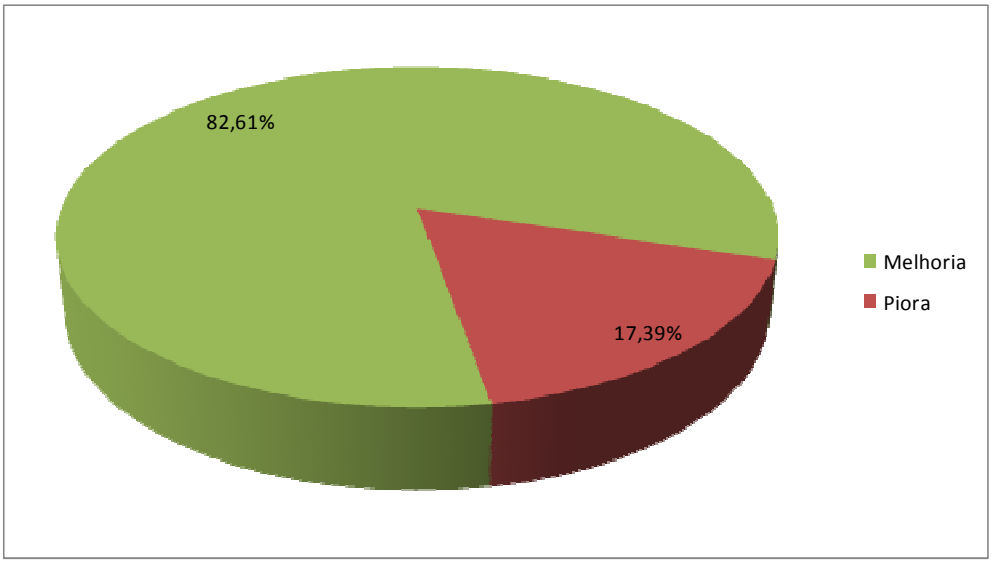


Gráfico 4 – Situação dos cortes para $k = 16$ subconjuntos

Para um particionamento dos grafos em $k = 32$ subconjuntos, os valores dos menores cortes encontrados e os seus respectivos desvios padrões com relação ao menor corte médio obtido pelas 10 execuções – tanto do algoritmo serial quanto do algoritmo paralelo – são apresentados na Tabela 7. Os menores cortes obtidos estão em destaque. Caso os menores cortes apareçam como resultado da execução de mais de um algoritmo, todos estão destacados.

Tabela 7 – Comparativo de cortes e desvios padrões para $k = 32$ subconjuntos

		H3 Serial		H3 Paralela	
Grafo	**p-METIS**	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**
144	62631	65735	646,3254	**62047**	382,7242
3elt	1082	1198	18,9106	**1045**	6,8007
598a	43685	46044	618,9253	**43135**	269,1242
add20	2997	2772	56,1060	**2590**	92,9664
add32	279	351	7,9197	**214**	0,3727
airfoil1	978	1095	12,9893	**940**	4,5343
bcsstk29	41835	40646	439,0496	**38577**	87,0885
bcsstk33	**83987**	85247	392,0570	85126	126,3549
big	1782	2093	25,5710	**1681**	14,6615
crack	1951	2084	30,5723	**1816**	11,6379
data	2096	2135	26,1287	**2012**	5,0990
fe_rotor	35799	40161	398,9744	**34755**	52,3637
fe_tooth	28532	32183	259,1284	**27043**	109,3008
G500.02	1683	1726	3,3682	**1647**	2,4875
Grid32x32	322	387	4,5473	**320**	0,000
memplus	16897	17210	235,8021	**16384**	32,9569
nasa1824	5529	5298	51,4570	**5196**	28,8083
stufe	313	334	3,5590	**291**	3,1875
U500.20	1975	1934	13,4098	**1928**	6,3019
U500.40	**5439**	5498	22,0719	5452	0,0000
vibrobox	**44665**	47545	378,1084	50039	158,9621
whitaker	1868	1995	35,3547	**1797**	3,8909
wing_nodal	13249	13518	79,3628	**12927**	64,0831

Fonte: Bonatto (2010)

Em 86,96% dos grafos o menor corte foi melhorado pela aplicação do algoritmo paralelo de particionamento com relação aos outros algoritmos particionadores e com relação à execução serial da Heurística 3 de Bonatto (2010). Em 13,04% dos grafos o menor corte foi piorado. Esses resultados são exibidos no Gráfico 5.

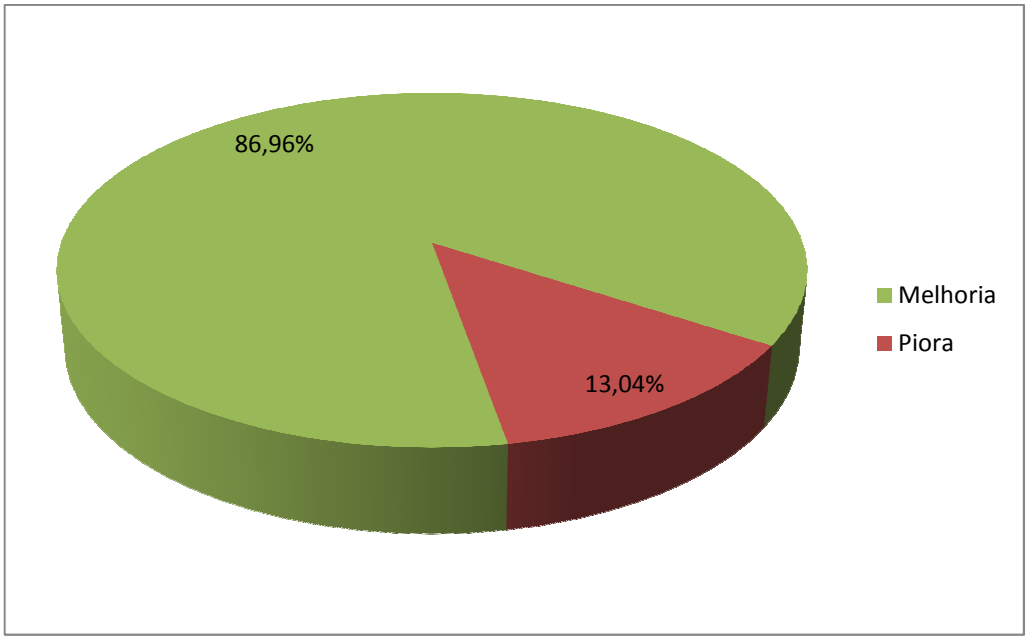


Gráfico 5 – Situação dos cortes para $k = 32$ subconjuntos

Os cortes foram comparados em particionamentos para $k = 2, 4, 8, 16$ e 32 subconjuntos. Em todas essas situações, para os grafos analisados, o algoritmo paralelo de particionamento apresentou uma maior quantidade de melhorias e inalterações nos cortes do que pioras dos mesmos. Tais resultados são apresentados no Gráfico 6. À medida que o valor de k aumenta, a tendência da situação de melhoria dos menores cortes ao se utilizar o particionador paralelo é aumentar.

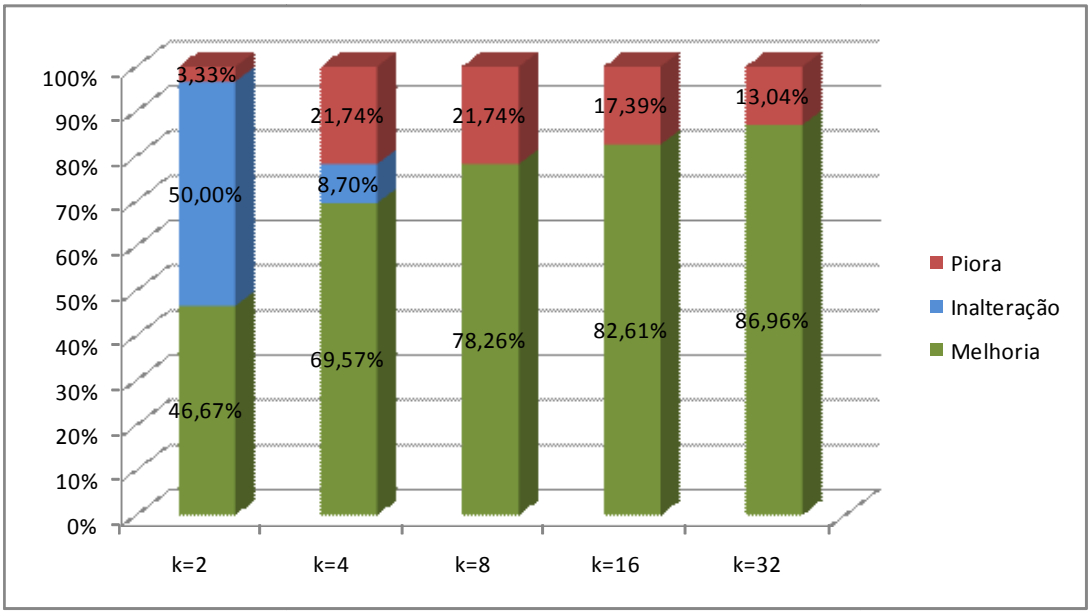


Gráfico 6 – Comparativo de 3 situações de cortes

As tendências de comportamento dos resultados dos particionamentos nessas três situações são apresentadas no Gráfico 7.

