Algoritmo BSP/CGM para Computação de Circuitos de Euler em Grafos

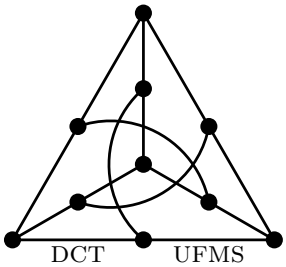
Claudia Yoshie Nasu

Dissertação de Mestrado

Orientação: Prof. Dr. Edson Norberto Cáceres

Área de Concentração: Ciência da Computação

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para a obtenção do título de mestre em Ciência da Computação.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Setembro de 2002

Algoritmo BSP/CGM para Computação de Circuitos de Euler em Grafos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Claudia Yoshie Nasu
e aprovada pela comissão julgadora.

Campo Grande, 12 de setembro de 2002.

Banca examinadora:

- Prof. Dr. Edson Norberto Cáceres - orientador (DCT-UFMS)
- Prof. Dr. Siang Wun Song (DCC-IME-USP)
- Prof. Dr. Henrique Mongelli (DCT-UFMS)

Aos meus pais

Agradecimentos

Ao meu orientador, Prof. Dr. Edson Norberto Cáceres, pela dedicação, pela paciência e, principalmente, por sua generosidade que tornou tudo isso possível. Obrigada.

Ao professor Henrique Mongelli, por estar sempre disposto a me ajudar, compartilhando suas experiências e seu conhecimento, e pelo constante incentivo.

Ao professor Marcelo Henriques de Carvalho, que tanto se dedicou para tornar realidade o Mestrado em Ciência da Computação da UFMS.

Aos colegas Amaury Antônio de Castro Junior e Liana Dessandre Duenha pelo apoio e pelo carinho. Nossos finais de semana depois que os créditos terminaram nunca mais foram os mesmos. Valeu a pena.

Aos professores do Departamento de Computação e Estatística da UFMS pela competência com que atuaram na minha formação. Seu entusiasmo em adquirir novos conhecimentos é sempre um incentivo ao meu aprendizado.

Ao Instituto de Computação da Universidade de Campinas (IC-UNICAMP) que nos permitiu utilizar a sua máquina paralela para a obtenção dos resultados práticos apresentados neste trabalho.

A todos que contribuíram para o desenvolvimento deste trabalho, direta ou indiretamente.

Resumo

Nesta dissertação descrevemos e implementamos um algoritmo paralelo utilizando o modelo BSP/CGM (*Bulk Synchronous Parallel/Coarse Grained Multicomputer*) para obtenção de circuitos de Euler em grafos. Este algoritmo é baseado no algoritmo proposto por Cáceres *et al* [CDSS92] que utiliza o modelo PRAM (*Parallel Random Access Machine*). Do nosso conhecimento, não há na literatura outros algoritmos paralelos em modelos de granularidade grossa para o problema de circuitos de Euler em grafos.

O algoritmo proposto foi implementado utilizando o padrão MPI (*Message Passing Interface*) e a linguagem C. O programa foi executado no *Beowulf* com 66 nós instalado no Instituto de Computação da UNICAMP. Os resultados obtidos com a implementação confirmaram os resultados teóricos da complexidade do algoritmo, o que é uma característica do modelo BSP/CGM.

Conteúdo

Capítulo 1

Introdução

Várias arquiteturas de computadores estão disponíveis atualmente para as mais variadas aplicações. A mais utilizada é a arquitetura idealizada por Von Neumann. Desde a sua criação até hoje muita coisa mudou: os processadores tornaram-se mais rápidos, a capacidade de memória aumentou, a transmissão de dados através das redes se tornou mais eficiente, entre outros fatores. Todo esse progresso foi acompanhado por uma diminuição impressionante do custo, o que tornou a tecnologia computacional mais acessível.

No entanto, está se tornando cada vez mais difícil aumentar o desempenho de máquinas que utilizam o modelo de Von Neumann. Aplicações que necessitam de um grande poder computacional como, por exemplo, a previsão de tempo, buscam suas soluções em computadores que utilizam outras arquiteturas. Contudo, melhorar o desempenho não depende somente do uso dos dispositivos de *hardware* mais rápidos e seguros, depende também de uma melhoria na arquitetura dos computadores e nas técnicas de processamento.

Desde que surgiu, há 20 anos, a computação paralela vem permitindo que problemas complexos sejam solucionados e aplicações de alto desempenho sejam desenvolvidas. O processamento paralelo pode ser definido com o uso de múltiplos processadores para executar diferentes partes do mesmo programa simultaneamente.

Vários fatores justificam a necessidade do processamento paralelo. A mais importante, que é a razão comercial do surgimento do processamento paralelo, é a capacidade de aumentar o desempenho e diminuir o tempo para se obter o resultado do problema. Outros fatores que contribuem para a grande evolução do processamento paralelo são:

- Limitação de velocidade das máquinas seqüenciais. Embora os proces-

sadores estejam se tornando cada vez mais rápidos, as leis da física limitam a velocidade de comunicação entre os componentes dos computadores, fazendo com que todo o ganho obtido no processamento perca-se na comunicação entre eles. A aproximação dos componentes poderia parecer uma solução, porém, existe um limite a partir do qual estes acabam interferindo entre si;

- Existência de problemas intrinsecamente paralelos.

Contudo, introduzir o conceito de processamento paralelo não é tão simples. Existem diversas formas de gerenciar os elementos de processamento e organizá-los em paralelo, além da necessidade de manter as informações que trafegam pela máquina coerentes. O tempo de execução de um programa seqüencial não varia muito quando a execução muda de uma máquina seqüencial para outra. Isto já não pode ser observado quando se trata de programas paralelos, pois um fator muito importante na elaboração de programas paralelos é a comunicação entre os elementos de processamento e esta varia muito de um sistema para outro.

Ao contrário da computação seqüencial que tem no RAM (*Random Access Machine*) - derivado do modelo de Von Neumann - um modelo padrão, a computação paralela enfrenta a falta de um modelo que seja amplamente aceito. Assim, devido a existência de vários modelos de computação paralela, a solução paralela proposta para um dado problema é totalmente dependente do modelo paralelo para o qual foi desenvolvida. Dentre os principais modelos de computação paralela destacamos: o modelo de memória compartilhada, o modelo de memória distribuída e os modelos realísticos.

O modelo de memória compartilhada PRAM (*Parallel Random Access Machine*) é uma extensão natural do modelo seqüencial. Consiste de um número de processadores, cada um com uma memória local e toda a comunicação é feita pela troca de informações através de uma memória global compartilhada. O modelo PRAM é um modelo de memória compartilhada que oferece simplicidade e independência de arquitetura, além de ser utilizado com sucesso para encontrar o paralelismo inerente dos problemas [Göt98]. Um dos problemas associados a este modelo é assumir que a comunicação através da memória compartilhada é tão eficiente quanto a computação local e que os processadores trabalham de maneira síncrona, o que torna o modelo não realístico, pois não reflete a realidade das máquinas paralelas atuais.

O modelo de memória distribuída corresponde a um conjunto de processadores com memórias locais e nenhuma memória global disponível. A comunicação é feita através de troca de mensagens entre os processadores.

Um fator muito importante neste modelo é a topologia de interconexão dos processadores que define com quais outros processadores cada processador possui um canal direto de comunicação. O principal problema deste modelo é a dependência existente entre as soluções propostas e a topologia para a qual foi desenvolvida.

Após adquirir experiência com as máquinas paralelas, os pesquisadores concluíram que a comunicação é o principal gargalo da computação paralela [Göt98]. Baseado nesta observação, foram propostos vários modelos que consideram os custos de comunicação como característica de um modelo de computação paralela o que torna os modelos mais realísticos.

Os principais modelos realísticos são o BSP (*Bulk Synchronous Parallel*) e o CGM (*Coarse Grained Multicomputer*). Os algoritmos BSP e CGM alternam a execução de superpassos/rodadas de computação local e comunicação. Entre os superpassos/rodadas os processadores são sincronizados. O objetivo dos algoritmos desenvolvidos para esses modelos é minimizar os custos de comunicação através da redução do número de superpassos/rodadas de comunicação.

Em função disso, a resolução de problemas que utilizem muita comunicação servem para analisar o comportamento dos modelos realísticos de computação paralela. Uma classe de problemas com essa característica é a de problemas que envolvem grafos.

O marco inicial da teoria de grafos é o conhecido problema das pontes de Königsberg. No rio Pregel, que atravessava a cidade de Königsberg, existiam duas ilhas formando quatro regiões distinguíveis de terra. Para que os moradores pudessem transitar facilmente de uma parte da cidade para outra, foram construídas sete pontes interligando as quatro regiões. Os moradores tentavam encontrar uma rota que cruzasse cada uma das sete pontes exatamente uma vez. Como todas as tentativas falhavam, muitos acreditavam que não existia tal rota. Este problema só foi resolvido em 1736 quando Euler encontrou uma solução matemática [CO93], mostrando que não existia tal trajeto.

Euler modelou o problema associando um vértice a cada região de terra e ligando dois vértices através de uma aresta sempre que havia uma ponte entre as duas regiões. Assim, o problema ficou reduzido a determinação de um ciclo contendo cada aresta do grafo exatamente uma vez. Um ciclo com tal característica ficou conhecido como circuito de Euler.

O algoritmo seqüencial para o problema possui complexidade $O(m+n)$, onde n é o número de vértices e m é o número de arestas do grafo [Szw88].

Este algoritmo é baseado em buscas no grafo e não pode ser implementado em paralelo porque não se conhece algoritmos paralelos eficientes para os problemas de busca em um grafo.

Vários algoritmos paralelos foram propostos para resolver o problema, entre os quais estão o proposto por Atallah-Vishkin [GR88] que no modelo PRAM CREW tem complexidade de tempo $O(\log^2 n)$ utilizando $O(m)$ processadores; e o proposto por Cáceres *et al* [CDSS92], que descreve uma implementação alternativa do primeiro, simplificando algumas estruturas nele utilizadas. Esse algoritmo também utiliza o modelo PRAM CREW e possui a mesma complexidade de tempo utilizando $O(\frac{m}{\log m})$ processadores.

Baseado no algoritmo proposto por Cáceres *et al* [CDSS92], desenvolvemos e implementamos um algoritmo paralelo no modelo BSP/CGM para encontrar circuitos de Euler em grafos. Não temos conhecimento da existência de algoritmos BSP/CGM para este problema. A existência de um algoritmo BSP/CGM para o problema da computação do circuito de Euler significa que problemas como emparelhamento máximo e coloração de determinadas classes de grafos também possuem algoritmos BSP/CGM.

O algoritmo para circuitos de Euler proposto no Capítulo 4 utiliza os resultados descritos em [CD99] (ordenação inteira), [CMS01] (ordenação inteira e *list ranking*), [LG99, LG00] (*list ranking*) e [CDF+97, Son99, DFC+02] (*list ranking*, árvore geradora e circuitos de Euler em árvores).

O algoritmo BSP/CGM proposto neste trabalho tem tempo de computação local $O(\frac{m+n}{p})$ e utiliza $O(\log p)$ rodadas de comunicação, respeitando a quantidade máxima de dados na comunicação imposta pelo modelo.

Este algoritmo foi implementado com o objetivo de comparar a complexidade teórica do algoritmo BSP/CGM com o tempo de execução obtido com a implementação, mostrando que o desempenho do algoritmo BSP/CGM é compatível com os resultados previstos no modelo teórico. A implementação utilizou a linguagem C e a biblioteca MPI (*Message Passing Interface*) e foi executada em um *Beowulf* com 66 processadores.

Vamos agora descrever como o presente trabalho está organizado. No Capítulo 2, apresentamos os conceitos em teoria dos grafos necessários para entender e desenvolver o problema e descrevemos o modelo computacional seqüencial RAM (*Random Access Machine*) e os principais modelos computacionais paralelos: PRAM (*Parallel Random Access Machine*), modelo distribuído, BSP (*Bulk Synchronous Parallel*) e CGM (*Coarse Grained Multicomputer*). Também descrevemos duas bibliotecas utilizadas para programação com troca de mensagens: PVM (*Parallel Virtual Machine*) e MPI

(*Message Passing Interface*), sendo que a biblioteca MPI será utilizada na implementação.

O Capítulo 3 descreve o problema de circuitos de Euler e algumas de suas aplicações. O capítulo ainda descreve alguns algoritmos para resolver esse problema encontrados na literatura. O primeiro é um algoritmo para o modelo seqüencial. Os outros dois algoritmos utilizam o modelo paralelo PRAM.

O Capítulo 4 contém a proposta de uma solução paralela utilizando o modelo BSP/CGM. Apresentamos os principais passos do algoritmo e comentamos os algoritmos BSP/CGM que implementam as subrotinas utilizadas pelo algoritmo, como ordenação, duplicação recursiva, árvore geradora e Euler Tour em árvores. Também analisamos a complexidade e a corretude do algoritmo proposto.

No Capítulo 5, descrevemos os detalhes da implementação do algoritmo BSP/CGM proposto no Capítulo 4 utilizando a interface MPI e a linguagem C. Apresentamos os tempos obtidos com a execução do programa e, ao final do capítulo, uma análise dos resultados. No Capítulo 6, apresentamos as conclusões e comentários finais.

No Apêndice A, descrevemos as principais funções de comunicação do padrão MPI. O Apêndice B contém o código-fonte do programa que implementa o algoritmo BSP/CGM descrito no Capítulo 4 e do programa utilizado para gerar os grafos de entrada.

Capítulo 2

Notação e Preliminares

2.1 Computação Paralela

Os computadores paralelos começaram a surgir na década de 80. As principais motivações da computação paralela são a busca de soluções cada vez mais rápidas e a necessidade de resolver-se problemas com grande quantidade de informação, que aparecem em várias áreas tais como: previsão de tempo, processamento de imagens, inteligência artificial, modelagem e simulação de grandes sistemas, etc.

A definição de um computador paralelo dada por JáJá [JáJ92] é a seguinte:

Definição 1 *Um **computador paralelo** é simplesmente uma coleção de processadores, tipicamente do mesmo tipo, interconectados de uma certa maneira a permitir a coordenação de suas atividades e a troca de dados.*

Segundo Mongelli e Terada [MT95], com a computação paralela a criação de algoritmos paralelos modifica-se sensivelmente. As estratégias utilizadas em algoritmos seqüenciais para resolver um problema não servem totalmente para a criação de algoritmos paralelos para o mesmo problema. Os recursos disponíveis são maiores e devemos aproveitá-los ao máximo. Na criação dos algoritmos devemos considerar a natureza dos problemas. Alguns são inerentemente paralelizáveis; outros apresentam um paralelismo menos transparente, exigindo estratégias mais elaboradas na montagem dos algoritmos; e existem aqueles em que o grau de paralelismo é muito baixo ou inexistente. Além disso, no desenvolvimento dos algoritmos temos que considerar a arquitetura paralela a ser utilizada, pois ela afeta o desempenho dos algoritmos.

Antes de tratarmos dos modelos computacionais, introduziremos alguns conceitos básicos de processamento paralelo e também de como deve ser feita a programação utilizando processamento paralelo.

2.1.1 Níveis de Paralelismo

A computação paralela divide uma tarefa em subtarefas (processos) que podem ser executadas simultaneamente por diferentes processadores. O nível de paralelismo (ou granularidade) está relacionado com o tamanho médio das subtarefas executadas pelos processadores. Wilkinson e Allen [WA99] classificam a granularidade como fina, média e grossa.

Na **granularidade grossa**, cada processo contém um grande número de instruções seqüenciais e gasta um tempo considerável para executá-las. Na **granularidade fina**, o processo consiste de poucas instruções, ou até mesmo uma única instrução. A **granularidade média** representa um grupo intermediário. Geralmente, desejamos aumentar a granularidade para reduzir os custos da comunicação entre os processos, principalmente em ambientes de troca de mensagens, mas devemos tomar cuidado com o aumento da granularidade pois ela reduz o número de processos concorrentes e o nível de paralelismo [WA99].

A granularidade não está envolvida diretamente com a otimização de algoritmos paralelos, porém as tendências atuais de máquinas paralelas incentivam o uso de algoritmos com granularidade grossa, com o intuito de minimizar a comunicação e aumentar a eficiência.

2.1.2 Desempenho

Um dos fatores que justifica a necessidade de processamento paralelo é a capacidade de aumentar a velocidade de processamento. Com esse aumento de processamento há também um aumento da velocidade com que os processadores chegam a solução. Para quantificar esse aumento utiliza-se diferentes parâmetros, entre eles *speedup* e eficiência.

O ***speedup*** (Sp) pode ser definido como o aumento da velocidade observado quando se executa um algoritmo paralelo em um computador paralelo em relação ao seqüencial, ou seja, quão mais rápida é a execução paralela.

$$S_p = \frac{t_1}{t_p}, \text{ onde:}$$

t_1 = tempo para executar o melhor algoritmo seqüencial

t_p = tempo para executar o algoritmo paralelo utilizando p processadores

O caso ótimo é obtido quando $S_p = p$, ou seja, na medida em que se aumenta o número de processadores, aumenta-se diretamente a velocidade de processamento.

A **eficiência** do processador mede a utilização efetiva dos p processadores e varia entre 0 (0%) e 1 (100%). Dificilmente obtém-se eficiência igual a 1 pois sempre haverá perdas na paralelização do algoritmo, com a sobrecarga de comunicação e no sincronismo entre os processadores.

2.1.3 Acesso à Memória

Para coordenar as tarefas dos vários elementos de processamento (processadores) que trabalham na solução do mesmo problema é necessário alguma forma de comunicação entre eles para:

- Transportar informações e dados entre os processadores;
- Sincronizar as tarefas.

A forma como os processadores se comunicam e se sincronizam depende do modo de acesso à memória. O acesso à memória significa o modo, do ponto de vista do programador, pelo qual as informações serão armazenadas e afeta diretamente como os programas paralelos são escritos.

Arquiteturas com Memória Compartilhada

Neste tipo de arquitetura, tem-se múltiplos processadores operando independentemente, mas compartilhando o mesmo espaço de endereçamento, ou seja, um único bloco de memória. Somente um processador pode acessar a memória por vez, por isso, deve haver por parte do programador um controle para manter os dados consistentes. Não deve ser permitido que enquanto um processador esteja lendo um dado, um outro processador escreva na mesma localização, isto é feito utilizando mecanismos como semáforos e monitores.

Podemos citar duas vantagens da memória compartilhada:

- A velocidade para a troca de dados entre as tarefas é igual à velocidade de acesso à memória, ou seja, é rápido;

- Simplicidade, o que permite uma utilização eficiente.

As desvantagens são:

- A memória pode tornar-se o gargalo do sistema;
- Aumento do número de processadores implica em perda de desempenho se a estrutura de acesso a memória não possuir *bandwidth* necessário;
- Usuário é responsável pela sincronização.

Arquiteturas com Memória Distribuída

Nestas arquiteturas, tem-se múltiplos processadores independentes, mas cada um possui uma memória local própria que é diretamente acessível somente pelo processador. Os dados podem ser compartilhados através de uma rede de comunicação utilizando troca de mensagens. Neste modo de acesso, o programador deve também preocupar-se com a sincronização, que é obtida através da troca de mensagens.