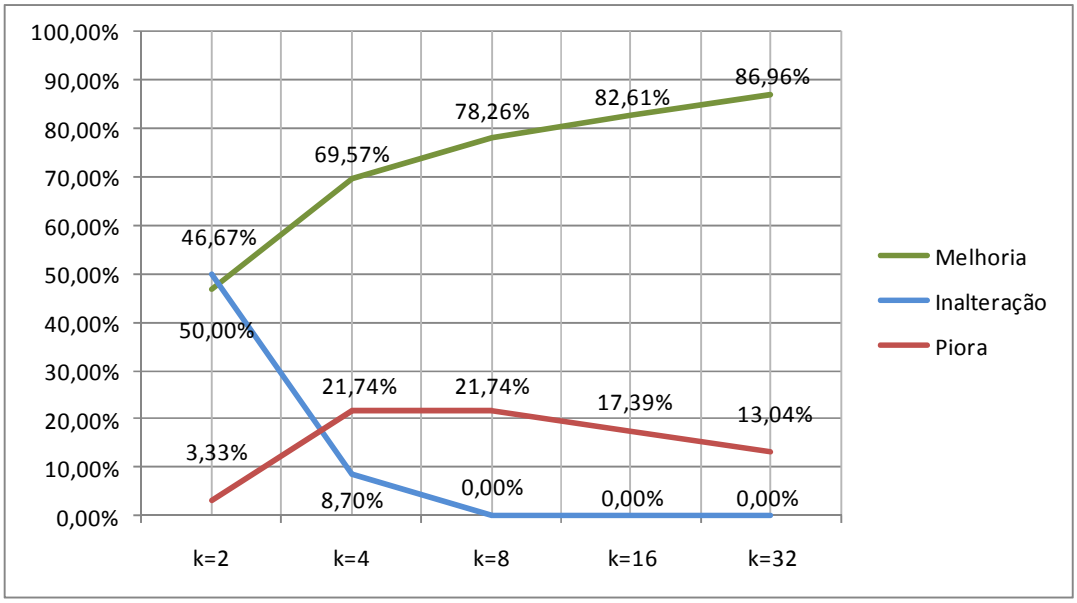


Gráfico 7 – Tendências de 3 situações de cortes

Caso sejam consideradas apenas as situações dos cortes como Melhoria/Inalteração e Piora, os resultados mostram-se ainda mais relevantes, como podem ser vistos no Gráfico 8.

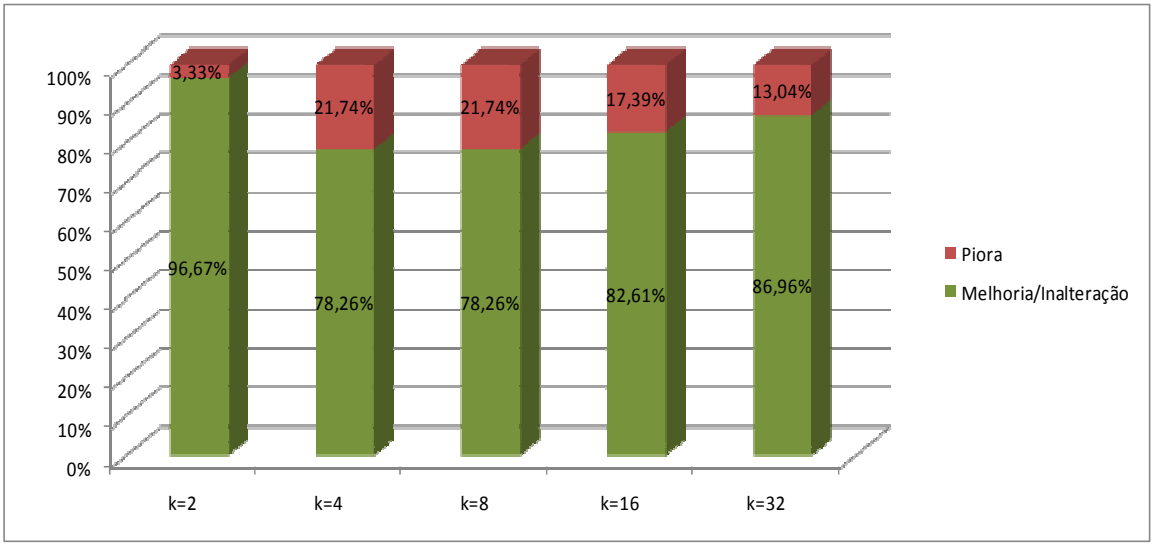


Gráfico 8 – Comparativo de 2 situações de cortes

As tendências de comportamento dos resultados dos particionamentos nessas duas situações são apresentadas no Gráfico 9.

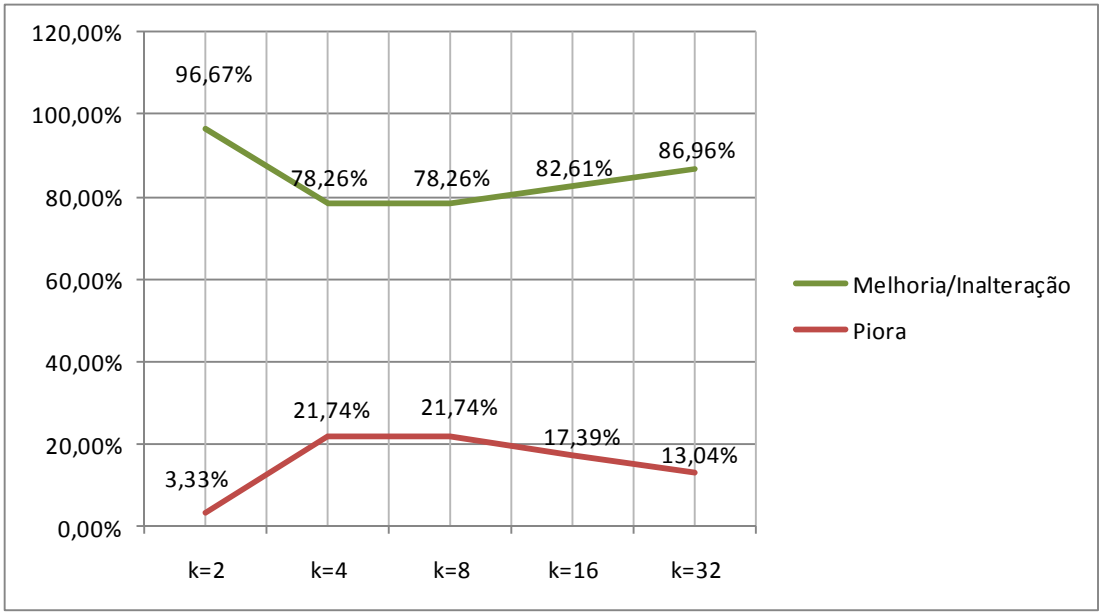


Gráfico 9 – Tendências de 2 situações de cortes

De todos grafos particionados para os diversos valores de k, o algoritmo particionador paralelo conseguiu uma melhoria nos cortes se comparado com o algoritmo serial de Bonatto (2010), METIS e CHACO em $71{,}31\%$ dos grafos. Em $13{,}93\%$ dos grafos os menores cortes mantiveram-se inalterados e em $14{,}75\%$ dos grafos o menor corte foi piorado. Esses resultados são apresentados no gráfico 10.

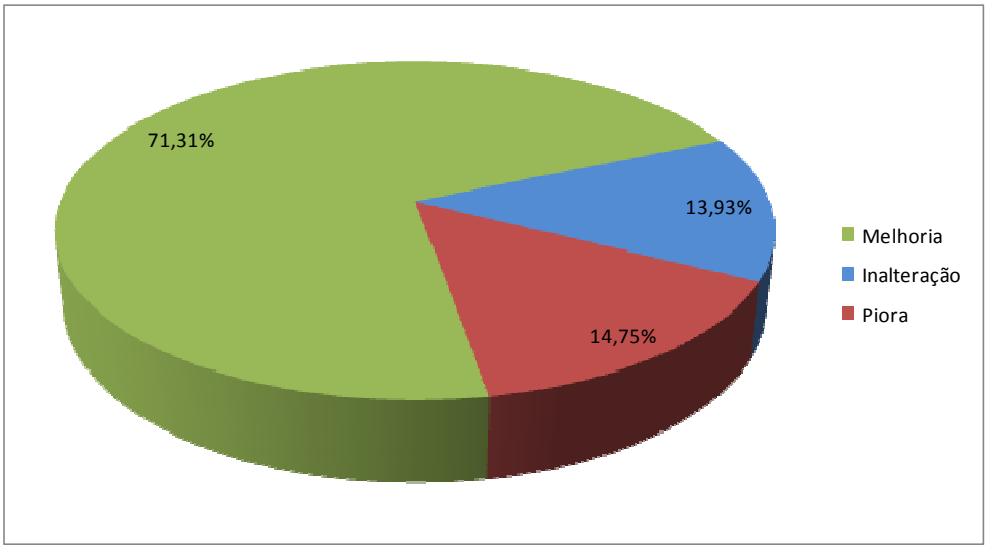


Gráfico 10 – Cortes para todos os grafos em 3 situações

Caso apenas as situações de Melhoria/Inalteração e Piora sejam consideradas, de todos os grafos particionados para os diversos valores de k, o algoritmo particionador paralelo conseguiu uma melhoria/inalteração nos cortes se comparado com o algoritmo serial de Bonatto (2010), METIS e CHACO em $85{,}25\%$ dos grafos. Em $14{,}75\%$ dos grafos o menor corte foi piorado. Esses resultados são apresentados no gráfico 11.

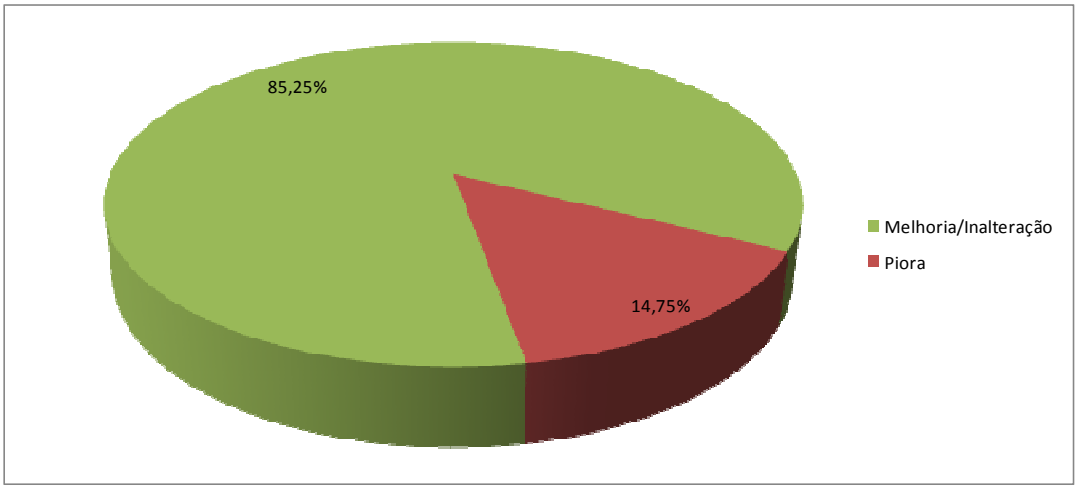


Gráfico 11 – Cortes para todos os grafos em 2 situações

4.2 COMPARATIVO ENTRE OS MÉTODOS DE MELHORAMENTO

Como uma das melhorias propostas por esse trabalho, uma alteração no Método de Melhoramento proposto por Bonatto (2010) propiciou sensíveis melhoras aos cortes dos grafos.

Em 10 execuções de 10 iterações cada uma utilizando a Heurística 3 aplicando o Método de Melhoramento Original de Bonatto (2010) e o Método de Melhoramento Modificado, houve uma melhoria média de $7{,}14\%$ nos cortes dos grafos.

A Tabela 8 apresenta um comparativo entre a aplicação dos dois Métodos de Melhoramentos.

Tabela 8 – Comparativo de cortes entre os métodos de melhoramento

	Original		Modificado		
Grafo	**Corte**	**Desv. Pad.**	**Corte**	**Desv. Pad.**	**Melhoria**
144	15633,5	2208,2125	9671,7	928,4161	38,13%
3elt	141,6	25,3605	128,6	25,3430	9,18%
598a	4255,9	397,9630	3253,5	233,2549	23,55%
add20	771,3	17,1662	755,5	10,7419	2,05%
add32	11,8	0,4216	11,7	0,4830	0,85%
airfoil1	126,3	23,3050	115,6	12,8513	8,47%
bcsstk29	4067,6	890,7095	3519,4	704,7739	13,48%
bcsstk33	11036,9	626,7194	10178,1	12,6881	7,78%
big	277,7	78,5918	236,7	67,8938	14,76%
CCC5	26,0	0,0000	24,2	0,6325	6,92%
crack	212,4	8,8719	209,7	6,7007	1,27%
data	236,2	13,4396	229,1	12,3329	3,01%
fe_rotor	2697,6	337,9353	2283,1	264,4522	15,37%
fe_tooth	5543,2	598,5076	5315,3	438,0477	4,11%
G124.04	70,1	1,7920	64,5	0,8498	7,99%
G124.16	455,4	5,3375	449	0,0000	1,41%
G250.08	864,0	9,2496	836,3	2,7508	3,21%
G500.02	692,9	13,5191	645,9	6,2973	6,78%
Grid32x32	36,0	1,0541	35,5	0,9718	1,39%
memplus	6556,6	5,5817	6549,3	2,6268	0,11%
nasa1824	963,4	46,8122	945,4	73,7084	1,87%
nasa2146	927,2	49,1320	877,9	17,0258	5,32%
stufe	31,1	6,9033	30,9	6,7074	0,64%
U500.20	187,4	13,6642	182,7	6,2548	2,51%
U500.40	422,9	34,4688	412,0	0,0000	2,58%
U1000.20	267,6	20,5437	253,1	15,5596	5,42%
U1000.40	786,4	58,9542	760,3	35,7151	3,32%
vibrobox	14084,1	635,4083	12351,7	737,2025	12,30%
whitaker	135,1	7,0151	132,2	2,5298	2,15%
wing_nodal	2094,3	231,5182	1923,9	64,4885	8,14%
Média	2453,75	212,2719	2079,43	123,0433	7,14%

Algumas melhorias são dignas de nota, como por exemplo, os grafos *144*, *598a* e *fe_rotor*, com melhorias de corte de, respectivamente, 38,13%, 23,55% e 15,37%.

4.3 ANÁLISE DE *SPEEDUP* E EFICIÊNCIA PARALELA

Os tempos obtidos (em segundos) para um particionamento em $k = 2$ subconjuntos são apresentados na Tabela 9.

Na última linha da tabela são apresentados os valores médios de cada coluna.

Tabela 9 – Tempos de particionamento dos grafos em $k = 2$ subconjuntos

	Configurações de execução					
Grafo	**01n01t**	**01n16t**	**02n16t**	**04n16t**	**08n16t**	**16n16t**
144	10.026,542	2.359,931	1.754,537	903,249	460,768	234,537
3elt	21,421	7,185	4,713	2,401	1,753	1,678
598a	4.958,073	1150,464	863,079	428,453	227,580	127,575
add20	34,021	7,432	5,193	3,000	2,242	1,840
add32	12,862	3,967	2,501	2,053	1,290	1,216
airfoil1	18,811	7,397	3,685	2,831	1,573	1,494
bcsstk29	866,011	166,337	118,070	61,495	31,942	17,061
bcsstk33	1.377,044	244,163	187,792	99,840	49,790	25,115
big	84,748	16,904	12,599	6,850	4,012	2,905
CCC5	0,702	1,107	0,758	0,583	0,444	0,353
crack	48,708	9,381	7,403	4,084	2,542	2,058
data	20,287	4,723	3,392	3,010	1,676	1,476
fe_rotor	2.903,147	668,780	468,868	245,231	127,434	71,026
fe_tooth	1.984,350	440,294	319,323	169,860	89,342	48,854
G124.04	1,750	2,419	1,389	0,792	0,533	0,416
G124.16	8,070	2,534	2,382	2,296	1,455	0,945
G250.08	18,930	3,506	4,717	2,110	2,026	1,731
G500.02	18,948	5,571	3,552	3,477	3,393	1,762
Grid32x32	2,670	2,443	1,580	0,913	0,697	0,582
memplus	392,016	57,475	51,185	26,089	13,402	7,259
nasa1824	43,796	6,978	5,951	3,415	2,829	2,392
nasa2146	62,216	11,147	8,615	5,051	2,637	2,204
stufe	3,914	2,828	2,379	1,509	1,046	0,744
U500.20	9,444	2,840	2,385	2,198	1,571	1,012
U500.40	23,564	6,286	3,862	2,204	2,411	1,635
U1000.20	17,395	4,412	3,194	2,158	2,194	1,342
U1000.40	46,231	8,971	6,872	3,808	3,289	2,076
vibrobox	1.447,952	212,310	204,773	105,422	52,277	26,461
whitaker	37,263	8,161	6,100	3,399	2,401	1,590
wing_nodal	277,704	39,913	36,859	18,333	9,844	5,678
Média	**825,620**	**182,195**	**136,590**	**70,537**	**36,813**	**19,834**

Tomando-se os valores médios como referência de análise, os tempos de execução diminuíram à medida que a quantidade de *threads* e nós processadores do *cluster* era aumentada.

Em alguns dos grafos, como por exemplo, o *144* e o *598a*, as execuções na configuração serial máxima (01n01t) despenderam 02h47m e 01h22m respectivamente contra 03m54s e 02m07s na configuração paralela máxima (16n16t).

O Gráfico 12 apresenta os tempos médios de execução das diversas configurações propostas.

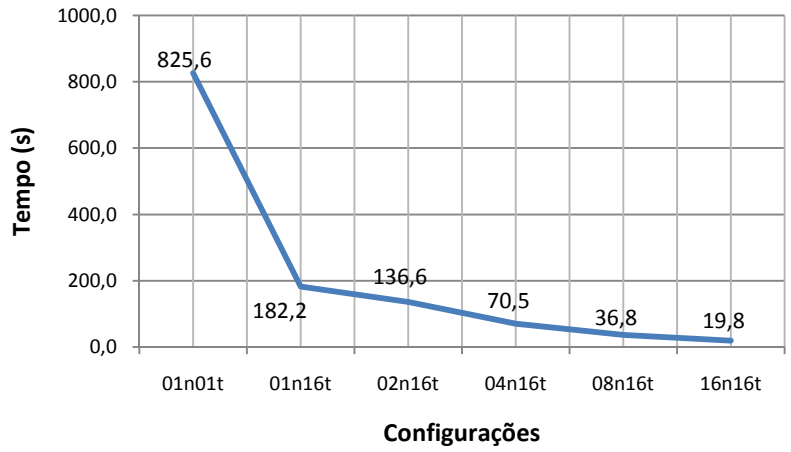


Gráfico 12 – Tempos médios de execução das configurações de paralelização

Um ganho expressivo de tempo foi conseguido com a implementação de uma configuração *multithreaded* se comparada com a implementação serial máxima. Em seguida, com o aumento do número de nós do *cluster* executando o particionador paralelo, o ganho de tempo obtido continuou de forma progressiva.

Os *speedups* obtidos com a execução das diversas configurações paralelas em comparação com a configuração serial máxima (01n01t) são apresentados na Tabela 10.