Como vantagens tem-se:

- A memória aumenta proporcionalmente com o aumento do número de processadores;
- Cada processador pode acessar rapidamente sua própria memória, sem interferências.

Como desvantagem:

- Usuário é responsável por enviar e receber dados entre os processos alocados em elementos de processamento (processador e memória) distintos, o que implica em controle de fluxo, controle sobre mensagens perdidas e *bufferização*.

2.2 Modelos Computacionais

Segundo Almasi e Gottlieb [AG94], existem dois aspectos que caracterizam um modelo computacional. No aspecto operacional, um modelo computacional especifica que tipo de algoritmo pode ser projetado. O modelo computacional descreve, sem considerar detalhes de tecnologia e *hardware*, o que o

computador pode fazer, ou seja, quais ações primitivas ele é capaz de executar.

O outro aspecto, mais matemático, de um modelo computacional permite a análise de um algoritmo. Isto significa que o modelo fornece o custo associado ao algoritmo. Tradicionalmente, estes custos incluem o tempo gasto para executar as operações e o espaço necessário para armazenar os dados. Quando consideramos a computação paralela, outros custos devem ser analisados como o número de processadores e a comunicação necessária entre eles.

De acordo com Browne [AG94], os cinco atributos seguintes devem ser especificados em um modelo computacional que inclui o paralelismo (os três primeiros atributos também podem ser considerados para modelos seqüenciais):

1. As **unidades primitivas** de computação (tipos de dados e operações definidas pelo conjunto de instruções);
2. O **mecanismo de controle** que seleciona as unidades primitivas em que o problema foi particionado para serem executadas;
3. O **mecanismo de dados** que define como os dados são acessados na memória;
4. As regras que regem a **comunicação** entre os processadores que trabalham em paralelo, para que eles possam trocar as informações necessárias;
5. Mecanismos de **sincronização**, para assegurar que estas informações alcancem o destino em tempo finito.

2.2.1 Modelo Seqüencial

Há mais de quarenta anos, a arquitetura dos computadores utiliza o modelo proposto por John von Neumann. Esta arquitetura é composta da unidade central de processamento (UCP) e da memória principal. A UCP, por sua vez, é composta de uma unidade de controle (UC) e uma unidade lógica e aritmética (ULA).

A memória principal armazena as instruções e os dados do programa que está sendo executado. A UC controla a execução do programa, buscando uma instrução depois da outra da memória para ser executada pela UCP. As

instruções e os dados trazidos da memória para serem executados são armazenadas nos registradores da UCP. Os registradores são pequenas memórias de acesso rápido. A ULA realiza as operações lógicas e aritméticas determinadas pelas instruções do programa.

O gargalo da arquitetura de Von Neumann é a transferência de dados e instruções entre a memória principal e a UCP, isto significa que, não importa o quanto a UCP seja rápida, a velocidade de execução dos programas é limitada pela velocidade de transferência das instruções e dados entre a memória e a UCP [Pac97].

Por esta razão, poucos computadores hoje utilizam exatamente a arquitetura clássica de Von Neumann. Por exemplo, a maioria das máquinas possuem uma hierarquia de memória, isto é, possuem uma memória intermediária, entre a memória principal e os registradores, chamada memória *cache*. O acesso a *cache* é mais rápido que o acesso à memória principal e mais lento que o acesso aos registradores. Com esse nível intermediário, a arquitetura ameniza os efeitos do gargalo de Von Neumann [Pac97].

O modelo RAM (*Random Access Machine*) é um modelo conceitual sólido que incorpora as principais características da computação seqüencial. A grande vantagem deste modelo é a sua capacidade de assimilar os impactos da tecnologia para computadores do tipo Von Neumann. Este modelo teórico permite o desenvolvimento de algoritmos seqüenciais bem como o estudo da complexidade desses algoritmos. Um algoritmo seqüencial é considerado eficiente quando pode ser resolvido em tempo polinomial com relação ao tamanho da entrada.

Na computação paralela, não temos um modelo de computação único como acontece na computação seqüencial. Isto faz com que cada algoritmo só faça sentido ser analisado, em termos de complexidade de tempo e uso de recursos, dentro de seu modelo. Para resolver o problema causado por esta heterogeneidade, há algumas tentativas de se criar um modelo que sirva de paradigma único para toda a área.

A seguir, vamos descrever alguns dos principais modelos de computação paralela, sendo que analisaremos com mais detalhes os modelos realísticos de computação paralela que utilizaremos neste trabalho.

2.2.2 Modelo de Memória Compartilhada

O modelo PRAM (*Parallel Random Access Machine*) corresponde a uma coleção de processadores, cada um com uma memória local, executando em

paralelo e comunicando-se através de uma memória global compartilhada. É uma extensão natural do modelo seqüencial RAM. Cada processador pode executar uma instrução distinta daquela executada por qualquer outro processador, em uma mesma unidade de tempo. Para um dado algoritmo, a quantidade de informação transferida entre a memória local de cada processador e a memória global é a quantidade de comunicação usada pelo algoritmo.

Há algumas variações no modelo PRAM baseadas na forma como a manipulação do acesso concorrente a uma mesma posição da memória global é efetuado:

- EREW (*Exclusive Read, Exclusive Write*): não permite qualquer acesso (leitura ou escrita) simultâneo a uma mesma posição de memória.
- CREW (*Concurrent Read, Exclusive Write*): permite leitura simultânea de uma mesma posição de memória por mais de um processador, mas não a escrita simultânea.
- CRCW (*Concurrent Read, Concurrent Write*): permite simultaneidade na leitura e escrita na mesma posição de memória por diferentes processadores. Neste caso há a necessidade de se estabelecer critérios para resolver o problema de escrita concorrente. Várias políticas podem ser adotadas para decidir qual dos processadores efetivamente efetuará a escrita:
  - escrita comum: é necessário que todos os processadores em conflito escrevam o mesmo valor.
  - escrita arbitrária: permite que um processador arbitrário tenha sucesso na escrita, não importando qual deles.
  - escrita prioritária: estabelece uma prioridade entre os processadores e aquele que tiver a maior prioridade terá sucesso na escrita.

O fato do modelo PRAM ignorar os custos associados à exploração do paralelismo incentiva o desenvolvimento de algoritmos paralelos, uma vez que pode-se extrair o máximo possível de paralelismo de um problema. Desta forma, o modelo fornece uma medida da complexidade de tempo paralelo ideal [MT95].

Um algoritmo paralelo no modelo PRAM é considerado eficiente se ele pertence à classe NC. **NC** é a classe de problemas que podem ser resolvidos em tempo paralelo poli-logarítmico usando um número polinomial de processadores.

Outro conceito importante sobre a eficiência de um algoritmo PRAM é dado pela definição de $P-$completude [GR88]:

Definição 2 ***P*** *é a classe de problemas que podem ser resolvidos em tempo paralelo polinomial usando um número polinomial de processadores. Um problema $L \in P$ é* ***P-completo*** *se todos os outros problemas em P podem ser transformados em L em tempo paralelo poli-logarítmico usando um número polinomial de processadores. Em outra palavras, L é o problema mais difícil de P no sentido de encontrar um algoritmo paralelo eficiente.*

Uma descrição do modelo PRAM e algoritmos para vários problemas podem ser vistos em Reif [Rei93].

2.2.3 Modelo de Memória Distribuída

O modelo de memória distribuída, também chamado de modelo de redes, consiste de um conjunto de processadores onde a memória está distribuída entre os processadores e nenhuma memória global está disponível.

Nesse modelo, a comunicação é feita através da troca de mensagens entre os processadores. O processo de entrega da mensagem do processador origem para o processador destino é chamado **roteamento**.

A topologia de interconexão dos processadores é incorporada pelo próprio modelo. O desempenho dessas topologias é dado em função de dois parâmetros: o **diâmetro**, que é a distância máxima entre quaisquer dois processadores; e o **grau máximo**, que corresponde ao número máximo de processadores ligados diretamente a um processador.

As principais topologias usadas no modelo de memória distribuída são:

- **Lista linear**: consiste de p processadores P_1, P_2, ..., P_p conectados linearmente, ou sejam P_i está conectado aos processadores P_{i-1} e P_{i+1}, se existirem. Possui diâmetro $p-1$ e seu grau máximo é 2.
- **Anel**: é uma lista linear onde o último processador está conectado ao primeiro. Possui diâmetro $\lfloor \frac{p}{2} \rfloor$ e grau máximo 2.
- **Grade**: é uma versão bidimensional da lista linear. Consiste de $p = n^2$ processadores conectados numa grade $n \times n$, tal que, cada processador $P_{i,j}$ está ligado aos processadores $P_{i\pm1,j}$ e $P_{i,j\pm1}$, se existirem. Tem diâmetro $2(\sqrt{p}-1)$ e grau máximo 4.

- **Árvore binária**: consiste de $p = 2^d - 1$ processadores onde cada processador P_i está conectado aos processadores P_{2i} e P_{2i+1} chamados filhos de P_i e ao processador $P_{\lfloor \frac{i}{2} \rfloor}$ chamado pai de P_i, se existirem. Possui diâmetro $2(\lfloor \log p \rfloor) - 1$ e seu grau máximo é 3.

- **Hipercubo**: consiste de $p = 2^d$ processadores interligados em um cubo d-dimensional definido a seguir. Considere os p processadores rotulados como P_0, P_1, ..., P_{p-1}. Seja a representação binária de i, $0 \leq i \leq p-1$ dada por $i = i_{d-1}i_{d-2}...i_2i_1i_0$. Dois processadores estão conectados se, e somente se, seus índices diferem em apenas um *bit*. O hipercubo tem diâmetro e grau máximo $d = \log p$.

O modelo de redes reflete as primeiras máquinas paralelas desenvolvidas. Estas máquinas tendiam a ter granularidade fina. O custo do acesso remoto é uma função da topologia e do roteamento. O modelo assume que a comunicação entre processadores vizinhos e a execução de uma instrução requerem custo unitário.

Nos modelos de memória distribuída, geralmente a computação é efetuada usando um número fixo de processadores interconectados numa determinada topologia. A análise da complexidade de algoritmos distribuídos leva em conta a comunicação e o tempo de processamento que são utilizados. No caso do tempo, é feito uma análise local e uma análise global. Com respeito a comunicação, são analisados o número e o tamanho das mensagens. Maiores detalhes a respeito da complexidade de algoritmos distribuídos pode ser encontrado em Barbosa [Bar96].

Mais detalhes sobre o modelo de memória distribuída podem ser vistos em Pacheco[Pac97] e Wilkinson e Allen[WA99].

2.2.4 Modelos Realísticos

O modelo PRAM, que poderia ser um padrão único para a computação paralela, não consegue capturar com exatidão a noção de paralelismo. As características não incorporadas ao modelo, como o custo da comunicação, tem grande impacto no desempenho dos algoritmos.

Já no modelo de redes, os algoritmos tendem a ser específicos para uma determinada topologia e o fato do modelo assumir que a comunicação entre processadores adjacentes pode ser feita em tempo constante não reflete a realidade dos computadores paralelos.

Assim, a computação paralela busca por um modelo adequado onde o desempenho dos algoritmos desenvolvidos sirva de referência para o desempenho das respectivas implementações. Os modelos apresentados a seguir, denominados **realísticos**, buscam estabelecer padrões amplamente aceitos que reflitam as dificuldades inerentes ao paralelismo.

Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*), introduzido por Valiant [Val90], é um dos principais modelos realísticos. Foi um dos primeiros modelos a considerar os custos de comunicação como característica de um modelo de computação paralela.

Uma máquina BSP consiste num conjunto de p processadores com memória local. Os processadores comunicam-se através de algum meio de interconexão, gerenciados por um roteador que pode transmitir mensagens ponto-a-ponto entre pares de processadores. Também oferece capacidade para sincronizar alguns, ou todos, os processadores em intervalos regulares de L unidades de tempo.

Um algoritmo BSP consiste de uma seqüência de super-passos (Figura 2.1). Em cada super-passo, cada processador executa uma combinação de computações locais, transmissão de mensagens e recebimento de mensagens de outros processadores. As tarefas de computação e comunicação podem ser separadas em super-passos de computação e super-passos de comunicação. Após cada período de L unidades de tempo, uma barreira de sincronização é realizada para determinar se o super-passo foi completado por todos os processadores. Em caso afirmativo, a máquina executa o próximo super-passo; caso contrário, há outro período de L unidades de tempo para terminar o super-passo. Isso assegura que os dados recebidos pelos processadores estarão disponíveis para o próximo super-passo.

O modelo possui os seguintes parâmetros:

- n: tamanho do problema;
- p: número de processadores com memória local;
- L: tempo máximo de um super-passo (periodicidade);
- g: descreve a taxa de eficiência de computação e comunicação, que corresponde a razão entre:

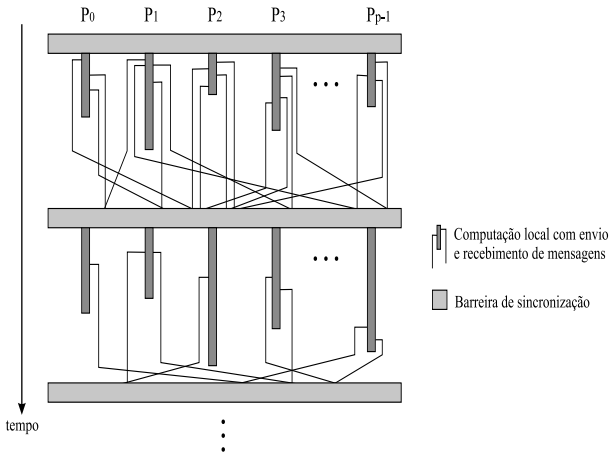


Figura 2.1: Algoritmo BSP [Göt98]

- o número total de operações de computação local de todos os processadores (em unidades básicas de computação) em uma unidade de tempo e,
- o número total de mensagens enviadas/recebidas (em palavras) em uma unidade de tempo.

Um algoritmo BSP com um total de λ super-passos apresenta os seguintes custos:

- Custo de Computação: $T_{comp} = \sum_{i=1}^{\lambda} w_{comp}^{i}$, onde o i-ésimo super-passo de computação possui custo $w_{comp}^{i} = max\{L, t_1, ..., t_p\}$, t_j é o número de operações básicas de computação executado pelo processador j no i-ésimo super-passo.
- Custo de Comunicação: $T_{comm} = \sum_{i=1}^{\lambda} w_{comm}^{i}$, onde o i-ésimo super-passo de comunicação possui custo $w_{comm}^{i} = max_{j=1}^{p}\{w_{comm,j}^{i}\}$, $w_{comm,j}^{i}$ é o custo de comunicação do processador j no i-ésimo super-passo. Assumindo que o processador P_j recebe mensagens de tamanho $r_1, ..., r_{j'}$ e envia mensagens de tamanho $s_1, ..., s_{j''}$ durante o i-ésimo super-passo, $w_{comm,j}^{i} = max\{L, g(\sum_{u=1}^{j'} r_i + \sum_{u=1}^{j''} s_i)\}$.

Modelo CGM

O modelo CGM (*Coarse Grained Multicomputer*) foi proposto por Dehne *et al* [DFRC93] com o propósito de ser um modelo de paralelismo suficiente-

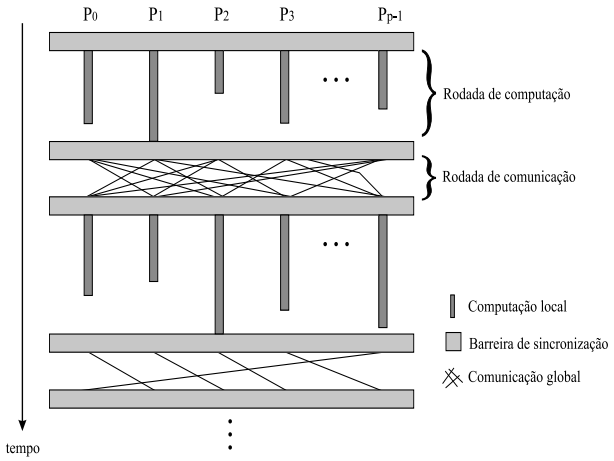


Figura 2.2: Algoritmo CGM [Göt98]

mente próximo às máquinas paralelas com memória distribuída existentes. É muito parecido com o modelo BSP, entretanto utiliza apenas dois parâmetros: n - o tamanho da entrada - e p - o número de processadores. O termo *coarse grained*[1] vem do fato do tamanho do problema ser consideravelmente maior que o número de processadores.

Uma máquina CGM(n, p) consiste de um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{n}{p})$. Cada processador é conectado a um roteador que pode enviar mensagens ponto-a-ponto para qualquer outro processador.

Um algoritmo CGM consiste de uma seqüência alternada de rodadas de computação e rodadas de comunicação separadas por uma barreira de sincronização (Figura 2.2). Na rodada de computação, cada processador pode utilizar o melhor algoritmo seqüencial para processar seus dados localmente. Em cada rodada de comunicação, cada processador troca no máximo um total de $O(\frac{n}{p})$ dados com outros processadores. No modelo CGM, o custo de comunicação é modelado pelo número de rodadas de comunicação.

Uma rodada de computação no modelo CGM é equivalente a um superpasso de computação no modelo BSP, e o custo total de computação T_{comp} é definido analogamente. Uma rodada de comunicação consiste de uma única h-relação com $h \leq \frac{n}{p}$, isto é, cada processador envia ou recebe no máximo h palavras de dados. O custo w^i_{comm} para cada rodada de comunicação possui um valor único denominado $H_{n,p}$. Então, o custo total de comunicação T_{comm}

[1] Granularidade grossa

de um algoritmo CGM com λ rodadas de comunicação é $T_{comm} = \lambda H_{n,p}$.

No modelo CGM, o esforço para reduzir a comunicação está centralizada na redução do número de rodadas de comunicação.

O modelo GGM proporciona o desenvolvimento simples e prático de algoritmos paralelos para sistemas paralelos de granularidade grossa ($\frac{n}{p} \gg 1$). Espera-se que esses algoritmos apresentem uma complexidade teórica bem próxima do tempo de execução observado em implementações reais. Os algoritmos geralmente requerem $\frac{n}{p} \geq p$ ou $\frac{n}{p} \geq p^{\epsilon}$. O modelo CGM é particularmente adequado para as máquinas paralelas atuais nas quais a velocidade global de computação é consideravelmente maior que a velocidade global de comunicação.

A principal vantagem do modelo CGM é que ele permite modelar o custo de comunicação de um algoritmo paralelo através de um único parâmetro, o número de rodadas de comunicação, fornecendo um prognóstico do desempenho bastante realista para os multi-processadores comerciais disponíveis [Deh99].

Os conceitos de algoritmo ótimo e algoritmo eficiente ainda não estão bem estabelecidos nos modelos realísticos. Considerando uma máquina CGM(n, p), onde n é o tamanho da entrada e p é o número de processadores, vamos considerar "aceitável", um algoritmo que efetue no máximo $O(p)$ rodadas de comunicação e cujo *speedup* seja próximo do linear (com relação ao número de processadores). Um dos objetivos do modelo CGM é o de diminuir o número de rodadas de comunicação. Nosso objetivo é o de tentar utilizar o menor número de rodadas possíveis. Nesse sentido, um algoritmo que efetue um número de rodadas $O(\log p)$ pode ser considerado mais "eficiente"que um que utilize p rodadas, desde que a computação local efetuada pelos dois algoritmos tenham a mesma ordem. Claramente, algoritmos que utilizem $O(1)$ rodadas de comunicação e tenham *speedup* linear, podem ser considerados como os melhores algoritmos para resolver um problema no modelo CGM. Com algumas adaptações, estas observações podem ser utilizadas também no modelo BSP.

Uma descrição dos modelos realísticos de computação paralela pode ser visto em Cáceres, Mongelli e Song [CMS01].

2.3 Programação com Troca de Mensagens

No paradigma seqüencial de programação o programador possui uma visão simplificada da máquina como sendo um único processador que pode acessar uma certa quantidade de memória. O programa desenvolvido neste paradigma é portável para qualquer arquitetura seqüencial.

O paradigma de troca de mensagens é uma busca da portabilidade para a programação paralela. Neste paradigma várias instâncias do paradigma seqüencial trabalham juntas. Isto significa que o programador pode imaginar vários processadores, cada um com a sua própria memória, e escrever um programa para executar em cada processador. Mas a programação paralela, por definição, requer a cooperação entre os processadores para resolver o problema, o que torna necessário a implementação de algum meio de comunicação.

A principal característica do paradigma da troca de mensagens é que os processadores comunicam-se através do envio de mensagens uns para os outros. Então, no modelo de troca de mensagens não há o conceito de memória compartilhada ou de processadores que possam acessar a memória de outro processador diretamente.

O paradigma de troca de mensagens vem se tornando cada vez mais popular. Uma das razões é o grande número de plataformas que oferecem suporte para o modelo. Programas escritos neste paradigma podem executar em multiprocessadores de memória distribuída ou compartilhada, redes de computadores e sistemas com um único processador. Assim como no paradigma seqüencial, o programador sabe que o seu programa, a princípio, é portável para qualquer arquitetura que ofereça suporte para o modelo de troca de mensagens. Este paradigma é popular não por ser particularmente simples, mas por ser genérico.

Segundo Wilkinson e Allen [WA99], existem três maneiras de programar-se em um ambiente de troca de mensagens:

1. desenvolvendo uma linguagem de programação paralela;
2. estendendo a sintaxe e as palavras reservadas de uma linguagem seqüencial de alto nível existente;
3. usando uma linguagem seqüencial de alto nível existente e providenciando uma biblioteca contendo as rotinas de troca de mensagens.

Neste trabalho, vamos utilizar a terceira opção. Nessa abordagem, é

necessário dizer explicitamente quais processos devem ser executados, quando deve ocorrer a troca de mensagens entre processos concorrentes e quais informações estão sendo passadas através dessas mensagens. Precisamos especificar dois mecanismos nesse ambiente de troca de mensagens [WA99]:

1. um mecanismo para criação de processos que são executados em diferentes processadores, e
2. um mecanismo para o envio e o recebimento de mensagens.

2.3.1 Criação de Processos

Um ambiente de troca de mensagem deve prover mecanismos que permitam criar processos e iniciar a execução desses processos. Os processos podem ser criados de modo **estático** ou **dinâmico**.

No modo estático, todos os processos são especificados antes da execução e o sistema executa um número pré-determinado de processos. Na maioria das aplicações existe um processo especial que controla a execução dos demais processos, denominado **processo mestre**. Os demais processos são denominados **processos escravos**. No modelo SPMD (*Single Program, Multiple Data*) o código dos diferentes processos são colocados dentro de um único programa onde é possível selecionar diferentes partes para cada processo. Cada processador carrega uma cópia deste programa na sua memória local para ser executado. O modelo SPMD implementa um paralelismo de dados.

Já no modo dinâmico, os processos podem ser criados e suas execuções iniciadas durante a execução de outros processos. Processos também podem ser destruídos. O modelo mais usado para criação de processos dinâmicos é o MPMD (*Multiple Program Multiple Data*), no qual programas diferentes são escritos para processadores diferentes. Neste modelo também podemos usar o paradigma mestre-escravo, um único processador executa o processo mestre e os processos escravos são criados e iniciados, nos demais processadores, pelo processo mestre. O modelo MPMD implementa um paralelismo funcional.

2.3.2 Rotinas para Troca de Mensagens

Envio e Recebimento

Existem duas rotinas básicas para o **envio** e o **recebimento** de mensagens entre os processadores. A **rotina de envio** é usada pelo processo que deseja

enviar uma mensagem e a **rotina de recebimento** é usada pelo processo que deseja receber a mensagem que foi enviada.

Os principais parâmetros utilizados por essas rotinas são o processador destino e a localização do dado a ser enviado na rotina de envio; e o processador origem e a localização onde o dado recebido será armazenado na rotina de recebimento.

As rotinas de envio e recebimento podem ser **bloqueantes** ou **não-bloqueantes**. O termo **bloqueante** é usado para descrever rotinas que bloqueiam a execução do processo, isto é, não permitem que o processo execute a próxima instrução até que a mensagem enviada tenha sido recebida.

Já o termo **não-bloqueante** é usado para descrever rotinas que não bloqueiam a execução do processo, ou seja, que permitem que o processo continue sua execução mesmo que a mensagem enviada ainda não tenha sido recebida. Mas, como uma rotina de troca de mensagens pode retornar antes que a transferência tenha sido completada? Geralmente, é necessário um ***buffer* de mensagens** entre os processos origem e destino para armazenar as mensagens que estão sendo enviadas. Assim, se o recebimento é executado antes do envio, ele encontra o *buffer* de mensagens vazio e aguarda a chegada da mensagem. Mas, se o envio é executando antes, uma vez que a mensagem esteja armazenada no *buffer*, o processo pode continuar a sua execução.

Broadcast, Gather e Scatter

Além das rotinas básicas de envio e recebimento existem muitas outras rotinas relacionadas a troca de mensagem. Uma situação bastante freqüente é a necessidade de um processo enviar a mesma mensagem para mais de um processo destino. A operação ***multicast*** é usada para enviar uma mensagem para um grupo definido de processos. Uma generalização é dada pela operação ***broadcast***, usada para enviar uma mensagem para todos os processos envolvidos com a solução do problema.

O ***broadcast*** é ilustrado na Figura 2.3. O Processo 0, identificado como processo raiz, armazena em um *buffer* o dado que deve ser enviado. Todos os processos executam a rotina *broadcast* e recebem o dado numa determinada variável. O *broadcast* não será executado até que todos tenham alcançado a sua chamada, e por isso sincroniza a execução dos processos.

A operação ***scatter*** é usada para o envio de cada elemento de um vetor, armazenado no processo raiz, para diferentes processos. O conteúdo da i-ésima posição do vetor é enviado para o i-ésimo processo. O *scatter* é

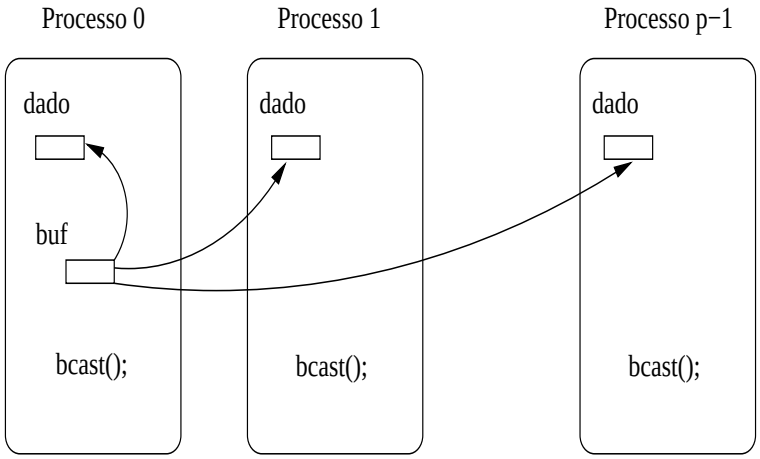


Figura 2.3: A operação *broadcast* [WA99]

ilustrado na Figura 2.4. Todos os processos executam a rotina *scatter*, e o processo raiz também recebe um elemento do vetor.

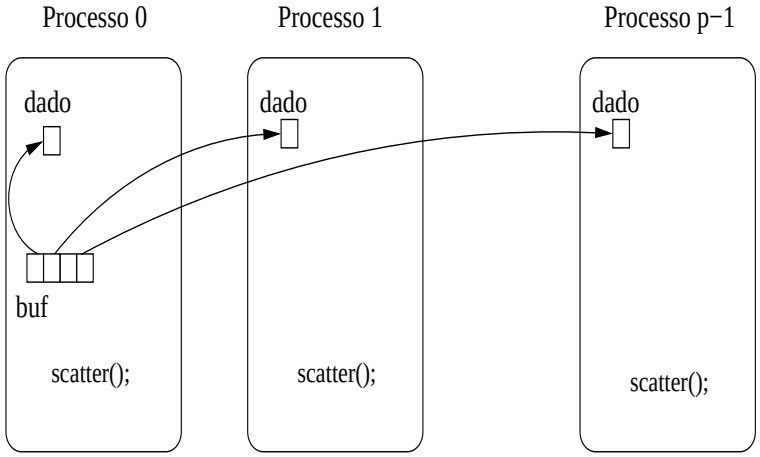


Figura 2.4: A operação *scatter*[WA99]

A operação ***gather*** é o oposto da operação *scatter*. O dado do i-ésimo processo é recebido pelo processo raiz e armazenado na i-ésima posição do vetor. O *gather* é ilustrado na Figura 2.5.

As vezes a operação *gather* pode ser combinada com uma operação aritmética específica ou uma operação lógica. Por exemplo, os valores podem ser coletados e depois adicionados pela raiz, como mostra a Figura 2.6. O

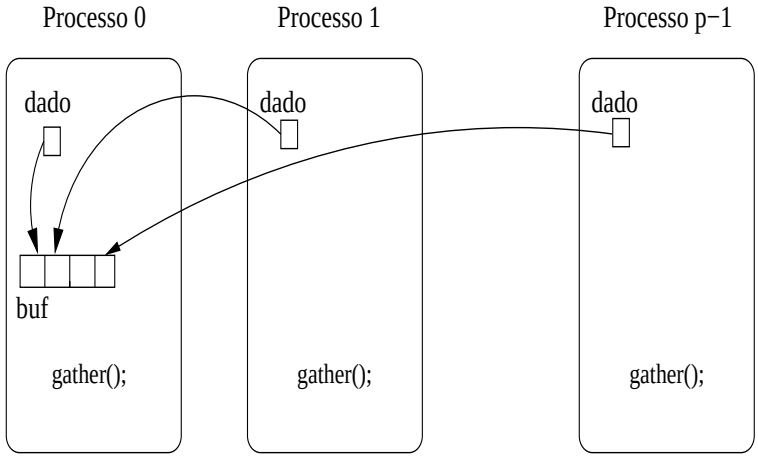


Figura 2.5: A operação *gather*[WA99]

termo ***reduce*** é utilizado para definir esta operação.

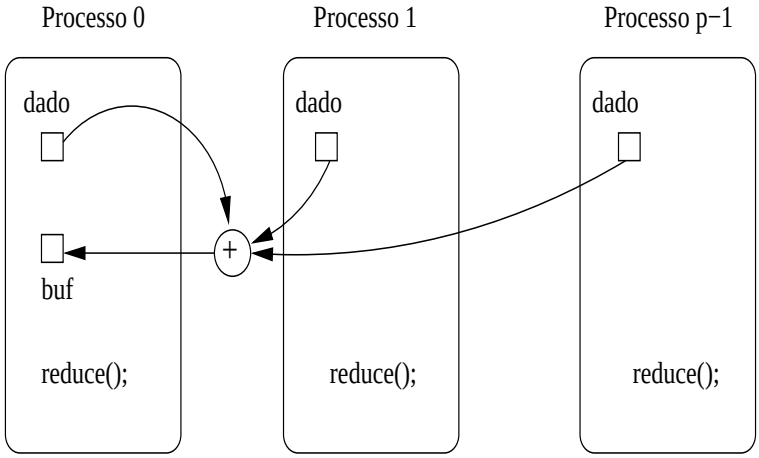


Figura 2.6: A operação *reduce*[WA99]

2.4 Ambientes de Troca de Mensagens

Os ambientes de troca de mensagens foram remodelados ou desenvolvidos com três grandes objetivos:

1. Utilizar o potencial dos sistemas distribuídos para o desenvolvimento

de aplicações paralelas;

2. Permitir a união de plataformas heterogêneas;

3. Permitir a portabilidade das aplicações paralelas desenvolvidas.

Baseados nesses objetivos e também devido a diversidade encontrada nos ambientes já existentes, vários grupos foram formados com a finalidade de desenvolver as plataformas de portabilidade de troca de mensagens. Essas plataformas tem por objetivo tornar os ambientes portáveis, ou seja, os programas podem ser executados em qualquer equipamento para o qual o ambiente foi desenvolvido. Podemos citar como exemplos dessas plataformas, o PVM (*Parallel Virtual Machine*) e o MPI (*Message Passing Interface*). As próximas seções apresentam uma introdução ao PVM e ao MPI. O Apêndice A descreve as principais rotinas do MPI com maiores detalhes.

2.4.1 PVM - Parallel Virtual Machine

O PVM é um conjunto integrado de bibliotecas de funções e de ferramentas de *software*, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e de propósito geral. Permite que uma coleção de sistemas de computadores heterogêneos possa ser visto como uma única máquina paralela virtual. O projeto do PVM teve início em 1989 no *Oak Ridge National Laboratory* - ORNL e seu protótipo (versão 1.0) foi implementado por *Vaidy Sunderam* e *Al Geist* (ORNL). Com a versão 2.0 deu-se início a disponibilidade pública do código. A versão mais recente e disponível é a 3.4. O PVM permite que sejam escritas aplicações nas linguagens Fortran, C e C++.

Como vantagem da utilização do PVM podemos citar: o ambiente é amplamente utilizado tanto pela comunidade científica quanto comercial, ou seja, é de grande aceitação; pode-se fazer utilização da computação paralela em equipamentos não paralelos; o *software* pode ser obtido através de uma distribuição não comercial, flexível e também de fácil programação pois se trata de uma extensão de linguagens conhecidas.

O modelo computacional do PVM

No PVM, um conjunto de computadores seriais, paralelos e vetoriais emula um grande computador de memória distribuída denominada **máquina virtual**. Cada componente desta máquina virtual será denominado ***host***.

Vários usuários podem definir diferentes máquinas virtuais na mesma rede, podendo algumas máquinas fazer parte de mais de uma máquina virtual, executando aplicações PVM simultaneamente. O PVM fornece funções para iniciar a execução de tarefas (*tasks*) na máquina virtual e também funções para que estas tarefas comuniquem-se e sincronizem-se entre si.

Uma aplicação é composta por diversas tarefas, segundo o modelo computacional do PVM, assumindo-se que qualquer tarefa pode enviar uma mensagem para qualquer outra tarefa. Cada tarefa é responsável por uma parte da carga de trabalho computacional da aplicação.

Em termos de implementação, o modelo computacional do PVM é composto por duas partes: um *daemon* (usualmente denotado por *pvmd*) e uma biblioteca de rotinas com a interface PVM (usualmente denotada por *libpvm*).

O usuário que deseje utilizar o PVM deve primeiramente configurar a máquina virtual especificando a lista de *hosts* e iniciar o *daemon* em todas as máquinas que fazem parte da máquina virtual. O *pvmd* não faz nenhum processamento, somente roteamento e controle de mensagens, ou seja, funciona como um ponto entre os *hosts* autenticando as mensagens, fazendo o controle de processos e detectando possíveis falhas.

Na *libpvm* encontram-se as primitivas necessárias à cooperação entre as tarefas de uma aplicação, funcionando como um elo de ligação entre uma tarefa e a máquina virtual, ou seja, são rotinas que podem ser chamadas pelo usuário para efetuar troca de mensagens, solicitar a geração de processos, coordenar tarefas e solicitar modificações na máquina virtual.

Criação e Execução de Processos

Os processos podem ser iniciados dinamicamente. Os programas PVM geralmente utilizam o modelo mestre-escravo, onde o processo mestre é o primeiro a ser executado e todos os outros processos são criados por ele. Para iniciar a execução de um ou mais processos idênticos, usamos a rotina `pvm_spawn()`.

Comunicação Ponto-a-ponto

A comunicação ponto-a-ponto é realizada no PVM através das rotinas `pvm_send()` e `pvm_recv()`. Todas as rotinas de envio do PVM são não-bloqueantes e as rotinas de recebimento podem ser bloqueantes ou não-bloqueantes. As rotinas de envio e recebimento de mensagens utilizam um *buffer* de mensagens. Se os dados estiverem armazenados no *buffer*, a rotina

de envio é usada para iniciar o envio do conteúdo do *buffer* de envio através da rede até o *buffer* de recebimento, preparado pela rotina de recebimento.

Os dados da mensagem a ser enviada devem ser empacotados em um *buffer* de envio antes de serem enviados. Existem rotinas específicas para empacotar e desempacotar cada tipo de dados, por exemplo, `pvm_pkint()` e `pvm_upkint()` para inteiros. O *buffer* de envio padrão deve ser inicializado pelo processo origem através da rotina `pvm_initsend()`. A rotina `pvm_recv()` cria um novo *buffer* de recebimento para receber a mensagem.

Comunicação Coletiva

O PVM também oferece várias rotinas para comunicação coletiva. Essas rotinas são utilizadas para a comunicação entre os membros de um grupo de processos. Em geral, os grupos envolvem mais que dois processos. Essas rotinas implementam as operações coletivas descritas na Seção 2.3.2.

As rotinas `pvm_bcast()`, `pvm_gather()`, `pvm_scatter()` e `pvm_reduce()` são as implementações das operações de *broadcast*, *gather*, *scatter* e *reduce*, respectivamente.

Considerações Finais sobre o PVM

Além das rotinas descritas nesta seção, o PVM oferece outras rotinas necessárias para o desenvolvimento de programas que utilizam a troca de mensagens como forma de comunicação entre as tarefas. A descrição dessas rotinas podem ser obtidas em [GBD+94]. Informações sobre o PVM podem ser obtidas em http://www.csm.ornl.gov/pvm/pvm_home.html.

2.4.2 MPI - Message Passing Interface

O MPI é um padrão que tenta definir a sintaxe e a semântica de uma biblioteca de rotinas de troca de mensagens. A principal vantagem de estabelecer um padrão é a portabilidade. O MPI foi desenvolvido por um comitê (*MPI Committee*) composto por cerca de 80 pesquisadores representando mais de 40 organizações que trabalham com processamento paralelo em todo mundo, especialmente na Europa e Estados Unidos.