Tabela 10 – *Speedups* comparados com a configuração 01n01t

(continua)

	Configurações de execução					
Grafo	**01n01t**	**01n16t**	**02n16t**	**04n16t**	**08n16t**	**16n16t**
144	1,000	4,249	5,715	11,101	21,760	42,750
3elt	1,000	2,981	4,545	8,922	12,220	12,766
598a	1,000	4,310	5,745	11,572	21,786	38,864
add20	1,000	4,578	6,551	11,340	15,174	18,490
add32	1,000	3,242	5,143	6,265	9,971	10,577
airfoil1	1,000	2,543	5,105	6,645	11,959	12,591
bcsstk29	1,000	5,206	7,335	14,083	27,112	50,760
bcsstk33	1,000	5,640	7,333	13,793	27,657	54,830
Big	1,000	5,013	6,727	12,372	21,124	29,173
CCC5	1,000	0,634	0,926	1,204	1,581	1,989
Crack	1,000	5,192	6,579	11,927	19,161	23,668
Data	1,000	4,295	5,981	6,740	12,104	13,745

Tabela 10 – *Speedups* comparados com a configuração 01n01t

(conclusão)

Grafo	Configurações de execução					
	01n01t	01n16t	02n16t	04n16t	08n16t	16n16t
fe_rotor	1,000	4,341	6,192	11,838	22,782	40,874
fe_tooth	1,000	4,507	6,214	11,682	22,211	40,618
G124.04	1,000	0,723	1,260	2,210	3,283	4,207
G124.16	1,000	3,185	3,388	3,515	5,546	8,540
G250.08	1,000	5,399	4,013	8,972	9,344	10,936
G500.02	1,000	3,401	5,334	5,450	5,584	10,754
Grid32x32	1,000	1,093	1,690	2,924	3,831	4,588
memplus	1,000	6,821	7,659	15,026	29,251	54,004
nasa1824	1,000	6,276	7,359	12,825	15,481	18,309
nasa2146	1,000	5,581	7,222	12,318	23,593	28,229
Stufe	1,000	1,384	1,645	2,594	3,742	5,261
U500.20	1,000	3,325	3,960	4,297	6,011	9,332
U500.40	1,000	3,749	6,102	10,691	9,774	14,412
U1000.20	1,000	3,943	5,446	8,061	7,928	12,962
U1000.40	1,000	5,153	6,727	12,140	14,056	22,269
vibrobox	1,000	6,820	7,071	13,735	27,698	54,720
whitaker	1,000	4,566	6,109	10,963	15,520	23,436
wing_nodal	1,000	6,958	7,534	15,148	28,210	48,909
Média	**1,000**	**4,170**	**5,420**	**9,345**	**15,182**	**24,085**

Os valores de *speedups* médios obtidos com as implementações das paralelizações em *threads* e em diversos nós do *cluster* são apresentados no Gráfico 13. Os valores de *speedup* têm como base a configuração serial máxima (01n01t), que é executada em apenas um nó processador do *cluster* utilizando apenas uma *thread*.

Alguns grafos apresentaram *speedups* consideráveis, como no caso dos grafos *bcsstk33*, *memplus* e *vibrobox*, com valores acima de 54.

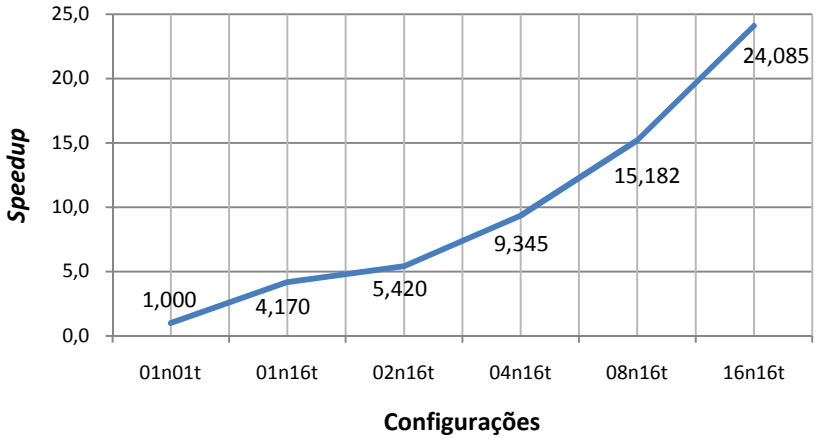


Gráfico 13 – *Speedups* médios comparados com a configuração 01n01t

Uma análise dos tempos de paralelização apenas das configurações que utilizaram 16 *threads*, excluída a configuração puramente serial 01n01t, apresenta uma relação entre o aumento do número de nós processadores utilizados no *cluster* e os tempos médios de execução do algoritmo particionador que proporciona uma análise mais fiel dos ganhos de tempo ao se utilizar o *cluster*, já que dependem apenas do número de nós processadores utilizados pelo mesmo. As configurações utilizadas foram a partir da 01n16t. Isso pode ser visto no Gráfico 14.

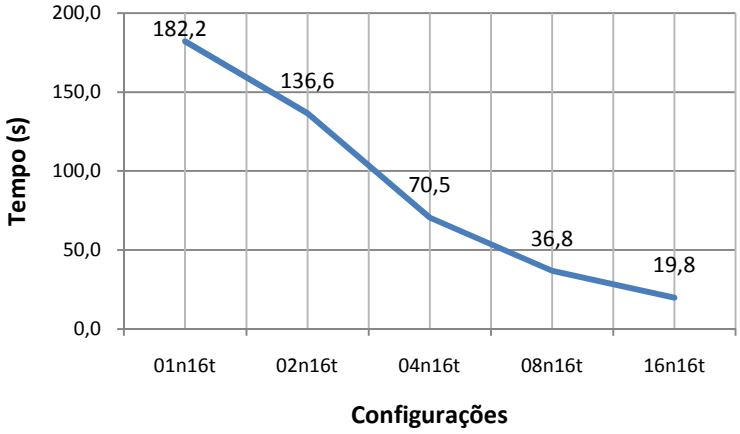


Gráfico 14 – Tempos médios de execução utilizando 16 *threads*

Os *speedups* obtidos pelas configurações comparados com a configuração de execução 01n16t são apresentados na Tabela 11.

Tabela 11 – *Speedups* comparados com a configuração 01n16t

(continua)

	Configurações de execução				
Grafo	**01n16t**	**02n16t**	**04n16t**	**08n16t**	**16n16t**
144	1,000	1,345	2,613	5,122	10,062
3elt	1,000	1,525	2,993	4,099	4,282
598a	1,000	1,333	2,685	5,055	9,018
add20	1,000	1,431	2,477	3,315	4,039
add32	1,000	1,586	1,932	3,075	3,262
airfoil1	1,000	2,007	2,613	4,702	4,951
bcsstk29	1,000	1,409	2,705	5,207	9,750
bcsstk33	1,000	1,300	2,446	4,904	9,722
big	1,000	1,342	2,468	4,213	5,819
CCC5	1,000	1,460	1,899	2,493	3,136
crack	1,000	1,267	2,297	3,690	4,558
data	1,000	1,392	1,569	2,818	3,200
fe_rotor	1,000	1,426	2,727	5,248	9,416

Tabela 11 – *Speedups* comparados com a configuração 01n16t

(conclusão)

Grafo	Configurações de execução				
	01n16t	**02n16t**	**04n16t**	**08n16t**	**16n16t**
fe_tooth	1,000	1,379	2,592	4,928	9,012
G124.04	1,000	1,742	3,054	4,538	5,815
G124.16	1,000	1,064	1,104	1,742	2,681
G250.08	1,000	0,743	1,662	1,731	2,025
G500.02	1,000	1,568	1,602	1,642	3,162
Grid32x32	1,000	1,546	2,676	3,505	4,198
;memplus	1,000	1,123	2,203	4,289	7,918
nasa1824	1,000	1,173	2,043	2,467	2,917
nasa2146	1,000	1,294	2,207	4,227	5,058
stufe	1,000	1,189	1,874	2,704	3,801
U500.20	1,000	1,191	1,292	1,808	2,806
U500.40	1,000	1,628	2,852	2,607	3,845
U1000.20	1,000	1,381	2,044	2,011	3,288
U1000.40	1,000	1,305	2,356	2,728	4,321
vibrobox	1,000	1,037	2,014	4,061	8,024
whitaker	1,000	1,338	2,401	3,399	5,133
wing_nodal	1,000	1,083	2,177	4,055	7,029
Média	**1,000**	**1,354**	**2,253**	**3,546**	**5,408**

Ao se compararem os valores de *speedups* médios obtidos com as implementações das paralelizações em *threads* apenas com a configuração de execução 01n16t, nota-se uma maior linearidade dos *speedups* com relação ao número crescente de nós processadores do *cluster* utilizados para a execução do algoritmo paralelo. Essa análise é apresentada no Gráfico 15.

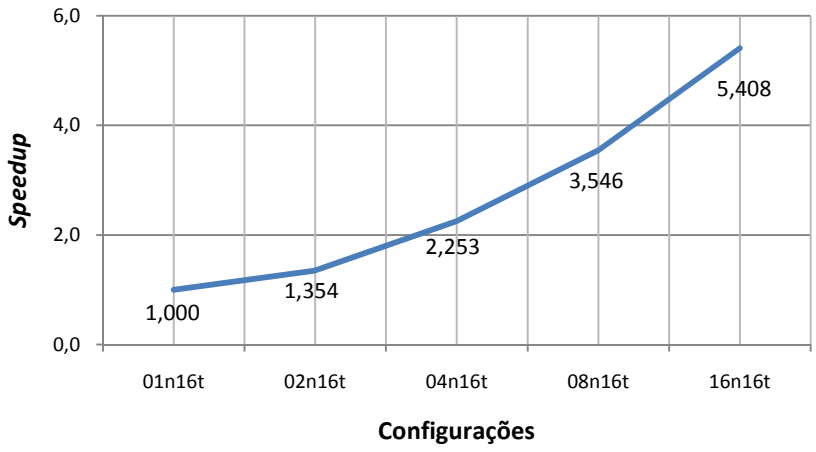


Gráfico 15 – *Speedup* médio comparado com a configuração 01n16t

A configuração utilizada no *cluster* foi de 16 nós processadores. Na teoria, os *speedups* obtidos deveriam ser de 16 ao se compararem os tempos de execução das configurações 01n16t e 16n16t. Porém, na prática, tais valores não são atingíveis, sendo os melhores valores encontrados $10{,}062$ para o grafo *144* e $9{,}750$ para o grafo *bcsstk29*.

A Tabela 12 apresenta os números obtidos na análise de desempenho da Eficiência Paralela.