A especificação do MPI

O MPI foi desenvolvido de modo a alcançar os seguintes objetivos:

- Portabilidade do código fonte, permitindo que uma implementação seja usada em um ambiente heterogêneo sem alterações;
- Possibilidade de ser utilizado tanto em sistemas de memória distribuída quanto em memória compartilhada e NOWs (*network of workstations*), ou em uma combinação deles;
- Definir uma interface semelhante à dos padrões PVM, Express[2], P4[3], Linda[4], etc.;
- Prover uma interface de comunicação confiável, de tal forma, que o usuário não precise se preocupar com falhas na comunicação;
- Definir uma interface que possa ser implementada em muitas plataformas sem mudanças muito significativas na parte de comunicação e do *software*.

Inicialmente um conjunto básico de rotinas foi definido. Este conjunto inclui rotinas para comunicação ponto-a-ponto e comunicação coletiva, suporte para grupo de processos (relaciona processos por grupos e os processos são classificados dentro de cada grupo), suporte para contexto de comunicação e suporte para topologia de processos (definir uma estrutura topológica que descreve o relacionamento entre os processos).

As principais implementações do MPI são:

- LAM: executa em redes com estações de trabalho. A implementação foi desenvolvida pela universidade de Ohio (*Ohio Supercomputer Center*) e pode ser obtida em http://www/mpi.nd.edu/lam/download.
- MPICH: executa em diferentes plataformas. A implementação foi desenvolvida no Argonne National Lab e na Universidade de Mississippi e pode ser obtida em ftp://info.mcs.anl.gov/pub/mpi.

[2]Conjunto de ferramentas que individualmente modelam vários aspectos da computação concorrente. Desenvolvido comercialmente pela ParaSoft Corporation.

[3]Biblioteca de sub-rotinas desenvolvida pelo Argonne National Laboratories para programação de diversas máquinas paralelas.

[4]Modelo de programação concorrente que foi desenvolvido por um projeto da Yale University.

- CHIMP: implementação desenvolvida no Edinburgh Parallel Computing Center pode ser obtido em ftp://ftp.epcc.ed.ac.uk/pub/chimp.
- W32MPI: executa em PC usando Windows e está disponível em ftp://dsg.dei.uc.pt/wmpi/introd.html.

Assim como o PVM, o MPI provê uma biblioteca de rotinas para troca de mensagens e outras operações relacionadas. Nas próximas seções, discutiremos algumas dessas rotinas.

Criação e Execução de Processos

A criação e a iniciação dos processos não estão definidos no padrão MPI e dependem da implementação. O MPI permite a criação estática de processos, o que possibilita a utilização do modelo SPMD. Geralmente, um programa executável é iniciado por linha de comando. Por exemplo, o mesmo programa pode ser iniciado em diferentes processadores simultaneamente através do comando `mpirun prog1 -np 4`.

Este comando inicia a execução de quatro cópias do programa `prog1`. O comando não especifica onde as cópias serão executadas. O mapeamento dos processos nos processadores também não está definido no padrão MPI e depende da implementação. O mapeamento pode ser feito através de linhas de comandos ou através de um arquivo contendo o nome dos programas e os processadores onde devem ser executados.

Antes que qualquer função MPI seja chamada, a função `MPI_Init()` deve inicializar o MPI, e após todas as funções MPI terem sido chamadas, a função `MPI_Finalize()` deve finalizar o MPI.

Inicialmente, todos os processos pertencem a um único comunicador denominado `MPI_COMM_WORLD`, e cada processo recebe um único identificador que corresponde a um número entre 0 e $n-1$, onde n é o número de processos. Um comunicador define o escopo de uma operação de comunicação, ou seja, o conjunto de processos que podem comunicar-se. O MPI permite que diferentes escopos sejam criados, isso significa que o programa pode definir diferentes escopos formados por diferentes subconjuntos dos processos. Cada escopo criado é denominado **grupo** na terminologia MPI. Cada processo possui um identificador único dentro de um grupo e um processo pode ser membro de mais de um grupo.

Comunicação Ponto-a-ponto

A comunicação ponto-a-ponto é feita através das funções de envio e recebimento. No MPI, essas funções podem ser bloqueantes ou não-bloqueantes.

No MPI, as rotinas bloqueantes retornam quando estão localmente completas. O envio bloqueante envia a mensagem e retorna. Isso não significa que a mensagem tenha sido recebida, mas que o processo pode continuar sua execução sem afetar a mensagem. O recebimento bloqueante retorna quando a mensagem é recebida. As rotinas de envio e recebimento bloqueantes no MPI são implementadas pelas funções `MPI_Send()` e `MPI_Recv()`, respectivamente.

Uma rotina não-bloqueante retorna imediatamente independente da rotina estar ou não localmente completa. Por exemplo, o recebimento não-bloqueante irá retornar mesmo que a mensagem ainda não tenha sido recebida. As funções `MPI_Isend()` e `MPI_Irecv()` implementam, respectivamente, as rotinas de envio e recebimento não-bloqueantes.

Comunicação Coletiva

As rotinas de comunicação coletiva envolvem um conjunto de processos, diferentemente das rotinas de comunicação ponto-a-ponto que envolvem apenas um processo origem e um processo destino. O comunicador define o conjunto de processos que irá participar da operação coletiva.

Assim como o PVM, o MPI oferece um conjunto de rotinas que implementam as operações coletivas descritas na Seção 2.3.2. As rotinas `MPI_Bcast()`, `MPI_Gather()`, `MPI_Scatter()` e `MPI_Reduce()` implementam as operações de *broadcast*, *gather*, *scatter* e *reduce*, respectivamente. Além dessas rotinas básicas o MPI oferece outras rotinas de comunicação coletiva ainda mais poderosas que são descritas no Apêndice A.

Considerações Finais sobre o MPI

Um dos aspectos mais importantes do MPI é o de prover um padrão para o desenvolvimento de bibliotecas para programação paralela. Por se tratar de um padrão, existe uma preocupação constante em manter o MPI atualizado para que ele não se torne obsoleto com o passar do tempo e o advento de novas tecnologias.

Pensando nisso, o fórum do MPI definiu uma coleção de extensões, deno-

minado MPI-2, para o MPI. Entre outras coisas, o MPI-2 define um padrão para o gerenciamento de processos dinâmicos e programação em tempo real, uma extensão para as rotinas de comunicação coletiva e uma biblioteca de classes C++. Esse novo padrão visa corrigir deficiências do padrão atual e adiciona novas e poderosas funcionalidades ao MPI (por exemplo, operações que permitem que os processos acessem diretamente posições de memória remota).

Usando o MPI como modelo, outros grupos estão trabalhando no desenvolvimento de uma das mais importantes omissões do padrão original: um padrão para entrada e saída em sistemas paralelos [Pac97].

Uma descrição detalhada do MPI pode ser encontrada em Snir *et al* [SOHL$^+$96] e Gropp *et al* [GLS99].

2.5 Grafos

Esta seção apresenta os conceitos de teoria dos grafos utilizados no presente trabalho. Uma descrição mais detalhada dos conceitos aqui apresentados pode ser encontrada em Bondy e Murty [BM76] e Szwarcfiter [Szw88].

Definição 3 *Um **grafo** G = (V, E) corresponde a um conjunto finito não-vazio V de elementos chamados vértices e um conjunto E de pares de vértices de V, chamados arestas.*

Se as arestas forem pares ordenados $\langle u, v\rangle$ o grafo é denominado **orientado**. Se as arestas forem pares não-ordenados (u, v) de vértices o grafo é dito **não-orientado**. Se (u, v) é uma aresta não-orientada, u e v são **adjacentes**. Se $\langle u, v\rangle$ é uma aresta orientada, u é o **predecessor** de v e v é o **sucessor** de u. Neste trabalho consideramos que $\mid V \mid = n$ e $\mid E \mid = m$.

Uma extensão possível consiste em substituir, na definição de grafo, o conjunto de arestas E por um multiconjunto. O efeito desta alteração é, naturalmente, permitir a existência de mais de uma aresta entre o mesmo par de vértices, as quais são denominadas **arestas paralelas**. A estrutura assim definida é um **multigrafo**.

Definição 4 *Em um grafo não-orientado, define-se **grau** de um vértice $v \in V$, denotado por D_v, como sendo o número de arestas $e \in E$ adjacentes à v. Em um grafo orientado, o **grau de entrada** de v é o número de arestas convergentes a v. O **grau de saída** de v é o número de arestas divergentes de v.*

Definição 5 *Um grafo G = (V, E) é* ***bipartido*** *quando o seu conjunto de vértices V pode ser particionado em dois subconjuntos V_1 e V_2, tais que toda aresta de G une um vértice de V_1 a outro de V_2.*

Definição 6 *Uma seqüência de vértices v_1,..., v_k tal que (v_j, v_{j+1}) $\in$ E, 1 $\leq j \leq k-1$, é denominado* ***caminho*** *de v_1 a v_k. Um caminho de k vértices é formado por k−1 arestas (v_1, v_2), (v_2, v_3),..., (v_{k-1}, v_k), onde k−1 é o* ***comprimento*** *do caminho.*

Definição 7 *Um* ***sub-grafo*** *$G_2 = (V_2, E_2)$ de um grafo $G_1 = (V_1, E_1)$, é um grafo tal que $V_2 \subseteq V_1$ e $E_2 \subseteq E_1$.*

Definição 8 *Seja G = (V, E) um grafo não-orientado. Um grafo é denominado* ***conexo*** *se, para todo par de vértices u e v em V, existe um caminho entre u e v.*

Seja S um conjunto e $S' \subseteq S$. Diz-se que S' é **maximal** em relação a uma certa propriedade P, quando S' satisfaz a propriedade P e não existe subconjunto $S'' \supset S'$, que também satisfaz P. Ou seja, S' não está propriamente contido em nenhum subconjunto de S que satisfaça P.

Definição 9 *Denominam-se* ***componentes conexos*** *de um grafo G aos sub-grafos maximais de G que sejam conexos. A propriedade P, neste caso, é equivalente a ser conexo.*

Definição 10 *Um* ***ciclo****, ou* ***circuito****, é um caminho v_1,...,v_k, v_{k+1}, sendo $v_1 = v_{k+1}$ e $k \geq 3$. Um grafo que não possui ciclos é denominado* ***acíclico****.*

Definição 11 *Denomina-se* ***árvore*** *um grafo T = (V, E) que seja acíclico e conexo. Se um vértice v da árvore T possuir grau igual a 1 então v é uma* ***folha****. Caso contrário, v é um* ***vértice interno****.*

Definição 12 *Denomina-se* ***sub-grafo gerador*** *de um grafo $G_1 = (V_1, E_1)$, a um grafo $G_2 = (V_2, E_2)$ de G_1 tal que $V_1 = V_2$ e $E_1 \subset E_2$. Quando o sub-grafo gerador é uma árvore, ele recebe o nome de* ***árvore geradora****. Todo grafo conexo G possui uma árvore geradora.*

Definição 13 *Um* ***circuito de Euler*** *de um grafo G é um ciclo que contém cada aresta de G exatamente uma única vez.*

Teorema 1 *Um grafo G possui um circuito de Euler se, e somente se, todo vértice de G possuir grau par.*

Demonstração [Szw88]: Seja C um circuito de Euler de G. Cada ocorrência de um dado vértice v em C contribui com 2 unidades para o cálculo do grau de v. Como cada aresta de G aparece exatamente uma vez em C, conclui-se que v possui grau par. Para a prova de suficiência, particiona-se as arestas de G em um conjunto C de ciclos simples, do seguinte modo. Como cada vértice de G possui grau ≥ 2, G contém necessariamente algum ciclo simples C_1. Se C_1 contém todas as arestas de G, o particionamento C está terminado. Caso contrário, remove-se de G todas as arestas do ciclo C_1 e os vértices isolados (se houver algum após a remoção das arestas). No grafo resultante, cada vértice possui ainda grau par. Determina-se assim um novo ciclo simples e assim por diante. Ao final, as arestas de G encontram-se particionadas em ciclos simples. Para determinar o circuito de Euler, considera-se um ciclo, por exemplo C_1, do particionamento C. Se não há mais ciclos de C a serem considerados, a prova está concluída. Caso contrário, como G é conexo, existe um ciclo $C_2 \in C$ tal que C_1 e C_2 possuem um vértice em comum v. Então o ciclo formado pela união das arestas de C_1 e C_2 contém cada uma dessas arestas exatamente uma vez. Repete-se o processo, considerando um novo ciclo $C_3 \in C$, ainda não considerado e assim por diante. □

Definição 14 *Um **grafo euleriano** é um grafo que possui um circuito de Euler.*

Definição 15 *Um **multigrafo euleriano** é um multigrafo que possui um circuito de Euler.*

2.6 Conclusão

Neste capítulo, descrevemos os principais conceitos da computação paralela. Também descrevemos alguns modelos computacionais, entre eles o modelo seqüencial RAM, o modelo paralelo de memória compartilhada PRAM, o modelo de memória distribuída e os modelos realísticos BSP e CGM.

Os principais aspectos da programação por troca de mensagens foram discutidos. Também mostramos dois ambientes para programação por troca de mensagens, o PVM e o MPI. Esses ambientes são bastante utilizados e apresentam características que motivam sua utilização na implementação de algoritmos para modelos realísticos de computação paralela.

Este capítulo ainda descreve alguns conceitos em teoria dos grafos. Esses conceitos são importantes para o entendimento dos algoritmos descritos nos próximos capítulos.

O algoritmo paralelo para computação de circuitos de Euler em grafos proposto neste trabalho utiliza o modelo de computação paralela BSP/CGM. Escolhemos este modelo porque o BSP/CGM permite o desenvolvimento de algoritmos adequados às características das máquinas paralelas reais. Com isso, espera-se que os resultados obtidos com a implementação sejam próximos da complexidade teórica do algoritmo.

Por se tratar de um modelo de memória distribuída, o paradigma de programação por troca de mensagens será utilizado no modelo. O algoritmo proposto será implementado na linguagem C utilizando a biblioteca MPI. A principal vantagem do MPI é ser um padrão para troca de mensagens. Existem várias implementações do MPI disponíveis, o que possibilita a execução do programa em diferentes tipos de máquinas paralelas.

Capítulo 3

Algoritmos para Computar Circuitos de Euler

3.1 Circuitos de Euler

3.1.1 Definição

Os grafos eulerianos podem ser utilizados na solução de diversos problemas práticos como, por exemplo, em telecomunicações. Como vimos anteriormente, a aplicação mais antiga conhecida para os grafos eulerianos é o problema das Pontes de Königsberg. Acredita-se que a teoria dos grafos teve início no século XVIII quando o famoso matemático Leonhard Euler resolveu esse problema [CO93]. A solução deste problema é considerado o primeiro teorema em teoria dos grafos (Teorema 1).

A idéia de muitos algoritmos, seqüenciais e paralelos, para obter circuitos de Euler é baseada no Teorema 1. Ele garante que qualquer grafo euleriano pode ser decomposto em um conjunto de circuitos disjuntos em arestas, denominado **Partição de Euler**. O próximo passo consiste em combinar circuitos vizinhos, ou seja, circuitos que compartilham um vértice, até que se obtenha um único circuito que envolva todas as arestas do grafo.

Este capítulo apresenta um algoritmo seqüencial e dois algoritmos paralelos no modelo PRAM (Atallah-Vishkin [GR88] e Cáceres *et al* [CDSS92]) para obter circuitos de Euler em grafos, que utilizam essa idéia. O algoritmo proposto por Cáceres *et al* [CDSS92] descreve uma implementação alternativa do algoritmo de Atallah-Vishkin [GR88], utilizando menos processadores e com estruturas mais simples.

3.1.2 Aplicação

O Problema de Carteiro Chinês

Uma das aplicações mais conhecidas de circuitos de Euler é o *Problema do Carteiro Chinês* [BM76, CO93]. Um carteiro pega a correspondência na agência dos correios, a entrega, e retorna à agência. Ele deve percorrer cada rua da sua área pelo menos uma vez. O carteiro deseja escolher a sua rota de modo que ele caminhe o mínimo possível (rota ótima). Este problema foi proposto pelo matemático chinês Kuan em 1962.

Esta situação pode ser modelada através de um grafo G onde as arestas correspondem às ruas e os vértices correspondem às intersecções entre as ruas, ou entre seções das ruas. Se G é euleriano, então qualquer circuito de Euler de G corresponde a uma rota ótima para o carteiro pois percorre cada aresta exatamente uma vez. O problema é facilmente resolvido neste caso pois existe um bom algoritmo para determinar circuitos de Euler em grafos eulerianos proposto por Fleury [BM76].

Uma alternativa para resolver o problema quando G não for euleriano é determinar um multigrafo H de tamanho mínimo que contenha G como seu grafo subjacente. Podemos obter H duplicando cada aresta de G, ou seja, substituindo cada aresta por um par de arestas paralelas. O resultado é um multigrafo euleriano H, pois H é conexo e grau(v) = 2$*$grau(v) para todo $v \in V(G)$. Um circuito de Euler de H produz um ciclo que contém todas as arestas de G [CO93].

3.2 Algoritmo Seqüencial

Dado um grafo euleriano $G = (V, E)$, o algoritmo apresentado nesta seção obtém um circuito de Euler de G. O algoritmo foi desenvolvido para o modelo seqüencial e possui complexidade $O(m+n)$, onde, n é o número de vértices e m é o número de arestas do grafo.

Algoritmo 1 Algoritmo seqüencial
Entrada: (1) V, o conjunto de vértices de G. (2) E, o conjunto de arestas de G.
Saída: Um circuito de Euler de G.

(1) nciclos := 0

(2) **enquanto** existem arestas em G **faça**

(2.1) Escolha um vértice v qualquer de G;

(2.2) Realize uma Busca em Profundidade em G, a partir de v, até achar um ciclo C;

(2.3) Retire de G as arestas de C;

(2.4) Retire de G os vértices de grau 0;

(2.5) nciclos := nciclos + 1;

(3) **enquanto** nciclos > 1 **faça**

(3.1) Escolha dois ciclos C_i e C_j que tenham algum vértice v em comum;

(3.2) Junte C_i e C_j em um único caminho fechado em v;

(3.3) nciclos := nciclos - 1;

— Fim do Algoritmo —

Busca em Profundidade. Vários problemas representados por grafos podem ser resolvidos efetuando uma busca no grafo. Seja um grafo $G = (V, E)$ e uma representação que indica para cada vértice, se ele foi visitado ou não. A seguir apresentamos uma versão recursiva do algoritmo de busca em profundidade que visita todos os vértices do grafo.

Algoritmo 2 Busca
Entrada: (1) V, o conjunto de vértices de G. (2) E, o conjunto de arestas de G.
Saída: Uma busca em profundidade de G.

(1) **para** cada vértice v de G **faça**

Marque v como não-visitado;

(2) **para** cada vértice v de G **faça**

se v não foi visitado **então**

Busca_Prof(v)

— Fim do Algoritmo —

Procedimento 1 Busca_Prof(v)
Entrada: (1) $v \in V$.
Saída: Uma busca em profundidade a partir de v.

(1) Marque v como visitado;

(2) **para** cada vértice w adjacente a v **faça**

se w não foi visitado **então**

Busca_Prof(w)

— Fim do Procedimento —

O Algoritmo 2 funciona com um grafo que não seja conexo. Se já sabemos que o grafo G é conexo, podemos chamar diretamente o Procedimento 1 escolhendo arbitrariamente um vértice inicial.

A complexidade do algoritmo seqüencial para circuitos de Euler em grafos é dada pelo resultado a seguir.

Teorema 2 *O Algoritmo 1 possui complexidade O(m+n).*

Demonstração: [Szw88]. □

Como vimos, o Algoritmo 1 utiliza como subrotina uma busca em profundidade. Infelizmente, não existe um algoritmo paralelo eficiente no modelo PRAM para a busca em profundidade. No caso de uma busca em profundidade lexicográfica, Gibbons e Rytter [GR88] apresentam o seguinte resultado:

Teorema 3 *O problema da busca em profundidade lexicográfica no modelo PRAM é P-completo.*

Demonstração: [GR88]. □

Existem algumas tentativas de paralelizar a busca em profundidade. Esses algoritmos geralmente utilizam uma estratégia de **banco de tarefas** (*working pool*) [Pac97, WA99], onde o número de troca de mensagens entre os processadores dependem de n. Os resultados obtidos não sugerem sua utilização para a exploração eficiente de um grafo.

Então, quando desenvolvemos um algoritmo paralelo para o problema de circuitos de Euler precisamos encontrar uma alternativa para a busca em profundidade.

3.3 Algoritmos Paralelos no Modelo PRAM

3.3.1 Algoritmo de Atallah-Vishkin

O algoritmo proposto por Atallah e Vishkin [GR88] encontra um circuito de Euler em um grafo euleriano. A implementação no modelo PRAM CREW possui complexidade de tempo paralelo $O(\log^2 n)$ utilizando $O(m)$ processadores.

Seja $G = (V, E)$ um grafo euleriano e $C = (C_1, C_2, ..., C_k)$ uma partição de Euler de G. O algoritmo obtém C através de duas ordenações lexicográficas aplicadas às arestas de G. Como o grafo é euleriano, o grau de saída de cada vértice é igual ao grau de entrada. As arestas são ordenadas duas vezes, a primeira pelo vértice de destino da aresta e a segunda pelo vértice origem da aresta.

Como a ordenação é lexicográfica, o número de arestas da primeira ordenação chegando ao vértice v é igual ao número de arestas saindo do vértice v na segunda ordenação. Isso garante que a i-ésima aresta (u, v) da primeira ordenação terá como correspondente a i-ésima aresta da segunda ordenação (v, w). Isso para todas as arestas. Logo as duas ordenações definem uma relação de sucessor. Novamente, pelo fato de que o grau de entrada é igual ao grau de saída, temos um ciclo ou um conjunto de ciclos.

Seja $H = (V', C', E')$ um grafo bipartido, tal que, $V' \subseteq V$ é o subconjunto dos vértices através dos quais passam mais de um ciclo de C; C' é um conjunto de vértices onde cada elemento corresponde a um ciclo de C; uma aresta de E' conecta um vértice $v \in V'$ a um vértice $u \in C'$, se o ciclo representado por u passa pelo vértice v.

Assim, o grafo bipartido H identifica os vértices de G através dos quais passam mais de um ciclo de C. O algoritmo constrói uma árvore geradora T do grafo bipartido H. T é usada para identificar os vértices através dos quais dois ou mais ciclos de C podem ser unidos. O algoritmo utiliza a técnica de Euler tour em árvores [GR88] para realizar a união dos circuitos.

Algoritmo 3 Algoritmo de Atallah-Vishkin
Entrada: (1) V, o conjunto de vértices de G. (2) E, o conjunto de arestas de G.
Saída: Um circuito de Euler de G.

(1) Obtenha uma partição de Euler do grafo G usando ordenações lexicográfica de suas arestas;

(2) Construa o grafo auxilar bipartido $H = (V', C', E')$

(2.1) Identifique, através de um representante do ciclo, a qual ciclo da partição de Euler cada aresta pertence;

(2.2) Construa o grafo bipartido $H = (V', C', E')$: V' contém os vértices através dos quais passam mais de um ciclo da partição; C' contém os representantes dos ciclos; um vértice u de V' é adjacente a um vértice v de C' se o ciclo representado por v passa pelo vértice u;

(3) Encontre uma árvore geradora T do grafo bipartido H

(4) Construa o grafo T' trocando cada aresta (i, j) de T por duas arestas anti-paralelas (i, j) e (j, i). O grafo T' contém um circuito de Euler porque o grau de entrada de cada vértice é igual ao grau de saída;

(5) Construa um ciclo L cujas arestas se alternam entre arestas de T' e arestas de G. L terá a seguinte propriedade: as arestas de G e as arestas de T' aparecerão em L na ordem, respectivamente, de um circuito de Euler em G e de um circuito de Euler em T'. Para tanto é utilizada a técnica de computar circuitos de Euler em árvores;

(6) Remova de L as arestas de T' deixando apenas as arestas que pertencem ao circuito de Euler de G;

— Fim do Algoritmo —

Teorema 4 *O algoritmo 3 computa corretamente um circuito de Euler num grafo euleriano utilizando o modelo PRAM CREW com complexidade de tempo paralelo $O(\log^2 n)$ utilizando $O(m)$ processadores.*

Demonstração: [GR88]. □

Uma descrição detalhada deste algoritmo, das rotinas de circuitos de Euler em árvores e árvore geradora, utilizadas pelo algoritmo, podem ser obtidas em [GR88]. A próxima seção descreve os passos do algoritmo de Cáceres *et al*[CDSS92].

3.3.2 Algoritmo de Cáceres *et al*

O algoritmo proposto por Cáceres *et al* [CDSS92] é uma variação do algoritmo anterior, utiliza o modelo PRAM CREW e possui complexidade de tempo paralelo $O(\log^2 n)$ utilizando $O(\frac{m}{\log m})$ processadores.

O algoritmo utiliza uma estrutura chamada **suporte** (*strut*). O suporte é usado para identificar diretamente os vértices em que uma operação denominada **costura** (*stitch*) deve ser aplicada. Esta operação une dois ou mais circuitos disjuntos em arestas, que passam por um determinado vértice, em um único circuito. Como vimos, qualquer circuito de Euler pode ser decomposto em uma coleção de circuitos disjuntos em arestas, então, a costura dos vértices apropriados produz um circuito de Euler.

Seja $G = (V, E)$ um grafo euleriano e $C = (C_1, C_2, ..., C_k)$ uma partição de Euler de G. Seja $V_1 \subseteq V$ o subconjunto dos vértices que possuem mais de um ciclo de C passando por eles. Seja C' um conjunto de vértices que possui uma correspondência um-para-um com os ciclos de C. Para rotular os vértices em C' escolhemos a menor aresta que pertence a cada um dos ciclos em C. A aresta escolhida é denominada **representante de ciclo**.

Construa $H = (V_1, C', E_1)$ um grafo bipartido auxiliar de G, tal que as arestas em E_1 conectam um vértice $v_i \in V_1$ com os vértices em C' que correspondem aos ciclos que passam por v_i.

Os vértices $v_i \in V_1$ de H devem ser ordenados, em ordem não-crescente, de seus graus e rotulados como $v_1, v_2, ..., v_{|V_1|}$. Isto ajuda a identificar um conjunto minimal de vértices onde a operação de costura deve ser executado.

Um **suporte** S em V_1 é uma floresta geradora de H tal que cada $c_i' \in C'$ incide em S com exatamente uma aresta de E_1, $e(v_j, c_i')$ é uma aresta de S se e somente se (v_k, c_i') não é uma aresta de H, para qualquer $v_k \in V_1$, $k < j$. Um suporte de V_1 é uma coleção de estrelas cujos centros são os vértices de V_1.

Um vértice $v \in V_1$ é chamado de **zero-diferença** se o grau de v em H é igual ao grau de v em S.

Sejam $e_1, e_2, ..., e_k$ as arestas de G que entram em um mesmo vértice $v_i \in V$, cada e_j pertencente a um ciclo diferente da partição de Euler C. Uma costura em v_i é uma operação que combina todos os ciclos que passam por v_i em um único ciclo.

Algoritmo 4 Algoritmo de Cáceres *et al*
Entrada: (1) V, o conjunto de vértices de G. (2) E, o conjunto de arestas de G.
Saída: Um circuito de Euler de G.

(1) Obtenha uma partição de Euler C do grafo G através de ordenações lexicográficas de suas arestas;

(2) Construa o grafo auxilar bipartido $H = (V_1, C', E_1)$

(2.1) Encontre o subconjunto dos vértices $V_1 \subseteq V$ que possuem mais de ciclo de C passando através dele;

(2.2) Identifique, através de um representante do ciclo, a qual ciclo da partição de Euler cada aresta pertence;

(2.3) Construa o grafo bipartido H;

(3) Ordene e rotule V_1, em ordem não-crescente, pelo grau de seus vértices;

(4) Determine os vértices a serem costurados

(4.1) Compute um suporte S em V_1;

(4.2) Para cada vértice $v_i \in V_1$ que não seja zero-diferença, adicione a S uma aresta arbitrária de $E_1 - S$ que seja incidente a v_i;

(4.3) Os vértices $v_i \in V_1$ tal que o grau de v_i em S é maior que 1 devem ser costurados;

(5) Para cada v que deve ser costurado, combine todos os circuitos que passam através de v em um único circuito;

(6) Repita os passos 2 a 5 até que um circuito de Euler seja encontrado;

— Fim do Algoritmo —

Corretude e complexidade do algoritmo

Lema 1 *Seja H = (V₁, C', E₁) um grafo bipartido e S um suporte em V₁ para H. Seja H' o grafo obtido, a partir de H, adicionando a S precisamente uma aresta de E₁− S incidente em cada vértice não zero-diferença de V₁. H' é acíclico. Além disso, se V₁ contém exatamente um vértice zero-diferença, então H' é uma árvore geradora de H.*

Demonstração [CDSS92]: S é uma coleção de estrelas cujo centro são vértices de V_1. Adicionando-se exatamente uma aresta de $E_1 - S$ incidente em cada vértices não zero-diferença de V_1, cada estrela cujo centro é um vértice não zero-diferença, v_j, está conectado com, no máximo, uma estrela diferente cujo centro é v_i e $j > i$. Então, H' não possui ciclo.

Pela definição de suporte, o grau de cada $c_i^{'}$ é exatamente 1, então, há exatamente $|C^{'}|$ arestas em S. Se V_1 contém exatamente um vértice não zero-diferença, então o número de arestas adicionadas é $V_1 - 1$. Portanto, H' possui $|V_1| + |C^{'}| - 1$ arestas, e H' é uma árvore geradora de H. □

Teorema 5 *O número de circuitos disjuntos em arestas restantes depois do*

Passo 5 é, no máximo, o número de circuitos disjuntos em arestas da iteração anterior dividido por dois.

Demonstração [CDSS92]: O grafo *H'* é uma floresta geradora com k árvores, onde k é o número de vértices zero-diferença em V_1. Seja v_i um vértice zero-diferença. Como todos os vértices em V_1 possuem mais de um circuito passando por eles, $D_{H'}[v_i] > 1$. Após a operação de costura, o número de circuitos que passam por v_i é reduzido para um. □

Por isso, o algoritmo repete os Passos 2 a 5 no máximo $O(\log p)$ vezes, no pior caso. Usando um algoritmo de ordenação inteira de complexidade de tempo $O(\log n)$ e $O(\frac{m}{\log \log m})$ processadores, temos o seguinte teorema [CDSS92]:

Teorema 6 *O algoritmo utiliza o modelo PRAM CREW e possui complexidade de tempo* $O(\log^2 n)$ *utilizando* $O(\frac{m}{\log m})$ *processadores.*

3.4 Conclusão

Neste capítulo, descrevemos o problema de circuitos de Euler em grafos e suas aplicações. Também descrevemos um algoritmo seqüencial e dois algoritmos paralelos no modelo PRAM para o problema.

O algoritmo seqüencial possui complexidade $O(m+n)$, onde n é o número de vértices e m é o número de arestas do grafo. Este algoritmo utiliza uma busca em profundidade como subrotina.

Os algoritmos paralelos descritos neste capítulo são o proposto por Atallah-Vishkin que possui complexidade de tempo $O(\log^2 n)$ utilizando $O(m)$ processadores, e o proposto por Cáceres *et al* que possui complexidade de tempo $O(\log^2 n)$ utilizando $O(\frac{m}{\log m})$ processadores. Ambos foram desenvolvidos para o modelo PRAM.

Os algoritmos de Atallah-Vishkin e Cáceres *et al* buscam uma alternativa para a busca em profundidade utilizada no algoritmo seqüencial, que não é implementada de forma eficiente em paralelo.

O algoritmo de Atallah-Vishkin utiliza uma árvore geradora como alternativa para a busca em profundidade. Já o algoritmo de Cáceres *et al* não utiliza explicitamente a computação de uma árvore geradora como subrotina. O algoritmo descarta os vértices e arestas que não serão utilizados na operação de costura, o que não ocorre com o algoritmo proposto por Atallah-Vishkin.

Capítulo 4

Algoritmo BSP/CGM para Computar Circuitos de Euler em Grafos

4.1 Introdução

Neste capítulo, apresentamos um algoritmo paralelo no modelo BSP/CGM para obtenção de circuitos de Euler em grafos. Como vimos, efetuar uma busca em profundidade em um grafo é um problema difícil de ser paralelizado. O algoritmo proposto neste trabalho utiliza idéias dos algoritmos de Cáceres *et al* [CDSS92] e Atallah-Vishkin [GR88] para o modelo PRAM.

Um dos objetivos de um algoritmo BSP/CGM é o de ir reduzindo o conjunto entrada de tal forma que ele possa ser movido para um único processador e resolvido seqüencialmente nesse processador.

No algoritmo que vamos apresentar, utilizamos a idéia de reduzir o conjunto de entrada proposto no algoritmo de Cáceres *et al* e os passos principais propostos no algoritmo de Atallah-Vishkin.

Como vamos utilizar o modelo BSP/CGM, temos que utilizar uma nova abordagem, visto que dispomos de apenas p processadores, e não é possível, neste problema, uma aplicação pura e simples do resultado do teorema de Brent [GR88].

4.2 O Algoritmo BSP/CGM

Nesta seção, apresentamos uma descrição dos principais passos do algoritmo BSP/CGM. Nas próximas seções, apresentamos a descrição detalhada de cada passo e de cada subrotina utilizada pelo algoritmo.

Algoritmo 5 Circuito de Euler BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Um grafo euleriano $G = (V, E)$ especificado através da lista de suas arestas armazenadas no vetor `edge`. O vetor `edge` está distribuído entre os p processadores, $\frac{m}{p}$ arestas e $\frac{n}{p}$ vértices em cada processador.
Saída: Um circuito de Euler de G.

(1) Obtenha uma partição de Euler do grafo G

- (1.1) Utilize um algoritmo BSP/CGM de ordenação para obter uma ordenação lexicográfica das arestas de G pelo vértice destino;
- (1.2) Utilize um algoritmo BSP/CGM de ordenação para obter uma ordenação lexicográfica das arestas de G pelo vértice origem;
- (1.2) Utilize um algoritmo BSP/CGM de Duplicação Recursiva para identificar, através de um representante de ciclo, a qual ciclo da partição de Euler cada aresta pertence;
- (1.3) Calcule o número de circuitos da Partição de Euler
  - (1.3.1) **Se** o número de circuitos for igual a 1 **então** o circuito de Euler foi encontrado e finalize o algoritmo.

(2) Construa o grafo bipartido H

- (2.1) Construa o grafo bipartido $H = (V^{'}, C^{'}, E^{'})$;
- (2.2) Elimine as arestas replicadas de H, se houver, em cada processador e também entre os processadores;
- (2.3) Elimine de H todas as arestas que incidem em vértices com grau menor que 2;

(3) Construa uma árvore geradora de H

- (3.1) Utilize um algoritmo BSP/CGM para obter uma árvore geradora $S = (V^{'}, C^{'}, E^{*})$ de H;
- (3.2) Elimine de S todas as arestas que incidem em vértices de $V^{'}$ com grau menor que 2;
- (3.3) Determine o número de arestas que restaram $|E^{*}|$;

(4) Realize a costura

(4.1) **Se** $|E^*| < O(\frac{n}{p})$ **então** envie as arestas a serem costuradas para um único processador e costure seqüencialmente;

(4.2) **Senão**

(4.2.1) Distribua E^* entre os processadores;

(3.5.2) Execute a costura distribuída das arestas;

— Fim do Algoritmo —

4.3 Técnicas Básicas BSP/CGM

Como vimos, o algoritmo BSP/CGM para computar o circuito de Euler utiliza como subrotinas os algoritmos de soma de prefixos, ordenação inteira, duplicação recursiva (*list ranking*), circuitos de Euler em árvores, componentes conexos e árvore geradora. Encontramos na literatura trabalhos que descrevem algoritmos BSP/CGM para esses problemas. Nesta seção, descrevemos esses algoritmos e discutimos algumas de suas características.

4.3.1 Soma de Prefixos

Considere a seqüência de n elementos $\{x_1, x_2, ..., x_n\}$ pertencentes a um conjunto S com uma operação binária associativa denotada por $*$. A **soma de prefixos** desta seqüência é composta pelas n somas parciais definidas por

$$s_i = x_1 * x_2 * ... * x_n, 1 \leq i \leq n$$

A solução para o problema de soma de prefixos no modelo BSP/CGM está descrito em [CMS01]. Seja A um vetor de ordem n, desejamos computar as somas de prefixos $S[i] = A[1] + ... + A[i], 1 \leq i \leq n$ com p processadores, onde $p \ll \frac{n}{p}$. Seja $r = \frac{n}{p}$. A é particionado em p blocos $A = (A_1, A_2, ..., A_p)$, onde cada A_i tem tamanho r. Para determinar as somas parciais $S[i]$, cada processador P_i computa a i-ésima soma $s_i = A_i[(i-1)r + 1] * ... * A_i[ir]$, para $1 \leq i \leq p$, e efetua um *broadcast* de s_i para todos os processadores P_i. Cada processador P_i computa as suas respectivas somas parciais $S[(i-1)r + 1], ..., S[ir]$, onde $S[(i-1)r + j] = \sum_{l=1}^{i-1} s_l + \sum_{j=1}^{r} A[(i-1)r + j]$.

Algoritmo 6 Soma de Prefixos BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) O número do processador

i. (3) O i-ésimo sub-vetor $B = A((i-1)r+1 : ir)$ de tamanho r, onde $r = \frac{n}{p}$.
Saída: Em cada processador P_i as somas de prefixos $S[(i-1)*r+j]$, $1 \leq j \leq \frac{n}{p}$.