Tabela 12 – Eficiências Paralelas

	Configurações de execução			
Grafo	**02n16t**	**04n16t**	**08n16t**	**16n16t**
144	0,673	0,653	0,640	0,629
3elt	0,762	0,748	0,512	0,268
598a	0,666	0,671	0,632	0,564
add20	0,716	0,619	0,414	0,252
add32	0,793	0,483	0,384	0,204
airfoil1	1,004	0,653	0,588	0,309
bcsstk29	0,704	0,676	0,651	0,609
bcsstk33	0,650	0,611	0,613	0,608
big	0,671	0,617	0,527	0,364
CCC5	0,730	0,475	0,312	0,196
crack	0,634	0,574	0,461	0,285
data	0,696	0,392	0,352	0,200
fe_rotor	0,713	0,682	0,656	0,588
fe_tooth	0,689	0,648	0,616	0,563
G124.04	0,871	0,764	0,567	0,363
G124.16	0,532	0,276	0,218	0,168
G250.08	0,372	0,415	0,216	0,127
G500.02	0,784	0,401	0,205	0,198
Grid32x32	0,773	0,669	0,438	0,262
memplus	0,561	0,551	0,536	0,495
nasa1824	0,586	0,511	0,308	0,182
nasa2146	0,647	0,552	0,528	0,316
stufe	0,594	0,469	0,338	0,238
U500.20	0,595	0,323	0,226	0,175
U500.40	0,814	0,713	0,326	0,240
U1000.20	0,691	0,511	0,251	0,205
U1000.40	0,653	0,589	0,341	0,270
vibrobox	0,518	0,503	0,508	0,501
whitaker	0,669	0,600	0,425	0,321
wing_nodal	0,541	0,544	0,507	0,439
Média	**0,677**	**0,563**	**0,443**	**0,338**

Os valores obtidos com a análise da eficiência paralela das configurações de execução são apresentados no Gráfico 16.

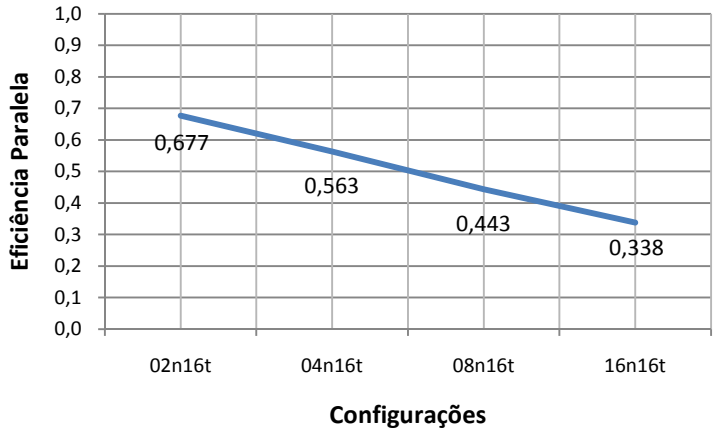


Gráfico 16 – Eficiência paralela média com 16 *threads*

No próximo capítulo são apresentadas as conclusões deste trabalho.

5 CONCLUSÕES

Este capítulo apresenta as conclusões acerca desta pesquisa.

O presente trabalho de dissertação de mestrado apresentou uma implementação paralela de heurísticas de particionamento de grafos, em especial motivada pelos problemas reportados na literatura de processos de simulação nas áreas de exploração e produção de petróleo e gás.

Em seu trabalho, Bonatto (2010) propôs algoritmos heurísticos para particionamento de grafos que obtiveram bons resultados com relação à diminuição dos cortes se comparados com aplicativos como o *METIS* e *CHACO*.

Essas heurísticas propostas baseiam-se em uma configuração *multistart*, ou seja, os algoritmos particionam os grafos diversas vezes e retornam como resultado o menor corte dentre todos os encontrados. Tal característica dessa configuração conduz à seguinte lógica: quanto mais resultados disponíveis, maior a probabilidade de se encontrar os menores cortes.

As heurísticas propostas por Bonatto (2010) foram implementadas para execução serial. Um aumento excessivo de iterações de execução na tentativa de se melhorar os cortes por meio das configurações *multistart* poderia inviabilizar os tempos de execução de particionamento dos grafos. A proposta de paralelização das heurísticas de Bonatto (2010) por este trabalho contornou esse problema.

Com o aumento da quantidade de iterações de 1.000 da execução serial de Bonatto (2010) para 256.000 da execução paralela deste trabalho, em sua maioria os cortes dos grafos foram diminuídos, sem o problema do excessivo aumento do tempo de execução, já que as iterações foram executadas de forma paralela tanto pelos *cores* dos processadores quanto pelos nós do *cluster*.

A paralelização do algoritmo não apenas utilizando a configuração de vários nós em um *cluster* de computadores, mas também as várias *threads* que um nó processador

multiprocessado/multinúcleo pode executar fez com que os ganhos de tempo com relação à execução serial fossem bem expressivos.

A análise de *speedup* foi feita com relação a dois tipos de configurações seriais: uma puramente serial, utilizando apenas um nó *monothreaded* e a outra utilizando um nó processador, porém, *multithreaded* com utilização de 16 *threads*. O fato de se ter utilizado 16 *threads* de execução em vez de 1 *thread* já elevou o desempenho em um fator médio de $4{,}170$. A partir dessa configuração de execução, os ganhos de *speedup* foram consequência do aumento do número de nós processadores no *cluster*, com ganhos médios de aumento de desempenho entre $5{,}420$ para dois nós no *cluster* e $24{,}085$ para dezesseis nós processadores.

Tais valores mostram que tanto a paralelização utilizando-se *threads* quanto a paralelização a partir do aumento do número de nós contribuíram de forma expressiva para o ganho de tempo de execução do algoritmo particionador paralelo.

As eficiências paralelas médias do algoritmo paralelo ficaram entre $0{,}677$ para a configuração com dois nós executando 16 threads e $0{,}338$ para a configuração com dezesseis nós executando também 16 threads, o que mostra uma ocupação efetiva do uso dos processadores entre $67{,}7\%$ e $33{,}8\%$ durante as execuções paralelas.

Esses valores demonstram o que ocorre realmente na prática ao se utilizar computadores paralelos de memória distribuída: nunca o processador dedica-se 100% à execução do algoritmo paralelo apenas. Dessa forma, ao aumentar-se o número de nós do *cluster*, os processadores de cada nó passam a depender mais de outros processadores do conjunto, aguardando, por exemplo, por entradas e saídas de nós do *cluster*, ficando, dessa forma, mais ociosos com relação ao processamento do algoritmo paralelo em si.

Os Tipos Abstratos de Dados propostos na implementação paralela do algoritmo propiciaram um acesso eficaz aos dados. Numa primeira implementação, a classe Grafo foi representada por uma lista de adjacências. Apesar se não terem sido realizados testes comparativos, assim que a representação do grafo foi alterada

para o TAD proposto nesse trabalho, o *CSRG*, os ganhos de desempenho das execuções eram facilmente perceptíveis.

A linguagem *Java* e a biblioteca de passagem de mensagens *MPJ Express* utilizadas na implementação deste trabalho permitiram uma utilização de classes para a declaração dos TADs com muita clareza, além de todas as outras características que uma linguagem de programação orientada a objetos de alto nível pode proporcionar.

A biblioteca de passagem de parâmetros *MPJ Express* possui a característica de aceitar objetos como parâmetros em seus comando de *send* e *receive*, tornando muito natural a implementação da troca de mensagens juntamente com o *Java*, permitindo, por exemplo, que objetos *subsets* ou grafos fossem enviados sem necessitar de nenhuma transformação em outro tipo de objeto.

Durante a implementação do algoritmo paralelo, algumas melhorias foram propostas por este trabalho. Dentre elas, a que provocou diminuição perceptível nos cortes dos grafos foi a alteração do Método de Melhoramento, que permitiu uma melhora média de 7,14% dos cortes com relação ao Método de Melhoramento proposto originalmente por Bonatto (2010).

As outras melhorias propostas neste trabalho, tais como divisão exclusiva das *seeds* entre os nós processadores, rotina de melhoramento do corte na iteração *versus* execução, cortes máximos para execução do melhoramento na iteração e estratégia de variação do alfa não surtiram efeito prático e expressivo na diminuição dos cortes dos grafos.

O próximo capítulo traz diversas sugestões para trabalhos futuros.

6 SUGESTÕES PARA TRABALHOS FUTUROS

Este capítulo apresenta sugestões para a realização de trabalhos futuros.

Como sugestão, assim como citado por Bonatto (2010), a implementação de métodos multiníveis para contração dos grafos antes do particionamento melhoraria ainda mais os tempos de particionamento, já que mesmo que os computadores paralelos de memória distribuída sejam mais rápidos que os computadores seriais, dada a característica *multistart* do algoritmo de particionamento, quanto mais iterações sejam executadas, maiores a probabilidades de se diminuir os cortes dos grafos em seus particionamentos. Essa tendência de aumentar muito o número de iterações pode também comprometer os tempos de execução do algoritmo, mesmo sendo executado em computadores paralelos.

Além disso, os cortes que foram usados como comparação neste trabalho foram obtidos a partir do trabalho de Bonatto (2010), utilizando particionadores seriais tais como o *METIS* e o *CHACO*, além do próprio particionador serial proposto por Bonatto (2010). Para efeitos de comparação não só de melhoria de cortes, mas também de ganhos de tempo de execução, essa implementação paralela pode ser comparada com outros particionadores paralelos, como por exemplo, o *ParMETIS*.

Outra sugestão para trabalhos futuros é que seja feita uma comparação da implementação desse trabalho em *Java* com um implementação em *C/C++* ou outra linguagem puramente compilada para efeitos de desempenho.

O tipo de representação utilizado pelos grafos nesse trabalho mostrou-se muito mais eficiente do que a proposta inicial do mesmo, que seria utilizar uma representação de lista de adjacências. Apesar de não ter sido feito nenhum teste efetivo para se comprovar isso, o fato do TAD Grafo ter sido alterado para a representação *CSRG* proposta nesse trabalho já apresentou visíveis melhoras de desempenho.

7 REFERÊNCIAS

AL-NASRA, M.; NGUYEN, D. **An Algorithm for Domain Decomposition in Finite Element Analysis**. Computer and Structures, v. 39, p. 277-289, 1991.

AZIZ, K.; SETTARI, A. **Petroleum Reservoir Simulation**. Applied Science Publishers. London, p. 450-468, 1979.

BENLIC, U.; HAO, J.-K.; **Hybrid Metaheuristics for the Graph Partitioning Problem**. Studies in Computational Intelligence, v. 434, p. 157-185, 2013.

BONATTO, R. S. **Algoritmos Heurísticos para Partição de Grafos Com Aplicação em Processamento Paralelo.** 2010. 94 f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Espírito Santo, Vitória, 2010.