(1) $s_i := B[1] + ... + B[r]$;

(2) **broadcast**$(s_i, p_j \neq i)$;

(3) $S[(i-1)*r] := s_1 + ... + s_{i-1}$;

(4) **para** $k = 1$ **até** r **faça**

$$S[(i-1)*r+k] := S[(i-1)*r+k-1] + B[k];$$

— Fim do Algoritmo —

A complexidade do Algoritmo 6, decorre do seguinte resultado:

Teorema 7 *O algoritmo Soma de Prefixos BSP/CGM computa as somas parciais de n elementos em tempo O($\frac{n}{p}$), com O(1) rodadas de comunicação.*

Demonstração [CMS01]: Cada processador P_i começa calculando $A_i[(i-1)r+1] + ... + A_i[ir]$ e armazena o valor resultante na variável local s_i. Isso é feito seqüencialmente sem a necessidade de nenhuma comunicação. No Passo 2, os processadores fazem um *broadcast* de s_i para os demais processadores. Essa comunicação pode ser feita em uma única rodada de comunicação. No Passo 3, cada processador calcula, seqüencialmente, o valor da soma dos $A(1 : (i-1)r)$ elementos do vetor e com esse valor, no Passo 4, são computadas as somas de prefixo dos $A((i-1)r+1 : ir)$ elementos do vetor A (com relação aos $(i-1)r+j, 1 \leq j \leq r$ elementos de A). Esses passos podem ser feitos sem necessidade de comunicação entre os processadores.

A computação local executada por cada processador P_i, $1 \leq i \leq p$ consiste de r operações nos Passos 1 e 4. No Passo 3, cada processador P_i executa $i-1, 1 \leq i \leq p$ operações. Portanto, o tempo de computação do algoritmo é $O(\frac{n}{p})$. Por outro lado, cada processador P_i tem que receber $p-1$ mensagens, todas no mesmo superpasso (BSP) ou rodada (CGM), logo o algoritmo termina com $O(1)$ rodadas de comunicação. □

4.3.2 Ordenação inteira

Um dos procedimentos básicos que será utilizado no algoritmo BSP/CGM de circuitos de Euler é a ordenação. Necessitamos de um algoritmo de ordenação

que utilize no máximo $O(\frac{n}{p})$ memória local com um número adequado de rodadas de comunicação.

O algoritmo de ordenação, no modelo PRAM, de Cole [Col88] usa tempo $O(\log\ n)$ com n processadores. Esse algoritmo foi adaptado por Goodrich [Goo96, Goo99] que obteve um algoritmo que utiliza $O(1)$ rodadas de comunicação desde que $\frac{n}{p} \geq p^{\varepsilon}$, $\varepsilon > 0$. Embora esse algoritmo seja ótimo do ponto de vista teórico, sua implementação envolve uma grande quantidade de detalhes.

Como estamos trabalhando com inteiros, podemos utilizar algoritmos mais simples de serem implementados, tais como o algoritmo baseado no *Bucket Sort* [GV94, Fer99, CMS01] ou o algoritmo de ordenação de inteiros baseado no *Radix Sort* [CD99], ambos utilizam estratégias para escolher um conjunto de separadores (*split*) para aplicação do algoritmo.

O único inconveniente desses algoritmos é que os dados podem não terminar igualmente distribuídos entre os processadores. Isso pode ser resolvido facilmente com a utilização de um algoritmo de soma de prefixos e uma redistribuição dos dados.

Esses dois algoritmos de ordenação utilizam no máximo $O(1)$ rodadas de comunicação e usam no máximo $O(\frac{n}{p})$ memória local. A complexidade da computação local é $O(\frac{n \log n}{p})$ e $O(\frac{n}{p})$, respectivamente.

Vamos descrever o algoritmo BSP/CGM de ordenação baseado no *Bucket Sort*. Esse algoritmo é denominado *Split Sort*. Maiores detalhes podem ser obtidos em [GV94, Fer99, CMS01].

Algoritmo 7 Split Sort BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Um vetor A de n elementos. Os elementos do vetor A estão distribuídos entre os p processadores ($\frac{n}{p}$ elementos por processador).
Saída: Todos os elementos ordenados dentro de cada processador e por processador, ou seja, se $i < j$, temos que os elementos em p_i são menores que os elementos pertencentes a p_j.

(1) Compute um conjunto divisor $S := \{s_1,\ s_2,\ ...\ s_{p-1}\}$;

(2) **broadcast**(S, p_0);

(3) Particionar os elementos de p_i em buckets B_j^i de acordo com S;

(4) **send**(B_j^i, p_j);

(5) Ordena $B_i^k := B_i^0 \cup B_i^1 \cup ... \cup B_i^{p-1}$;

— Fim do Algoritmo —

Como vimos anteriormente, necessitamos fazer uma distribuição balanceada global, tal que, todos os inteiros sejam distribuídos igualmente entre os processadores sem alterar a sua ordem. Isto pode ser efetuado através de uma soma de prefixos seguido de uma redistribuição dos inteiros.

Agora vamos descrever um algoritmo BSP/CGM para computar o conjunto S (conjunto *splitter*), que utiliza apenas $O(p)$ espaço de memória por processador. O método é baseado no particionamento da entrada em p subconjuntos do mesmo tamanho como segue.

Definição 16 *A **mediana** de um conjunto ordenado de n números é o* $(\frac{n+1}{2})$*-ésimo elemento de n para n ímpar ou a média do* $(\frac{n}{2})$*-ésimo com o* $(\frac{n+1}{2})$*-ésimo elemento se n for par.*

Definição 17 *Os p-**quartis** de um conjunto ordenado A de tamanho n são os p - 1 elementos, de índices,* $\frac{n}{p}, \frac{2n}{p}, ..., \frac{(p-1)n}{p}$*, que dividem A em p partes de igual tamanho.*

Procedimento 2 p-quartis seqüencial
Entrada: (1) Um vetor A com n elementos.
Saída: O conjunto A dividido em p-quartis.

(1) Compute a mediana de A;

(2) Usando a mediana, divida A em dois conjuntos A_1 e A_2;

(3) Aplique o algoritmo recursivamente, até que p-1 divisores sejam encontrados;

— Fim do Procedimento —

Procedimento 3 p-quartis BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Um vetor A com n elementos. Os elementos de A estão distribuídos entre os p processadores ($\frac{n}{p}$ elementos por processador).
Saída: O conjunto A dividido em p-quartis.

(1) $Q_i := p$-quartis(A_i);

(2) **send**(Q_i, p_0);

(3) **se** $i = 0$ **então** $Q :=$ Ordena($Q_0 \cup Q_1 \cup ... \cup Q_{p-1}$);

(4) $S :=$ p-quartis(Q);

(5) **broadcast**(S, P_i);

— Fim do Procedimento —

A corretude e complexidade do algoritmo *Split Sort*, decorrem dos seguintes resultados:

Teorema 8 *O algoritmo para a determinação dos p-quartis de um conjunto de n elementos é executado no modelo BSP/CGM com p processadores em O(1) rodadas de comunicação e tempo de computação local de* $O(\frac{n \log p}{p})$*, onde* $\frac{n}{p} = \Omega(p^2)$.

Demonstração: [CMS01]. □

Corolário 1 *A ordenação de um conjunto de n elementos pode ser executada no modelo BSP/CGM com p processadores em O(1) rodadas de comunicação e tempo de computação local de* $O(\frac{n \log n}{p})$*, onde* $\frac{n}{p} = \Omega(p)$.

Demonstração: [CMS01]. □

O algoritmo 7 pode ser adaptado para ordenação de inteiros usando o mesmo número de rodadas de comunicação com computação local $O(\frac{n}{p})$.

4.3.3 List ranking e Duplicação Recursiva

Uma **lista ligada** é um conjunto de nós, tal que, cada nó aponta para um outro nó denominado **sucessor**. O problema do ***list ranking*** consiste em determinar a **distância** de cada nó até o último nó da lista. Uma generalização do problema considera a lista decomposta em um conjunto de sublistas e o problema consiste, então, em determinar para todos os nós a distância até o último nó da sua sublista. Muitos algoritmos paralelos para problemas em grafos utilizam o *list ranking* como subrotina.

Diferentemente do algoritmo seqüencial para esse problema, que pode ser facilmente resolvido de forma eficiente, o algoritmo paralelo na maioria dos casos, não leva a uma implementação eficiente [LG00]. Segundo [CMS01], isto ocorre porque o número de nós da lista cujos sucessores não estão armazenados no mesmo processador pode variar de 0 a $\frac{n}{p}$. Mesmo no caso em que

todos os sucessores estejam no mesmo processador no passo inicial, nada garante que com a aplicação da duplicação recursiva isso continuará ocorrendo no passo seguinte. Pode ocorrer que a cada iteração, pelo menos um dos processadores sempre necessite efetuar uma comunicação para obter o sucessor de um de seus elementos. Logo o número de rodadas de comunicação para obter o *list ranking* pode chegar a $O(\log n)$.

Cáceres *et al* [CDF+97, DFC+02] apresentam um algoritmo CGM para o problema do *list ranking*. Lassous e Gustedt [LG00] implementam e fazem a análise de vários algoritmos paralelos. O algoritmo de Cáceres *et al* utiliza $O(\log p)$ rodadas de comunicação e $O(\frac{n}{p})$ tempo de computação local por rodada.

Seja L uma lista ligada de tamanho n representada pelo vetor $s[1...n]$. Para cada $i \in \{1...n\}$, $s[i]$ é um ponteiro para o elemento seguinte a i na lista L. Denominamos i e $s[i]$ por **vizinhos**. O último elemento da lista L, λ é aquele onde $s[\lambda] = \lambda$. A **distância** entre i e j, $d_L(i, j)$, é o número de nós entre i e j mais um. O problema do *list ranking* consiste em computar para cada $i \in L$ a distância entre i e λ, $rank_L(i) = d_L(i, \lambda)$.

Definição 18 *Um conjunto **r-ruling** $L^{'}$ de L é definido como um subconjunto de elementos selecionados de L que possuem as seguintes propriedades: (1) Elementos vizinhos não são selecionados. (2) A distância entre qualquer elemento não-selecionado ao próximo elemento selecionado é no máximo r.*

Para cada $i \in L$, seja $s_{L'}[i]$ o próximo elemento selecionado $j \in L^{'}$. Representamos um conjunto *r-ruling* de $L^{'}$ como uma lista ligada onde a cada elemento $i \in L^{'}$ é associado um ponteiro $s_{L'}[i]$ e o valor $w(i) = d_L(i, s_{L'}[i])$ representa a distância entre i e $s_{L'}(i)$. O problema do *list ranking* em $L^{'}$ com pesos $w(.)$ refere-se a computar para cada $i \in L^{'}$ a soma dos pesos $w(j)$ de todos os nós $j \in L^{'}$ entre i e λ.

Algoritmo 8 List Ranking BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Uma lista ligada L de tamanho n. Cada processador armazena $\frac{n}{p}$ elementos $i \in L$ e seus respectivos ponteiros $s[i]$.
Saída: Para cada elemento i a sua posição $rank_L(i)$ em L.

(1) Compute um conjunto $O(p^2)$-*ruling*, R, de tamanho $|R| = O(\frac{n}{p})$.

(2) R é enviado para todos os processadores.

(3) Cada processador, seqüencialmente, executa o *list ranking* em R com pesos $w(.)$, computando para cada $j \in R$ sua posição $rank_L(j)$ em L.

(4) Cada elemento $i \in L$ - R está a uma distância, de no máximo, $O(p^2)$, do seu próximo elemento $s_R[i]$ em R. Usando o algoritmo PRAM de *pointer jumping*, compute para cada $i \in L$ - R a distância $d_L(i,\ s_R[i])$ até o próximo elemento $s_R[i]$ em R.

(5) Cada processador localmente computa a distância de seus elementos $i \in L$ - R da seguinte maneira: $rank_L(i) = d_L(i,\ s_R[i]) + rank_L(s_R[i])$.

— Fim do Algoritmo —

Segundo Cáceres *et al* [CDF+97], os passos 2−5 podem ser implementados em BSP/CGM com $O(\log p)$ rodadas de comunicação e $O(\frac{n}{p})$ computação local; o Passo 4 simula os passos do algoritmo *pointer jumping* PRAM em BSP/CGM, cada um desses passos pode ser implementado em $O(1)$ rodadas aplicando o algoritmo de ordenação de Goodrich [Goo96] para $\frac{n}{p} \geq p^{\varepsilon}$.

A parte mais difícil do algoritmo é a computação do conjunto $O(p^2)$-*ruling* R de tamanho $O(\frac{n}{p})$, a seguir descrevemos uma técnica chamada ***deterministic list compression*** proposta por Cáceres *et al* [CDF+97] para computar o conjunto $O(p^2)$-*ruling* R utilizando $O(\log p)$ rodadas de comunicação.

Definição 19 *Definimos um* ***intervalo-s*** *de tamanho k como uma seqüência* $I = (i_1,..., i_k)$ *de elementos com* $s[i_j] = 1 \leq j \leq k-1$. *Os dois vizinhos de um intervalo-s I são dois elementos da lista* n_1, n_2 *que não pertencem a I tal que* $s[n_1] = i_1$ *e* $s[i_k] = n_2$. *Chamamos o intervalo-s maximal* $I = (i_1,..., i_k)$ *de elementos da lista que estão todos armazenados no mesmo processador de* ***intervalo-s local***. *Um intervalo-s maximal* $I = (i_1,..., i_k)$ *de elementos da lista onde não há dois elementos subseqüentes* i_j, i_{j+1} *armazenados no mesmo processador é chamado de* ***intervalo-s não-local***.

Utilizamos o rótulo $l(i)$ para denotar o número do processador que armazena o elemento i da lista. Um elemento $s[i]$ da lista é chamado de **máximo local** se $l(s^{-1}[i]) < l(i) > l(s[i])$ onde $l(s^{-1}[i])$ é o elemento da lista j, tal que, $s[j] = i$.

Procedimento 4 BSP/CGM $O(p^2)$-Ruling Set
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Uma lista ligada L de tamanho n. Cada processador armazena $\frac{n}{p}$ elementos $i \in L$ e seus respectivos ponteiros $s[i]$.
Saída: Um conjunto de nós de L representando um conjunto $O(p^2)$-*ruling* de tamanho $O(\frac{n}{p})$.

(1) Cada processador localmente marca todos os elementos de sua lista como não-selecionados.

(2) Usando ordenação global, todos os processadores determinam para cada elemento i da lista seus dois vizinhos $s^{-1}[i]$ e $s[i]$. Então cada processador localmente executa para cada elemento i da lista local:

se $l(s^{-1}[i]) < l(i) > l(s[i])$ **então** marque i como selecionado.

(3) Cada processador localmente determina seu intervalo-s local. Usando ordenação global, todos os processadores determinam os dois vizinhos de cada intervalo-s local. Cada processador examina localmente todos os seus intervalos-s locais. Para cada intervalo-s local de tamanho maior que dois, todo segundo elemento é marcado como selecionado. Se um intervalo-s local tem tamanho dois e nenhum dos seus vizinhos tem um rótulo menor, então os dois elementos do intervalo-s local são marcados como não-selecionado.

(4) **para** $k = 1$ **até** $\log p$ **faça**

(4.1) Usando ordenação global, todos os processadores determinam para cada elemento i da lista $s[i]$ e $s[s[i]]$. Depois cada processador, localmente, executa para cada elemento i da lista local:

se $s[i]$ não está selecionado **então** $s[i] := s[s[i]]$.

(4.2) Usando a ordenação global, todos os processadores determinam, para cada elemento i da lista, seus vizinhos $s^{-1}[i]$ e $s[i]$. Então, cada processador localmente executa para cada elemento i da lista local:

se ($s^{-1}[i]$, i e $s[i]$ não estão selecionados) **and not** ($l(s^{-1}[i])$ $< l(i) > l(s[i])$) **and** ($l(s^{-1}[i]) \neq l(i)$) **and** ($l(i) \neq (l(s[i]))$) **então** marque i como não-selecionado.

(4.3) Cada processador localmente determina seu intervalo-s local. Usando ordenação global, todos os processadores determinam os dois vizinhos de cada intervalo-s local. Cada processador examina localmente todos os seus intervalos-s locais. Para cada intervalo-s local de tamanho maior que dois, todo segundo elemento é marcado como não-selecionado. Se um intervalo-s local tem tamanho dois e nenhum dos seus vizinhos tem um rótulo menor, então os dois elementos do intervalo-s local são marcados como não-selecionado.

(5) O processador que armazena o último elemento λ de L marca λ como selecionado.

— Fim do Procedimento —

O Teorema 9 garante a complexidade do algoritmo BSP/CGM para computar o *list ranking*. Inicialmente, vamos demonstrar que o conjunto de elementos selecionados ao final do Procedimento 4 é de tamanho $O(\frac{n}{p})$.

Lema 2 *Após a k-ésima iteração no Passo 4, não existem mais que dois elementos selecionados entre quaisquer intervalo-s de tamanho* 2^k *na lista original L.*

Demonstração [CDF+97, DFC+02]: Por indução em k. O lema é trivial para $k = 1$. Seja S um intervalo-s de tamanho 2^k na lista original L e sejam S_1 e S_2 a primeira e a segunda metade de S, respectivamente. Assuma que depois da iteração $k-1$ no Passo 4, S_1 e S_2 contém no máximo dois elementos selecionados cada. Denote por $e_1, ..., e_4$ os quatro elementos (no máximo) selecionados, ordenados em relação a L. A distância entre esses elementos, em relação a L, é no máximo 2^k. Considere agora a iteração k no Passo 4. Após o Passo 4.1, quaisquer dois elementos selecionados que possui uma distância de no máximo 2^k e o elemento não-selecionado entre eles (em relação a L) estão diretamente conectados por uma ligação que é representada pelo vetor s atual. Então, $s[e_1] = e_2, s[e_2] = e_3, s[e_3] = e_4$. Há no máximo quatro possíveis casos (não simétricos) para aplicar os Passos 4.2 e 4.3 em $e_1, ..., e_4$. Se $e_1, ..., e_4$ são dois a dois distintos, então apenas o Passo 4.2 é aplicado. Se $e_1, ..., e_4$ são todos iguais, então o Passo 4.3 é aplicado. Se $e_1, ..., e_3$ são dois a dois distintos e $e_3 = e_4$, então o Passo 4.2 é aplicado em $e_1, ..., e_3$ e o Passo 4.3 em e_3, e_4. O outro caso possível é $e_1 \neq e_2 = e_3 \neq e_4$. Então, o Passo 4.2 é aplicado em e_1 e e_4, e o Passo 4.3 aplicado em e_2 e e_3. Em qualquer caso, após os Passos 4.2 e 4.3, no máximo dois elementos em $e_1, ..., e_4$ ainda estão selecionados. □

Agora vamos mostrar que elementos subseqüentes selecionados ao final do Procedimento 4 tem, no máximo, distância $O(p^2)$.

Lema 3 *Após cada execução do Passo 4.3, a distância de dois elementos subseqüentes selecionados, com respeito aos ponteiros atuais (representado pelo vetor s), é no máximo O(p).*

Demonstração [CDF+97, DFC+02]: Considere dois elementos subseqüentes e_1 e e_2 selecionados. Existem três casos possíveis: (1) e_1, e_2 e todos os elementos entre eles, em relação a s, possuem o mesmo rótulo. (2) Para e_1, e_2 e todos os elementos entre eles, em relação a s, qualquer par de elementos consecutivos possuem rótulos diferentes. (3) O caso onde alguns pares de elementos consecutivos possuem rótulos iguais. Observe que para o Caso (3)

é impossível que três ou mais elementos consecutivos tenham o mesmo rótulo, porque um deles será um elemento selecionado (Passo 4.3). No Caso (1) a distância entre e_1 e e_2 é, no máximo, 2 devido ao Passo 4.3. No Caso (2), há no máximo p rótulos diferentes. Conseqüentemente, por [CV88], temos que a distância é, no máximo, $O(p)$. O Caso (3) é equivalente ao Caso (2) exceto por um fator de dois. □

Lema 4 *Após a k-ésima iteração do Passo 4.3, dois elementos subseqüentes com respeito aos ponteiros atuais (representado pelo vetor s) tem distância $O(2^k)$ com respeito a lista original L.*

Demonstração [CDF+97, DFC+02]: Temos que somente k duplicações recursivas foram efetuadas no Passo 4.1. □

Lema 5 *Quaisquer dois elementos subseqüentes selecionados não distam um do outro mais que $O(p^2)$ com respeito a lista original L.*

Demonstração [CDF+97, DFC+02]: Lema 3 e Lema 4. □

Lema 6 *O Procedimento 4 determina um $O(p^2)$-ruling set de tamanho $O(\frac{n}{p})$ em $O(\log p)$ rodadas de comunicação com $O(\frac{n}{p})$ computação local por rodada.*

Demonstração [CDF+97, DFC+02]: A corretude do Procedimento 4 segue dos Lemas 2 a 5. O algoritmo de ordenação descrito em [Goo96] utiliza $O(1)$ rodadas com $O(\frac{n}{p})$ computação local por rodada. A comunicação utilizada pelo Procedimento 4 consiste de duas ordenações globais nos Passos 2 e 3, e $3 \log p$ ordenações globais no Passo 4. Toda computação local em cada rodada pode ser feita em tempo linear. □

Em resumo, temos que

Teorema 9 *O problema do list ranking para uma lista ligada com n vértices pode ser resolvido em BSP/CGM, com p processadores e $O(\frac{n}{p})$ memória local por processador onde $\frac{n}{p} \geq p^{\varepsilon} (\varepsilon > 0)$, em $O(\log p)$ rodadas de comunicação e $O(\frac{n}{p})$ computação local por rodada.*

4.3.4 Circuito de Euler em Árvores

Seja $T = (V, E)$ uma árvore não-orientada e $T^* = (V, E^*)$ um grafo orientado com $E^* = \{\langle v, w\rangle, \langle w, v\rangle \mid (v, w) \in E\}$. Então T^* é euleriano pois o grau de entrada de v é igual ao grau de saída de v para qualquer vértice $v \in V$.

O problema do circuito de Euler para T consiste em computar para T^* um caminho que percorre cada aresta exatamente uma vez e retorna ao ponto inicial, definindo para cada vértice a sua ordem neste caminho.

O Teorema 10 garante a complexidade do algoritmo de circuito de Euler em árvores.

Teorema 10 *O circuito de Euler de uma árvore T com n vértices pode ser computado em BSP/CGM utilizando p processadores e $O(\frac{n}{p})$ memória local por processador, $\frac{n}{p} \geq p^{\epsilon}$ ($\epsilon > 0$), usando $O(\log p)$ rodadas de comunicação e $O(\frac{n}{p})$ computação local por rodada.*

Demonstração [CDF+97]: Nós computamos T^* e sua lista de adjacências duplicando todas as arestas de T e aplicando uma ordenação [Goo96]. Além disso, construímos a lista de adjacências para cada vértice circular aplicando o *list ranking*. Para cada arestas (i, j) de T^* seja $prox(i, j)$ o sucessor desta entrada na respectiva lista de adjacências. Agora, aplicamos o conhecido método de Tarjan e Vishkin [TV85] para definir uma ordenação das arestas de T^* que atribui a cada aresta (i, j) a aresta $prox(j, i)$ como sucessora. □

4.3.5 Árvore Geradora

Vamos utilizar o algoritmo para os problemas de componentes conexos e floresta geradora descrito por Cáceres *et al* [CDF+97] para calcular uma árvore geradora de um grafo.

O Algoritmo 9 computa os componentes conexos de um grafo $G = (V, E)$, com p processadores e $O(\frac{n+m}{p})$ memória local por processador. O Passo 1 simula o algoritmo PRAM proposto por Shiloch e Vishkin [SV82], mas utiliza apenas $\log p$ iterações do laço principal ao invés das $O(\log n)$ iterações do algoritmo original. O Passo 2 converte todas as árvores resultantes em estrelas. O grafo $G' = (V', E')$ obtido possui no máximo $O(\frac{n}{p})$ vértices. Assim, V' pode ser enviado para todos os processadores. E' ainda pode ter tamanho $O(m)$ e é distribuído entre os p processadores. E_i se refere as arestas de E' armazenadas no processador i. A floresta geradora de $(V', E_i \cup E_j)$

para dois conjuntos E_i, E_j possui tamanho $O(|\,V'|) = O(\frac{n}{p})$. Portanto, cada floresta geradora computada no Passo 3 pode ser armazenada na memória local de um processador. Combinamos pares de florestas geradoras até que, após $\log p$ rodadas, a floresta geradora de G' está armazenada no processador P_0. No Passo 4, todos os processadores atualizam as informações sobre os componentes conexos parciais obtidos no Passo 1.

Algoritmo 9 Componentes Conexos BSP/CGM
Entrada: (1) p processadores p_0, p_1, ..., p_{p-1}. (2) Um grafo $G = (V, E)$ com n vértices e m arestas. Cada processador armazena $\frac{n}{p}$ vértices e $\frac{m}{p}$ arestas.
Saída: Os componentes conexos de G representados pelos valores `parent(v)` para todos os vértices $v \in V$.

(1) Usando ordenação [Goo96], simule $\log p$ iterações do laço principal do algoritmo PRAM de Shiloch e Vishkin [SV82].

(2) Use o algoritmo de circuito de Euler em árvores para converter todas as árvores resultantes em estrelas. Para cada $v \in V$, atribua a *parent(v)* a raiz da estrela que contém v. Seja $G' = (V', E')$ o grafo obtido. Distribua G' tal que cada processador armazene todo o conjunto V' e um subconjunto de $\frac{m}{p}$ arestas de E'. Seja E_i as arestas armazenadas no processador i, $0 \leq i \leq p-1$.

(3) Atribua o estado *ativo* para todos os processadores

para k = 1 **até** $\log p$ **faça**

Particione os processadores ativos em grupos de tamanho dois.

para cada grupo P_i, P_j de processadores ativos, $i < j$ **faça em paralelo**

Processador P_j envia seu conjunto de arestas E_j para o processador P_i.

Atribua ao processador P_j o estado `passivo`.

Processador P_i computa a floresta geradora (V', E_s) do grafo `SF` $= (V', E_i \cup E_j)$ e atribui $E_i := E_s$.

Atribua a todos os processadores o estado `ativo` e envie E_0 para todos.

(4) Cada processador P_i computa seqüencialmente os componentes conexos do grafo $G'' = (V', E_0)$. Para cada vértice `v` de V' seja `parent'(v)` o menor rótulo `parent(w)` de um vértice `w` $\in V'$ que está no mesmo componente conexo em $G'' = (V', E_0)$. Para cada vértice `u` $\in V$ armazenado no processador P_i faça `parent(u)` := `parent'(parent(v))`.

— Fim do Algoritmo —

O Teorema 11 garante a complexidade do algoritmo *Componentes Conexos BSP/CGM.*

Teorema 11 *O Algoritmo 9 computa os componentes conexos e a floresta geradora de um grafo G = (V, E) com n vértices e m arestas, usando O(*$\log p$*) rodadas de comunicação e O(*$\frac{n+m}{p}$*) computação local por rodada.*

Demonstração: [CDF+97]. □

Com uma modificação simples, o Algoritmo 9 pode computar uma árvore geradora de um grafo. Essa modificação pode ser feita sem que nenhum custo adicional, ou seja, mantendo-se a complexidade do algoritmo.

4.4 Descrição do Algoritmo BSP/CGM

Agora vamos fazer uma descrição dos passos do algoritmo BSP/CGM para computar um circuito de Euler de um grafo euleriano. Este algoritmo utiliza os mesmos passos iniciais dos algoritmos de Atallah-Vishkin e Cáceres *et al.*

Considere $G = (V, E)$ um grafo euleriano e $C = C_1, C_2, ..., C_k$ uma partição de Euler de G. Utilizamos o método descrito pelo algoritmo de Atallah-Vishkin para obter C.

Seja $H = (V', C', E')$ o grafo bipartido, definido pelo algoritmo de Atallah-Vishkin, que identifica os vértices de G através dos quais passam mais de um ciclo de C, e seja S uma árvore geradora de H. O algoritmo BSP/CGM utiliza uma estrutura denominada **esteio**, para identificar os vértices através dos quais dois ou mais ciclos de C devem ser unidos.

Definição 20 *Um **esteio** $S^* = (V^*, C^*, E^*)$ é um subgrafo de uma árvore geradora S de um grafo bipartido $H = (V', C', E')$, tal que, o grau de cada vértice em V^* é maior ou igual a 2.*

O algoritmo utiliza a operação de costura, definida pelo algoritmo de Cáceres *et al*, para realizar a união de dois ou mais ciclos que passam através de um mesmo vértice. No algoritmo utilizamos as estruturas de dados descritas a seguir.

Os vetores `edge`, `succ` e `cycrep` cada um com m elementos, representando as arestas (u, v) de G.

O vetor `bipart[u][v][st][sp]`, onde os campos `u` e `v` representam os vértice origem e destino da aresta respectivamente, o campo `st` indica se a aresta pertence ou não a árvore geradora e o campo `sp` indica se a aresta pertence ou não ao esteio.

O vetor `vertice[v][grau]`, onde o campo `v` representa um vértice de grafo G e o campo `grau` representa o grau do vértice `v` no grafo bipartido.

Para facilitar o entendimento do algoritmo, vamos acompanhar cada passo através de um exemplo. Para cada passo do algoritmo, o exemplo mostra o conteúdo de cada processador e também uma representação gráfica do grafo. A Figura 4.1 mostra o grafo euleriano de entrada G que possui $m = 32$ arestas e $n = 16$ vértices.

Considere um sistema com quatro processadores P_0, P_1, P_2 e P_3. Cada processador recebeu $\frac{m}{p} = 8$ arestas. A Tabela 4.1 mostra a distribuição inicial das arestas em cada processador P_i. As arestas foram distribuídas pelo processador raiz, que envia para o processador P_0 as $\frac{m}{p}$ primeiras arestas de `edge`, para P_1 as $\frac{m}{p}$ arestas seguintes, e assim sucessivamente, até enviar a $\frac{m}{p}$ últimas arestas para P_3.

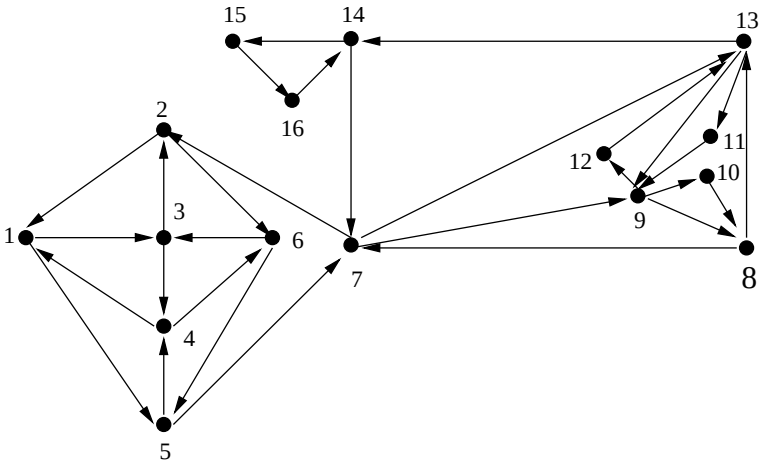


Figura 4.1: Grafo G de entrada

Algoritmo 10 Circuito de Euler BSP/CGM

Entrada: (1) p processadores $p_0, p_1, ..., p_{p-1}$. (2) Grafo euleriano $G = (V, E)$ especificado através da lista de suas arestas armazenadas no vetor `edge`. O vetor `edge` está distribuído entre os p processadores, $\frac{m}{p}$ arestas e $\frac{n}{p}$ vértices em cada processador.

P_0	(1,5) (2,1) (1,3) (2,6) (3,4) (6,5) (4,1) (7,2)
P_1	(6,3) (4,6) (5,4) (3,2) (5,7) (8,7) (14,7) (7,9)
P_2	(8,13) (10,8) (7,13) (9,12) (11,9) (13,9) (13,11) (9,10)
P_3	(12,13) (9,8) (13,14) (16,14) (14,15) (15,16)

Tabela 4.1: Distribuição inicial das arestas de G

Saída: `succ(i)` contém o sucessor da aresta `edge(i)` no circuito de Euler obtido pelo algoritmo.

Passo 1 Obtenha uma Partição de Euler

(1.1) Ordene as arestas pelo vértice destino

(1.1.1) Obtenha uma ordenação lexicográfica dos elementos de `edge`, definida como: dadas duas arestas $(i,\ j)$ e $(k,\ l)$ então $(i,\ j) < (k,\ l)$ se $j < l$ ou ($j = l$ e $i < k$). A ordenação é feita através de um algoritmo BSP/CGM para ordenação de inteiros. Ao final do algoritmo todas as arestas estarão ordenadas em cada um dos processadores de modo que o primeiro processador conterá as primeiras $\frac{m}{p}$ arestas da ordenação e assim sucessivamente.

(1.1.2) Depois da ordenação aplique um algoritmo BSP/CGM de soma de prefixos para atribuir a cada aresta um índice global na ordenação, ou seja, as arestas do primeiro processador serão numeradas de 0 à $\frac{m}{p} - 1$, e assim sucessivamente.

(1.2) Ordene as arestas pelo vértice origem

Copie o vetor `edge` para o vetor `succ`, e então reordene seus elementos de acordo com a seguinte ordenação lexicográfica: $(i,\ j) < (k,\ l)$ se $i < k$ ou ($i = k$ e $j < l$). A ordenação é feita através de um algoritmo BSP/CGM para ordenação de inteiros. Ao final do algoritmo o vetor `succ` conterá a aresta sucessora de cada aresta em `edge`. Juntos `edge` e `succ` definem um conjunto de ciclos disjuntos em arestas (partição de Euler) para um grafo euleriano orientado. Em um ciclo, a aresta seguinte à aresta armazenada em `edge(i)` é encontrada em `succ(i)`.

(1.3) Determine o ciclo ao qual pertence cada aresta

Crie o vetor `cycrep` que conterá o representante do ciclo ao qual cada aresta pertence. Cada ciclo será representado por uma de suas

arestas. Para dar nome ao ciclo será escolhida a menor aresta na seguinte ordenação lexicográfica: $(i, j) < (k, l)$ se $i < k$ ou ($i = k$ e $j < l$). Aplica-se um algoritmo BSP/CGM de duplicação recursiva para obter o vetor `cycrep`. Ao final do algoritmo `cycrep(i)` conterá o representante do ciclo ao qual pertence a aresta `edge(i)`.

(1.4) Determinar o número de ciclos da partição de Euler

(1.4.1) Cada processador calcula quantas de suas arestas são representantes de ciclos armazenando o valor em `cyc`.

(1.4.2) Todos os processadores enviam o `cyc` calculado para o processador raiz.

(1.4.3) O processador raiz acumula os `cyc` de todos os processadores em `ncyc`.

(1.4.4) O processador raiz envia `ncyc` para todos os demais processadores.

(1.4.5) **se** `ncyc` $= 1$ **então** o circuito de Euler foi encontrado e o algoritmo é finalizado.

P_0	(2,1) (4,1) (3,2) (7,2) (1,3) (6,3) (3,4) (5,4)
P_1	(1,5) (6,5) (2,6) (4,6) (5,7) (8,7) (14,7) (9,8)
P_2	(10,8) (7,9) (11,9) (13,9) (9,10) (13,11) (9,12) (7,13)
P_3	(8,13) (12,13) (13,14) (16,14) (14,15) (15,16)

Tabela 4.2: Arestas ordenadas pelo vértice destino

P_0	(1,3) (1,5) (2,1) (2,6) (3,2) (3,4) (4,1) (4,6)
P_1	(5,4) (5,7) (6,3) (6,5) (7,2) (7,9) (7,13) (8,7)
P_2	(8,13) (9,8) (9,10) (9,12) (10,8) (11,9) (12,13) (13,9)
P_3	(13,11) (13,14) (14,7) (14,15) (15,16) (16,14)

Tabela 4.3: Arestas ordenadas pelo vértice origem

A Tabela 4.2 mostra o conteúdo do vetor `edge`, em cada processador P_i, após a ordenação das arestas pelo vértice destino. A Tabela 4.3 mostra o conteúdo do vetor `succ`, em cada processador P_i, após a ordenação das arestas pelo vértice origem. A Tabela 4.4 mostra o conteúdo do vetor `cycrep`, em cada processador P_i, indicando o ciclo da partição de Euler ao qual

pertence cada aresta do grafo. O algoritmo encontrou uma partição de Euler em G composta por seis ciclos $C_1 = \{(2,1), (1,3), (3,2)\}$, $C_2 = \{(1,5), (5,4), (4,6), (6,5), (5,7), (7,2), (2,6), (6,3), (3,4), (4,1)\}$, $C_3 = \{(7,9), (9,8), (8,7)\}$, $C_4 = \{(9,10), (10,8), (8,13), (13,11), (11,9)\}$, $C_5 = \{(9,12), (12,13), (13,14), (14,7), (7,13), (13,9)\}$ e $C_6 = \{(14,15), (15,16), (16,14)\}$. A Figura 4.2 mostra estes ciclos representando as arestas de cada ciclo com um tipo diferente de linha. Tabela 4.5 mostra o valor das variáveis `cyc` e `ncyc` em cada processador P_i. A variável `ncyc` indica o número de ciclos da partição de Euler.