BONATTO, R. S.; AMARAL, A. R. S. **Algoritmo Heurístico para Partição de Grafos com Aplicação em Processamento Paralelo**". In: CONGRESSO DA SOCIEDADE BRASILEIRA DE PESQUISA OPERACIONAL, 42, Bento Gonçalves, 2010.

BONDY, J. A.; MURTY, S. R. **Graph Theory with Applications**. 5. ed. New York: Elsevier Science Publishing, 1982.

BOOST. **Compressed Sparse Row Graph**. 2013. Disponível em: < http://www.classes.cs.uchicago.edu/archive/2013/fall/51025-1/boost_1_50_0/libs/graph/doc/compressed_sparse_row.html>. Acesso em: 14 novembro 2013.

BOSTON UNIVERSITY. **Speedup Ratio and Parallel Efficiency**. 2013. Disponível em: <http://www.bu.edu/tech/about/research/training/online-tutorials/matlab-pct/scalability/>. Acesso em: 29 novembro 2013.

BUI, T.N.; MOON, B. R. **Genetic Algorithm and Graph Partiotining**. IEEE Transactions and Computers, n. 45, p. 841-855, 1996.

BUYYA, R. **High Performance Cluster Computing**: Architectures and Systems. v. 1, New Jersey: Prentice Hall, 1999. 881 p.

CORDAZZO, J. **Simulação de Reservatórios de Petróleo Utilizando o Método Ebfvm e Multigrid Algébrico.** 2006. 250 f. Tese (Doutorado em Engenharia Mecânica) – Programa de Pós-Graduação em Engenharia Mecânica, Universidade Federal de Santa Catarina, Florianópolis, 2006.

CORNELL, G.; HORSTMANN, C. **Core *Java* 2:** fundamentos. 7. ed. Rio de Janeiro: Alta Books, 2005. 368 p.

COSSÉ, R. **Basics of Reservoir Engineering:** Oil and Gas Field Development Techiniques. Paris: Éditions Technip, 1993. 151 p.

CULLER, D; SINGH, J. P.; GUPTA, A. **Parallel Computer Architecture**: a

Hardware/Software Approach. San Francisco: Morgan Kaufmann Publishers, Inc., 1998. 1056 p.

DEITEL, H. M.; DEITEL, P. J. ***Java*, como programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2005. 1110 p.

DIESTEL, R. **Graph Theory**. New York: Springer-Verlag, 2000. 312 p.

DORNELES, R. V. **Particionamento de Domínio e Balanceamento de Carga no Modelo HIDRA.** 2003. 136 f. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre.

ECONOMIDES, M. J.; HILL, A. D.; EHLIG-ECONOMIDES, C. **Petroleum Production Systems**. New Jersey: Prentice Hall, 1994. 611 p.

FANCHI, J. R. **Principles of Applied Reservoir Simulation**. 3. ed. Burlington: Elsevier, 2006. 516 p.

FIDUCCIA, C.; MATTHEYESES, R. **A Linear-Time Heuristic for Improving Network Partitions.** 19th IEEE Design Automation Conference, p. 175-181, 1982.

FJÄLLSTRÖM, P.-O. **Algorithms for Graph Partitioning: a survey**. Linkoping Electronic Articles in Computer and Information Science, v. 3, n. 10, 1998.

GALANTE, G. **Métodos *Multigrid* Paralelos em Malhas Não Estruturadas Aplicados à Simulação de Problemas de Dinâmica de Fluidos Computacional e Transferência de Calor.** 2006. 130 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2006.

GEBALI, F. **Algorithms and Paralell Computing**. Victoria: Wiley, 2011. 341 p.

GUEDES, M. J. M. **Paralelização da Resolução de EDPS pelo Método Hopscotch Utilizando Refinamento Adaptativo e Balanceamento Dinâmico de Carga.** 2009. 122 f. Tese (Doutorado em Modelagem Computacional) – Programa de Pós-Graduação em Computação, Universidade Federal Fluminense, Niterói, 2009.

GUO, B.; LYONS, W. C.; GHALAMBOR, A. **Petroleum Production Engineering:** A Computer-Assisted Approach. Burlington: Elsevier, 2007. 287 p.

HOPKINS, B.; WILSON, R. J. **The Truth about Königsberg**. The College Mathematics Journal, v. 35, n. 3, p. 198-207, 2004.

HORSTMANN, C. **Big *Java***. Porto Alegre: Bookman, 2004. 1125 p.

INFOWESTER. **Cluster:** conceito e características. 2013. Disponível em: <http://www.infowester.com/cluster.php>. Acesso em: 17 dezembro 2013.

INTERNATIONAL ENERGY AGENCY. **Key World Energy Statistics.** Paris. 2012. 81 p.

JOHNSON, D.S.; ARAGON, C.R.; MCGEOCH, L.A.; SCHEVON, C. **Optimization by Simulated Annealing: an Experimental Evaluation; Part I, Graph Partitioning** Oper. Res., n. 37, p. 865-892, 1989.

KARYPIS, G.; KUMAR, V. **A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs**. Siam J. Sci Comput., v. 20, n. 1, p. 359-392, 1998.

KERNIGHAN, B.W.; LIN, S. **An Efficient Heuristic Procedure for Partitioning Graphs**. The Bell System Technical Journal, p. 291-307, 1970.

KISCHINHEVSKY, M.; ROBAINA, D. ; GUEDES, M.; DRUMMOND, L. M. A.; SILVEIRA FILHO, O. T. **Solução Numérica de Equações Diferenciais Parciais Parabólicas empregando um Método Hopscotch com refinamento não-uniforme**. In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, 2005, São João Del Rei. Anais do Congresso Nacional de Matemática Aplicada e Computacional, 2005, v. 1, p. 1-5.

MARQUES, A. C. **Desenvolvimento de Modelo Numérico Utilizando o Método dos Volumes Finitos em Malhas Não-Estruturadas.** 2005. 93 f. Dissertação (Mestrado em Ciências em Engenharia Ambiental) – Programa de Pós-Graduação em Engenharia Ambiental, Universidade Federal do Espírito Santo, Vitória, 2005.

MATTOS, G. O. **Aspectos de Desempenho da Computação Paralela em *Clusters* e *Grids* para Processamento de Imagens.** 2008. 167 f. Tese (Doutorado em Engenharia Elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Pernambuco, Recife, 2008.

METAHEURISTICS NETWORK. **Project Summary**. 2013. Disponível em: <http://www.metaheuristics.net/index.php?main=1>. Acesso em: 16 setembro 2013.

MPJ EXPRESS PROJECT. **MPJ Express**. 2008. Disponível em: <http://mpj-express.org/>. Acesso em: 31 outubro 2013.

NEWMAN, A. *et al*. **Usando *Java*:** o guia de referência mais completo. Rio de Janeiro: Campus, 1997. 861 p.

OU, C-W.; RANKA, S. **SPRINT: Scalable Partitioning, Refinement, and INcremental partitioning Techniques**. Syracuse, 1997.

PADUA, D. **Encyclopedia of Parallel Computing**. Urbana: Springer, 2011. 2175 p.

PEACEMAN, D. W. **Fundamentals of Numerical Reservoir Simulation**. Amsterdam: Elsevier, 1977. 176 p.

PEREIRA, R. L. **Desenvolvimento de uma Biblioteca Eficiente Baseada em Sockets para Clusters e Cálculo de Tensões Induzidas em Linhas de Transmissão com Catenárias.** 2011. 70 f. Dissertação (Mestrado em Engenharia

Elétrica) – Universidade Federal do Pará, Belém, 2011.

PETROLEUM. **Petroleum composition**. 2013. Disponível em: <http://www.petroleum.co.uk/composition>. Acesso em: 14 julho 2013.

PITANGA, M. **Construindo supercomputadores com Linux**. 3. ed. Rio de Janeiro: Brasport, 2008. 358 p.

QIU, H.; HANCOCK, E. R. **Graph matching and clustering using spectral partitions**. Pattern Recognition, n. 39, p. 22-24, 2006.

RAUBER, T.; RÜNGER, G. **Parallel Programming for Multicore and Cluster Systems**. Springer, 2010. 455 p.

RIZZI, R. L. **Modelo Computacional Paralelo para a Hidrodinâmica e para o Transporte de Massa Bidimensional e Tridimensional.** 2002. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2002.

ROLLAND, E.; PIRKUL, H.; GLOVER, F. **Tabu Search for Graph Partitioning.** Ann. Oper. Res., n. 63, p. 209-232, 1996.

ROSA, A. J.; CARVALHO, R. S.; XAVIER, J. A. D. **Engenharia de Reservatórios de Petróleo**. Rio de Janeiro: Interciência, 2006. 832 p.

RUOHONEN, K. **Graph Theory**. Notas de aula, 2013.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. PWS Publishing Company, 1995. 547 p.

SCHAEFFER, S.E. **Graph Clustering**. Computer Science Review, n. I, p. 27-64, 2007.

SCHLOEGEL K.; KARYPIS G.; KUMAR V. **Graph partitioning for high performance scientific simulations**. CRPC Parallel Computing Handbook. Morgan Kaufmann, 2001.

SHEWCHUK, J. R. **Lecture notes on Delaunay mesh generation**. Technical Report. Department of Electrical Engineering and Computer Science. University of California at Berkley. 1999. 119 p.

SILVA, R. S. **Simulação de Escoamento Bifásico Oléo-Água em Reservatórios de Petroléo Usando Computadores Paralelos de Memória Distribuída.** 2008. 144 f. Tese (Doutorado em Ciências de Engenharia Civil) – Programa de Pós-Graduação em Engenharia Civil, Universidade Federal de Pernambuco, Recife, 2008.

SOARES, A. A. M. **Simulação de Reservatórios de Petróleo em Arquiteturas Paralelas com Memória Distribuída**. 2002. 119 f. Dissertação (Mestrado em Ciências em Engenharia Civil) – Universidade Federal de Pernambuco, Recife,

2002.

SOARES, A. A. M.; ARAÚJO, E. R. **Reservoir Simulations in Clusters of PCs**. Mecánica Computacional, v. XXI, p. 2967-2976, 2002.

SOUZA, J. N. M. **Modelagem e Simulação de Escoamento Multifásico em Dutos de Produção de Óleo e Gás Natural**. 2010. Tese (Doutorado em Tecnologia de Processos Químicos e Bioquímicos) – Escola de Química, Universidade Federal do Rio de Janeiro, 2010.

TAO, L.; ZHAO, C.; THULASIRAMAN, K.; SWAMY, M. N. S. **Simulated Annealing and Tabu Search Algorithms for Multiway Graph Partition**. Journal of Circuits, Systems and Computers, v. 2, n. 2, p. 159-185, 1992.

THOMAS, J. E. *et al*. **Fundamentos de Engenharia de Petróleo**. Rio de Janeiro: Interciência, 2001. 271 p.

UNIVERSIDADE FEDERAL DO CEARÁ. **Curso de Engenharia de Petróleo**. 2013. Disponível em: <http://www.petroleo.ufc.br/index.php?option=com_content&task=view&id=394&Itemid=56>. Acesso em: 14 julho 2013.

APÊNDICE A – ARQUIVO DE CONFIGURAÇÃO DO PARTICIONADOR

```
# Arquivo de configuração gerado em 19/11/2013 às 14:53:44
0 : Caminho completo do grafo (terminar com /) : /home/cluster/particionador/grafos/
1 : Nome do arquivo do grafo : 144
2 : Número de partições do grafo (2 ou mais, potência de 2) : 2
3 : Heurística da rotina CreateSubsetP a ser utilizada (1 / 2 / 3) : 3
4 : Dividir de forma exclusiva as seeds iniciais entre os nós (0 = não / 1 = sim) : 0
5 : Variação do alfa durante as iterações (0.0 a 1.0 / 2 = Equiprovável / 3 = Híbrida / 4 = Equiprovável Gulosa) : 3
6 : Número de iterações para calcular o melhor corte em cada nó (1 ou mais) : 100
7 : Endereço de acesso ao cluster : 10.20.30.40
8 : Porta de acesso ao cluster : 22
9 : Usuário de acesso ao cluster : cluster
10 : Senha de acesso ao cluster : senha
11 : Rank do nó root : 0
12 : Tag das mensagens : 0
13 : Size do cluster : 16
14 : Caminho do diretório dos arquivos de execução (terminar com /) : /home/cluster/particionador/
15 : Nome do JAR principal : ParticionadorParalelo.jar
16 : Nome do arquivo de configurações : configuracoes.txt
17 : Nome do script de execução : executa
18 : Nome do arquivo de nós do cluster : machines
19 : Modo de execução do MPJ (1 = Cluster / 2 = Multicore) : 1
20 : Outras opções da JVM : -Xmx1024m
21 : Caminho remoto dos arquivos de resultados e logs (terminar com /) : /home/cluster/particionador/logs/
22 : Nome do arquivo de resultados gerais : resultados
23 : Nome dos arquivos de logs de execução de cada nó : execucao_
24 : Nome dos arquivos de logs de comunicação de cada nó : comunicacao_
25 : Extensão padrão dos arquivos de resultados e logs : dat
26 : Usar rank ou nome do processador nos nomes dos arquivos remotos de cada nó (1 = Rank / 2 = Rank + Nome) : 2
27 : Tamanho da máscara do sequencial de mensagens nos arquivos remotos de logs de comunicação : 3
28 : Tamanho da máscara dos números dos nós no arquivo remoto de resultados de cada nó : 2
29 : Imprimir partições finais no arquivo de resultados gerais (0 = não / 1 = sim) : 0
30 : Imprimir progresso das execuções em cada nó nos arquivos de logs de execução (0 = não / 1 = sim) : 0
31 : Imprimir progresso das iterações de corte nos arquivos de logs de execução (0 = não / 1 = sim) : 0
32 : Imprimir progresso das comunicações em cada nó nos arquivos de logs de comunicação (0 = não / 1 = sim) : 0
33 : Imprimir tempos de execução de cada rodada no arquivo de resultados gerais (0 = não / 1 = sim) : 1
34 : Caminho local da pasta de arquivos de resultados gerais e logs (terminar com /) : /home/leonardo/logs_head/
35 : Aplicar a rotina de melhoramento do corte (0 = Não / 1 = Na Execução / 2 = Na Iteração / 3 = Na Iteração com Cortes) : 2
36 : Modo execução da rotina de melhoramento do corte (1 = Original / 2 = Modificado) : 2
37 : Número de threads para calcular o melhor corte em cada nó (1 ou mais) : 8
38 : Cortes máximos para aplicar o melhoramento na iteração (valores dos cortes, separar com ;) : 16500;
```

APÊNDICE B – ARQUIVO DE RESULTADOS DO PARTICIONADOR

```
####################################################################
#         Particionamento de Grafos - Resultados Gerais            #
####################################################################

Execução iniciada em 21/01/2014 às 13:25:35

Grafo: /home/clusteruser/particionador/grafos/144
Quantidade de vértices do grafo: 144649
Quantidade de arestas do grafo: 1074393
Grau mínimo do grafo: 4
Grau máximo do grafo: 26
Densidade do grafo: 0.0103%
Número de partições: 32
Heurística de criação das partições: 3
Variação do alfa: Híbrida
Divisão das seeds iniciais: Não
Melhoramento após o particionamento: Na Iteração
Tipo de melhoramento: Modificado
Quantidade de iterações por nó: 100
Quantidade de threads por nó: 16
Modo de execução do MPJ: Cluster
Quantidade de nós utilizados no cluster: 16
Total de iterações (size x threads x iterações por nó): 16 x 16 x 100 = 25600

Partições = 0 e 1 / Nós = 0 a 15 / Nó de menor corte = 12 / Valor do corte = 7747
Tempo médio de cada iteração da rodada 1: 00 hora(s), 00 minuto(s) e 06 segundo(s) = 6.9559 segundo(s)
Tempo total de execução da rodada 1: 00 hora(s), 11 minuto(s) e 35 segundo(s) = 695.5940 segundo(s)

Partições = 0 e 2 / Nós = 0 a 7 / Nó de menor corte = 3 / Valor do corte = 4470
Partições = 1 e 3 / Nós = 8 a 15 / Nó de menor corte = 9 / Valor do corte = 5847
Tempo médio de cada iteração da rodada 2: 00 hora(s), 00 minuto(s) e 02 segundo(s) = 2.4168 segundo(s)
Tempo total de execução da rodada 2: 00 hora(s), 04 minuto(s) e 01 segundo(s) = 241.6820 segundo(s)

Partições = 0 e 4 / Nós = 0 a 3 / Nó de menor corte = 2 / Valor do corte = 2103
Partições = 1 e 5 / Nós = 4 a 7 / Nó de menor corte = 6 / Valor do corte = 2334
Partições = 2 e 6 / Nós = 8 a 11 / Nó de menor corte = 9 / Valor do corte = 3259
Partições = 3 e 7 / Nós = 12 a 15 / Nó de menor corte = 12 / Valor do corte = 2908
Tempo médio de cada iteração da rodada 3: 00 hora(s), 00 minuto(s) e 00 segundo(s) = 0.8920 segundo(s)
Tempo total de execução da rodada 3: 00 hora(s), 01 minuto(s) e 29 segundo(s) = 89.1960 segundo(s)

Partições = 0 e 8 / Nós = 0 a 1 / Nó de menor corte = 0 / Valor do corte = 1858
Partições = 1 e 9 / Nós = 2 a 3 / Nó de menor corte = 3 / Valor do corte = 2005
Partições = 2 e 10 / Nós = 4 a 5 / Nó de menor corte = 4 / Valor do corte = 1877
Partições = 3 e 11 / Nós = 6 a 7 / Nó de menor corte = 6 / Valor do corte = 1510
Partições = 4 e 12 / Nós = 8 a 9 / Nó de menor corte = 9 / Valor do corte = 1563
Partições = 5 e 13 / Nós = 10 a 11 / Nó de menor corte = 11 / Valor do corte = 1927
```

```
Partições = 6 e 14 / Nós = 12 a 13 / Nó de menor corte = 13 / Valor do corte = 2151
Partições = 7 e 15 / Nós = 14 a 15 / Nó de menor corte = 14 / Valor do corte = 2639
Tempo médio de cada iteração da rodada 4: 00 hora(s), 00 minuto(s) e 00 segundo(s) = 0.3926 segundo(s)
Tempo total de execução da rodada 4: 00 hora(s), 00 minuto(s) e 39 segundo(s) = 39.2590 segundo(s)

Partições = 0 e 16 / Nós = 0 a 0 / Nó de menor corte = 0 / Valor do corte = 1069
Partições = 1 e 17 / Nós = 1 a 1 / Nó de menor corte = 1 / Valor do corte = 1093
Partições = 2 e 18 / Nós = 2 a 2 / Nó de menor corte = 2 / Valor do corte = 1160
Partições = 3 e 19 / Nós = 3 a 3 / Nó de menor corte = 3 / Valor do corte = 1359
Partições = 4 e 20 / Nós = 4 a 4 / Nó de menor corte = 4 / Valor do corte = 978
Partições = 5 e 21 / Nós = 5 a 5 / Nó de menor corte = 5 / Valor do corte = 1032
Partições = 6 e 22 / Nós = 6 a 6 / Nó de menor corte = 6 / Valor do corte = 1034
Partições = 7 e 23 / Nós = 7 a 7 / Nó de menor corte = 7 / Valor do corte = 1969
Partições = 8 e 24 / Nós = 8 a 8 / Nó de menor corte = 8 / Valor do corte = 817
Partições = 9 e 25 / Nós = 9 a 9 / Nó de menor corte = 9 / Valor do corte = 1221
Partições = 10 e 26 / Nós = 10 a 10 / Nó de menor corte = 10 / Valor do corte = 868
Partições = 11 e 27 / Nós = 11 a 11 / Nó de menor corte = 11 / Valor do corte = 1174
Partições = 12 e 28 / Nós = 12 a 12 / Nó de menor corte = 12 / Valor do corte = 1066
Partições = 13 e 29 / Nós = 13 a 13 / Nó de menor corte = 13 / Valor do corte = 1423
Partições = 14 e 30 / Nós = 14 a 14 / Nó de menor corte = 14 / Valor do corte = 1116
Partições = 15 e 31 / Nós = 15 a 15 / Nó de menor corte = 15 / Valor do corte = 793
Tempo médio de cada iteração da rodada 5: 00 hora(s), 00 minuto(s) e 00 segundo(s) = 0.2217 segundo(s)
Tempo total de execução da rodada 5: 00 hora(s), 00 minuto(s) e 22 segundo(s) = 22.1750 segundo(s)

Corte total do grafo: 62370

Tempo total de execução: 00 hora(s), 18 minuto(s) e 07 segundo(s) = 1,087.9340 segundo(s)

Execução finalizada em 21/01/2014 às 13:43:43
```

APÊNDICE C – TRECHO DO ARQUIVO DE LOG DE COMUNICAÇÃO

```
#########################################################
#         Log de Comunicação - head (rank = 0)          #
#########################################################

| 001 | Send | nó 00 -> nó 01 | Id da partição e partição 0 enviados do root para o nó 1
| 001 | Send | nó 00 -> nó 02 | Id da partição e partição 0 enviados do root para o nó 2
| 001 | Send | nó 00 -> nó 03 | Id da partição e partição 0 enviados do root para o nó 3
| 001 | Send | nó 00 -> nó 04 | Id da partição e partição 0 enviados do root para o nó 4
| 001 | Send | nó 00 -> nó 05 | Id da partição e partição 0 enviados do root para o nó 5
| 001 | Send | nó 00 -> nó 06 | Id da partição e partição 0 enviados do root para o nó 6
| 001 | Send | nó 00 -> nó 07 | Id da partição e partição 0 enviados do root para o nó 7
| 001 | Send | nó 00 -> nó 08 | Id da partição e partição 0 enviados do root para o nó 8
| 001 | Send | nó 00 -> nó 09 | Id da partição e partição 0 enviados do root para o nó 9
| 001 | Send | nó 00 -> nó 10 | Id da partição e partição 0 enviados do root para o nó 10
| 001 | Send | nó 00 -> nó 11 | Id da partição e partição 0 enviados do root para o nó 11
| 001 | Send | nó 00 -> nó 12 | Id da partição e partição 0 enviados do root para o nó 12
| 001 | Send | nó 00 -> nó 13 | Id da partição e partição 0 enviados do root para o nó 13
| 001 | Send | nó 00 -> nó 14 | Id da partição e partição 0 enviados do root para o nó 14
| 001 | Send | nó 00 -> nó 15 | Id da partição e partição 0 enviados do root para o nó 15
| 003 | Recv | nó 01 -> nó 00 | Menor corte = 7901 entre as partições 0 e 1 recebido do nó 1 para o root
| 003 | Recv | nó 02 -> nó 00 | Menor corte = 8360 entre as partições 0 e 1 recebido do nó 2 para o root
| 003 | Recv | nó 03 -> nó 00 | Menor corte = 8289 entre as partições 0 e 1 recebido do nó 3 para o root
| 003 | Recv | nó 04 -> nó 00 | Menor corte = 8179 entre as partições 0 e 1 recebido do nó 4 para o root
| 003 | Recv | nó 05 -> nó 00 | Menor corte = 8203 entre as partições 0 e 1 recebido do nó 5 para o root
| 003 | Recv | nó 06 -> nó 00 | Menor corte = 7794 entre as partições 0 e 1 recebido do nó 6 para o root
| 003 | Recv | nó 07 -> nó 00 | Menor corte = 7976 entre as partições 0 e 1 recebido do nó 7 para o root
| 003 | Recv | nó 08 -> nó 00 | Menor corte = 8247 entre as partições 0 e 1 recebido do nó 8 para o root
| 003 | Recv | nó 09 -> nó 00 | Menor corte = 7805 entre as partições 0 e 1 recebido do nó 9 para o root
| 003 | Recv | nó 10 -> nó 00 | Menor corte = 7809 entre as partições 0 e 1 recebido do nó 10 para o root
| 003 | Recv | nó 11 -> nó 00 | Menor corte = 8002 entre as partições 0 e 1 recebido do nó 11 para o root
| 003 | Recv | nó 12 -> nó 00 | Menor corte = 7747 entre as partições 0 e 1 recebido do nó 12 para o root
| 003 | Recv | nó 13 -> nó 00 | Menor corte = 8303 entre as partições 0 e 1 recebido do nó 13 para o root
| 003 | Recv | nó 14 -> nó 00 | Menor corte = 8291 entre as partições 0 e 1 recebido do nó 14 para o root
| 003 | Recv | nó 15 -> nó 00 | Menor corte = 8093 entre as partições 0 e 1 recebido do nó 15 para o root
| 004 | Send | nó 00 -> nó 01 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 1
| 004 | Send | nó 00 -> nó 02 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 2
| 004 | Send | nó 00 -> nó 03 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 3
| 004 | Send | nó 00 -> nó 04 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 4
| 004 | Send | nó 00 -> nó 05 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 5
| 004 | Send | nó 00 -> nó 06 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 6
| 004 | Send | nó 00 -> nó 07 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 7
| 004 | Send | nó 00 -> nó 08 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 8
| 004 | Send | nó 00 -> nó 09 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 9
| 004 | Send | nó 00 -> nó 10 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 10
| 004 | Send | nó 00 -> nó 11 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 11
| 004 | Send | nó 00 -> nó 12 | Nó com o menor corte = 12 entre as partições 0 e 1 enviado do root para o nó 12
```

APÊNDICE D – TRECHO DO ARQUIVO DE LOG DE EXECUÇÃO

```
#####################################################
#         Log de Execução - head (rank = 0)         #
#####################################################

Execução iniciada em 21/01/2014 às 13:25:35 (rank = 00 / nó head)

1/5 : Rodada iniciada no nó head (rank = 00)
1/5 : O root envia para os nós as ids e as partições iniciais da rodada
1/5 : Iniciado o cálculo local do menor corte com 100 iteração(ões) em 16 thread(s)
1/5 : Thread 2/16 / Menor cte final = 8119 / Cte médio ptc = 32,548.5801 / Melh média = 48.7047% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 13/16 / Menor cte final = 8385 / Cte médio ptc = 32,914.7383 / Melh média = 51.0624% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 14/16 / Menor cte final = 8304 / Cte médio ptc = 35,032.3711 / Melh média = 47.8576% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 6/16 / Menor cte final = 8371 / Cte médio ptc = 35,849.7891 / Melh média = 51.7887% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 5/16 / Menor cte final = 9229 / Cte médio ptc = 35,446.8789 / Melh média = 51.6759% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 15/16 / Menor cte final = 8938 / Cte médio ptc = 37,236.1992 / Melh média = 50.3070% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 4/16 / Menor cte final = 8597 / Cte médio ptc = 38,296.5586 / Melh média = 53.4524% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 11/16 / Menor cte final = 8237 / Cte médio ptc = 40,382.3086 / Melh média = 54.2930% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 9/16 / Menor cte final = 9070 / Cte médio ptc = 39,518.5000 / Melh média = 55.1600% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 1/16 / Menor cte final = 8798 / Cte médio ptc = 41,630.6289 / Melh média = 53.5627% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 8/16 / Menor cte final = 8545 / Cte médio ptc = 43,306.2109 / Melh média = 55.5519% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 12/16 / Menor cte final = 9168 / Cte médio ptc = 43,915.1211 / Melh média = 54.0838% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 16/16 / Menor cte final = 8079 / Cte médio ptc = 41,447.8711 / Melh média = 53.2989% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 10/16 / Menor cte final = 9288 / Cte médio ptc = 44,236.8516 / Melh média = 55.1887% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 3/16 / Menor cte final = 8956 / Cte médio ptc = 48,032.3086 / Melh média = 57.7323% / Melh exec = 100/100 (100.0000%)
1/5 : Thread 7/16 / Menor cte final = 8150 / Cte médio ptc = 47,820.2383 / Melh média = 52.0416% / Melh exec = 100/100 (100.0000%)
1/5 : Corte particionado médio das threads = 39,850.9492
1/5 : O root recebe os menores cortes de todos os nós para verificar qual deles é o menor
1/5 : O root envia para todos os nós quem é o nó de menor corte
1/5 : Envio das ids das partições ao nó de menor corte que devem ser enviadas ao root
1/5 : Envio das partições de menor corte para o root
1/5 : Rodada finalizada no nó head (rank = 00)

2/5 : Rodada iniciada no nó head (rank = 00)
2/5 : O root envia para os nós as ids e as partições iniciais da rodada
2/5 : Iniciado o cálculo local do menor corte com 100 iteração(ões) em 16 thread(s)
2/5 : Thread 10/16 / Menor cte final = 5143 / Cte médio ptc = 17,834.8398 / Melh média = 44.6414% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 12/16 / Menor cte final = 4973 / Cte médio ptc = 17,750.5605 / Melh média = 44.2582% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 4/16 / Menor cte final = 4894 / Cte médio ptc = 18,987.5391 / Melh média = 42.3802% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 7/16 / Menor cte final = 4876 / Cte médio ptc = 18,015.7305 / Melh média = 42.6798% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 9/16 / Menor cte final = 4925 / Cte médio ptc = 18,892.8906 / Melh média = 45.0934% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 2/16 / Menor cte final = 5044 / Cte médio ptc = 19,491.0898 / Melh média = 43.1577% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 11/16 / Menor cte final = 4789 / Cte médio ptc = 18,844.1992 / Melh média = 44.0492% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 14/16 / Menor cte final = 4947 / Cte médio ptc = 20,046.2891 / Melh média = 46.1746% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 13/16 / Menor cte final = 4719 / Cte médio ptc = 21,430.6191 / Melh média = 45.9430% / Melh exec = 100/100 (100.0000%)
2/5 : Thread 3/16 / Menor cte final = 4812 / Cte médio ptc = 20,211.8496 / Melh média = 43.9287% / Melh exec = 100/100 (100.0000%)
```

APÊNDICE E – TELAS DO CONFIGURADOR DO PARTICIONADOR