P_0	(1,3) (1,5) (1,3) (1,5) (1,3) (1,5) (1,5) (1,5)
P_1	(1,5) (1,5) (1,5) (1,5) (1,5) (7,9) (7,13) (7,9)
P_2	(8,13) (7,9) (8,13) (7,13) (8,13) (8,13) (7,13) (7,13)
P_3	(8,13) (7,13) (7,13) (14,15) (14,15) (14,15)

Tabela 4.4: Representante do ciclo ao qual pertence cada aresta

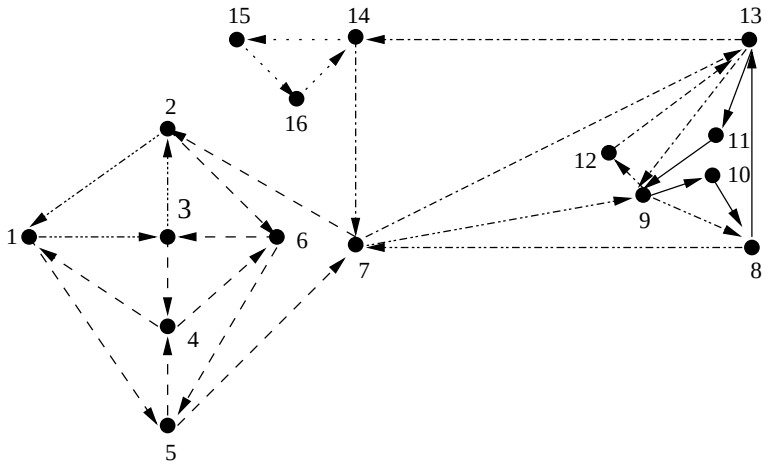


Figura 4.2: Partição de Euler de G

P_0	cyc = 1 ncyc = 6
P_1	cyc = 1 ncyc = 6
P_2	cyc = 2 ncyc = 6
P_3	cyc = 2 ncyc = 6

Tabela 4.5: Número de ciclos da partição de Euler

Passo 2 Construa o grafo bipartido

(2.1) Construa o grafo auxiliar bipartido $H = (V', C', E')$

Cada processador P_i possui um vetor `bipart` para armazenar o grafo bipartido. Para construir o grafo bipartido cada processador P_i irá executar o seguinte código:

para i = 0 **até** $\frac{m}{p}$ **faça**
`bipart(i).u := edge(i).v`
`bipart(i).v := cycrep(i)`

(2.2) Elimine arestas replicadas

(2.2.1) A replicação de algumas arestas do grafo bipartido surgem quando, por um mesmo vértice, passam duas arestas que pertencem ao mesmo circuito. Quando as réplicas aparecem dentro do processador eliminá-las é trivial.

(2.2.2) Quando as réplicas estão espalhadas entre os processadores, basta que cada processador P_i envie todas as arestas cujo vértice origem é o último vértice origem presente em suas arestas para o processador P_{i+1} (pois as arestas estão ordenadas). Então cada processador P_i verifica se as arestas recebidas também pertencem ao seu conjunto de arestas, do grafo bipartido, e elimina as possíveis réplicas. Ao final deste passo obtemos o grafo bipartido H.

(2.3) Elimine as arestas que incidem em vértices com grau menor que 2

(2.3.1) Cada processador P_i possui um vetor `vertice` que armazena os $\frac{n}{p}$ vértices atribuídos a ele. Cada processador P_i calcula para cada vértice que aparece em suas arestas o seu grau e envia para o processador P_j que armazena o vértice.

(2.2.2) O processador P_j recebe o grau parcial de seus vértices, em cada um dos processadores, calcula o grau global e o envia de volta para os processadores P_i.

(2.2.3) Cada processador P_i recebe o grau global dos vértices que aparecem em suas arestas e elimina as arestas que incidem em vértices com grau menor que 2.

A Tabela 4.6 mostra o conteúdo do vetor `bipart`, em cada processador P_i, após eliminadas as arestas replicadas. A Tabela 4.7 mostra o conteúdo do vetor `bipart`, em cada processador P_i, após a eliminação das arestas que incidem em vértices com grau menor que 2. A Figura 4.3 mostra o grafo bipartido H obtido para o grafo G.

P_0	(1,(1,3)) (1,(1,5)) (2,(1,3)) (2,(1,5)) (3,(1,3)) (3,(1,5)) (4,(1,5))
P_1	(5,(1,5)) (6,(1,5)) (7,(1,5)) (7,(7,9)) (7,(7,13)) (8,(7,9))
P_2	(8,(8,13)) (9,(7,9)) (9,(8,13)) (9,(7,13)) (10,(8,13)) (11,(8,13)) (12,(7,13)) (13,(7,13))
P_3	(13,(8,13)) (14,(7,13)) (14,(14,15)) (15,(14,15)) (16,(14,15))

Tabela 4.6: Grafo auxiliar sem arestas replicadas

P_0	(1,(1,3)) (1,(1,5)) (2,(1,3)) (2,(1,5)) (3,(1,3)) (3,(1,5))
P_1	(7,(1,5)) (7,(7,9)) (7,(7,13)) (8,(7,9))
P_2	(8,(8,13)) (9,(7,9)) (9,(7,13)) (9,(8,13)) (13,(7,13))
P_3	(13,(8,13)) (14,(7,13)) (14,(14,15))

Tabela 4.7: Grafo auxiliar sem arestas que incidem em vértices com grau menor que 2

Passo 3 Construa uma Árvore Geradora

(3.1) Construa uma árvore geradora

Obtenha uma árvore geradora do grafo bipartido através do algoritmo BSP/CGM Árvore Geradora. Ao final do algoritmo todas as arestas que pertencem a árvore geradora terão o campo `st` igual a 1.

(3.2) Compute o esteio da árvore geradora.

(3.2.1) Cada processador P_i soma quantas vezes cada vértice aparece nas suas arestas da árvore geradora e envia o resultado para o proces-

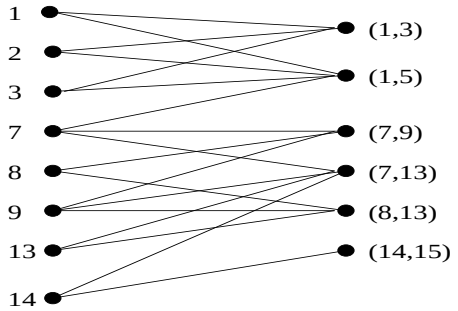


Figura 4.3: Grafo bipartido H obtido para o grafo G de entrada

sador P_j que armazena o vértice. Cada processador P_j recebe dos demais processadores esta quantidade e as acumula na variável `nst` associada a cada vértice.

(3.2.2) Em seguida, cada processador P_j envia para os demais processadores o valor acumulado de `nst`, para cada um de seus vértices. Quando termina a rodada de comunicação cada P_i remove da árvore geradora os vértices com grau menor que 2, obtendo o esteio.

(3.3) Calcule o número de arestas do esteio

Depois de remover os vértices, cada processador P_i deve calcular o número de arestas restantes e enviar para os demais processadores. Cada processador P_i acumula as somas parciais e obtém o número total de arestas do esteio.

A Tabela 4.8 mostra o conteúdo do vetor `bipart`, em cada processador P_i, após a eliminação das arestas que não pertencem à árvore geradora. A Figura 4.4 mostra a árvore geradora obtida para o grafo do exemplo.

P_0	(1,(1,3)) (1,(1,5)) (2,(1,5)) (3,(1,5))
P_1	(7,(1,5)) (7,(7,9)) (7,(7,13)) (8,(7,9))
P_2	(9,(7,9)) (9,(8,13)) (13,(7,13))
P_3	(14,(7,13)) (14,(14,15))

Tabela 4.8: Árvore geradora obtida para o grafo G

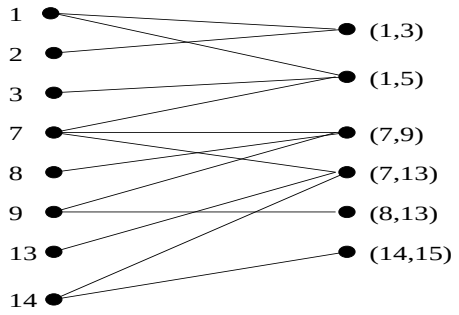


Figura 4.4: Árvore geradora para o grafo G de entrada

A Tabela 4.9 mostra o conteúdo do vetor `bipart`, em cada processador P_i, após a eliminação das arestas da árvore geradora que incidem em vértices com grau menor que 2. A Figura 4.5 mostra o esteio obtido para o grafo G.

P_0	(1,(1,3)) (1,(1,5))
P_1	(7,(1,5)) (7,(7,9)) (7,(7,13))
P_2	(9,(7,9)) (9,(8,13))
P_3	(14,(7,13)) (14,(14,15))

Tabela 4.9: Esteio obtido para o grafo G

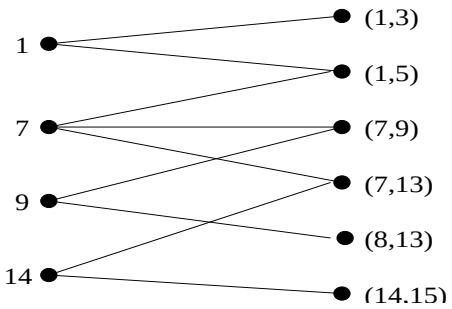


Figura 4.5: Esteio para o grafo G de entrada

Passo 4 Realize a Costura

(4.1) **Se** o número de arestas a serem costuradas for menor ou igual a $O(\frac{m}{p})$, os processadores devem enviar suas arestas para um único processador, por exemplo o processador raiz, que realiza a operação de costura seqüencialmente. A operação de costura consiste em trocar os sucessores das arestas que pertencem ao esteio, da seguinte forma:

para cada aresta do esteio **faça**

```
succ(esteio(i)) := succ(esteio(i+1 mod k))
```

onde k é o número de arestas do esteio que chegam no vértice v, para todo v que deve ser costurado.

(4.2) **Senão** a operação de costura deverá ser realizada de maneira distribuída.

(4.2.1) Cada processador P_i deve ordenar as arestas do esteio pelo vértice $v \in V^*$ e calcular o início e o fim de cada sublista. Uma sublista l é formada pelas arestas do esteio que incidem em um mesmo vértice $v_l \in V^*$. Cada processador P_i envia o início e o fim de cada sub-lista l para o processador P_j que armazena o vértice v_l.

(4.2.2) Cada processador P_j recebe o início e o fim das sublistas de seus

vértices dos demais processadores e calcula, para cada sublista l, em qual o processador a sublista l começa e em qual processador a sublista l termina.

(4.2.3) P_j calcula também para cada processador P_i, que armazena partes de uma sublista l, qual é o seu vizinho esquerdo P_x e o seu vizinho direito P_y, onde, x é o maior número menor que l, tal que, P_x armazena parte da sublista l e y é o menor número maior que l tal que P_y armazena parte da sublista l, se houver.

(4.2.4) P_j envia para cada P_i a identificação dos seus vizinhos esquerdo x e direito y, para cada sublista l que P_i armazena.

(4.2.5) Cada processador P_i recebe o início e o fim de cada uma de suas sublistas. **Se** uma sublista l começa e termina em P_i **então** basta que se faça uma costura local desta sublista.

(4.2.6) **Senão** existem partes da sublista l distribuídas entre os processadores. Então, P_i envia o sucessor da primeira aresta da sublista l para o seu vizinho esquerdo e recebe o sucessor da última aresta da sublista l do seu vizinho direito e realiza a costura global.

— Fim do Algoritmo —

Para o grafo do exemplo a operação de costura deve ser distribuída, pois as arestas a serem costuradas não cabem em um único processador. A Tabela 4.10 mostra o início e o fim (posição no vetor) de cada sublista, dos vértices que devem ser costurados, em cada processador P_i. A Tabela 4.11 mostra os vizinhos esquerdo e direito (processadores) de cada sub-lista, dos vértices que devem ser costurados, em cada processador P_i.

P_0	vértice: 1 inicio: 0 fim: 1
P_1	vértice: 7 inicio: 0 fim: 2
P_2	vértice: 9 inicio: 0 fim: 1
P_3	vértice: 14 inicio: 0 fim: 1

Tabela 4.10: Início e fim de cada sublista

Como todas as sublistas do nosso exemplo começam e terminam dentro de um único processador, a costura pode ser feita localmente, ou seja, a sublista do vértice 1 será costurada localmente no processador 0; a sublista do vértice 7 será costurada localmente no processador 1 e assim sucessivamente. A

P_0	vértice: 1 esquerdo: 0 direito: 0
P_1	vértice: 7 esquerdo: 1 direito: 1
P_2	vértice: 9 esquerdo: 2 direito: 2
P_3	vértice: 14 esquerdo: 3 direito: 3

Tabela 4.11: Vizinhos esquerdo e direito de cada sublista

P_0	(2,1) (1,3) ⇒ (2,1) (1,5) (4,1) (1,5) ⇒ (4,1) (1,3)
P_1	(5,7) (7,2) ⇒ (5,7) (7,9) (8,7) (7,9) ⇒ (8,7) (7,13) (14,7) (7,13) ⇒ (14,13) (7,2)
P_2	(7,9) (9,8) ⇒ (7,9) (9,10) (11,9) (9,10) ⇒ (11,9) (9,8)
P_3	(13,14) (14,7) ⇒ (13,14) (14,15) (16,14) (14,15) ⇒ (16,14) (14,7)

Tabela 4.12: Operação de costura sobre cada sublista

Tabela 4.12 mostra as arestas que serão costuradas, em cada processador P_i, e seus respectivos sucessores antes e depois da operação de costura.

A Tabela 4.13 mostra o vetores `edge` e `succ`, depois da operação de costura, em cada processador P_i. Juntos, `edge` e `succ`, definem um circuito de Euler encontrado pelo algoritmo para o grafo G.

P_0	(2,1) (4,1) (3,2) (7,2) (1,3) (6,3) (3,4) (5,4) (1,5) (1,3) (2,1) (2,6) (3,2) (3,4) (4,1) (4,6)
P_1	(1,5) (6,5) (2,6) (4,6) (5,7) (8,7) (14,7) (9,8) (5,4) (5,7) (6,3) (6,5) (7,9) (7,13) (7,2) (8,7)
P_2	(10,8) (7,9) (11,9) (13,9) (9,10) (13,11) (9,12) (7,13) (8,13) (9,10) (9,8) (9,12) (10,8) (11,9) (12,13) (13,9)
P_3	(8,13) (12,13) (13,14) (16,14) (14,15) (15,16) (13,11) (13,14) (14,15) (14,7) (15,16) (16,14)

Tabela 4.13: Circuito de Euler de G

A Figura 4.6 mostra um exemplo onde existem partes de uma mesma sublista distribuídas entre os processadores (Subpasso 4.2.6). Neste exemplo, os processadores P_0, P_1 e P_3 possuem partes da sublista 1. Cada P_i envia

o sucessor da sua primeira aresta da sublista 1 para o seu vizinho esquerdo, e recebe o sucessor da sua última aresta da sublista 1 do seu vizinho direito, realizando a costura global da sublista 1.

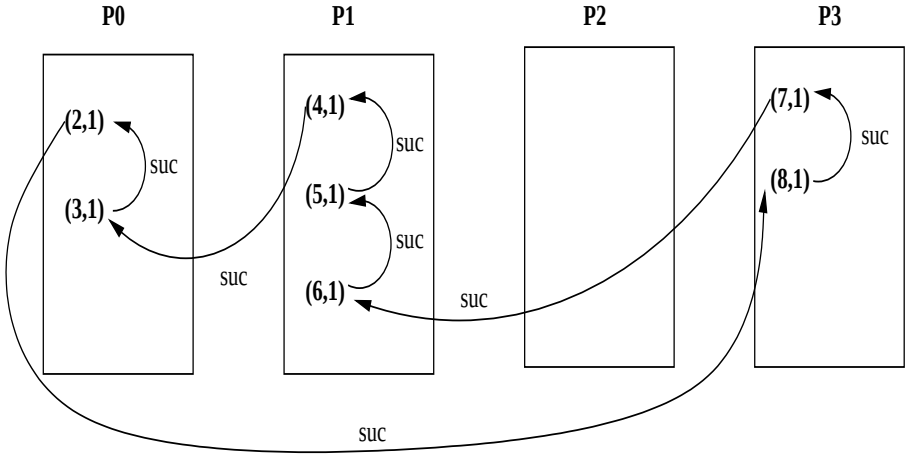


Figura 4.6: Costura distribuída

4.5 Complexidade e Corretude

Vamos agora analisar a complexidade e a corretude do algoritmo.

Lema 7 *O algoritmo BSP/CGM para computar o circuito de Euler de um grafo euleriano G = (V, E) utiliza O(*$\log p$*) rodadas de comunicação e* $O(\frac{m+n}{p})$ *computação local.*

Demonstração: No Passo 1 obtemos uma partição de Euler do grafo. Os subpassos 1.1 e 1.2 efetuam duas ordenações inteiras em um conjunto com m elementos. Utilizando um algoritmo BSP/CGM de ordenação inteira [Fer99, CD99], esses subpassos podem ser executados em O(1) rodadas de comunicação e computação local $O(\frac{m+n}{p})$.

No Subpasso 1.3 utilizamos um algoritmo de duplicação recursiva. Utilizando o algoritmo proposto por [CDF+97], esse subpasso pode ser efetuado em $O(\log p)$ rodadas de comunicação e computação local $O(\frac{m}{p})$.

O Subpasso 1.4 pode ser facilmente implementado com a utilização de um algoritmo de soma de prefixos que utiliza $O(1)$ rodadas de comunicação e $O(\frac{m}{p})$ computação local.

No Passo 2 construímos o grafo bipartido auxiliar. Para construir o grafo e eliminar as arestas replicadas dentro de cada processador (subpassos 2.1 e 2.2.1) utiliza $O(\frac{m}{p})$ computação local. Depois o algoritmo utiliza uma rodada de comunicação e $O(\frac{m}{p})$ computação local para eliminar as réplicas presentes nos outros processadores (Subpasso 2.2.2). Este passo ainda utiliza $O(\frac{m}{p})$ computação local e 2 rodadas de comunicação para eliminar as arestas do grafo com grau < 2 (Subpasso 2.3).

No Subpasso 3.1, construímos uma árvore geradora do grafo através do algoritmo BSP/CGM para árvore geradora proposto por [CDF+97] que utiliza $O(\log p)$ rodadas de comunicação e $O(\frac{m+n}{p})$ computação local por rodada. Depois de encontrar a árvore geradora o algoritmo elimina as arestas que possuem grau < 2 (Subpasso 3.2) utilizando $O(\frac{m}{p})$ computação local e 1 rodada de computação. O Subpasso 3.3 utiliza 1 rodada de comunicação para calcular o número de arestas a serem costuradas.

O Passo 4 realiza a operação de costura. Se a costura seqüencial for realizada (Subpasso 4.1) o algoritmo precisa de $O(\frac{m}{p})$ computação local. Se a costura distribuída for realizada (Subpasso 4.2) o algoritmo utiliza 3 rodadas de comunicação e $O(\frac{m}{p})$ computação local por rodada. □

Teorema 12 *O algoritmo 5 computa corretamente o circuito de Euler de um grafo euleriano G = (V, E).*

Demonstração: O Passo 1 computa corretamente uma partição de Euler para o grafo $G = (V, E)$ [GR88, Fer99]. O Passo 2 computa corretamente o grafo auxiliar bipartido H [GR88, CDF+97].

É fácil verificar que utilizamos o algoritmo BSP/CGM para computar a árvore geradora [CDF+97] e algoritmos BSP/CGM básicos de soma e soma de prefixos, além de computação local, o esteio é computado corretamente. O Passo 4 computa corretamente a costura com a qual obtemos o circuito de Euler de G [GR88, CDSS92]. □

4.6 Conclusão

Neste capítulo, apresentamos algoritmos que implementam algumas técnicas básicas em BSP/CGM, tais como, ordenação de inteiros, duplicação recursiva, circuitos de Euler em árvores, componentes conexos e árvore geradora.

Para a ordenação, descrevemos um algoritmo BSP/CGM denominado *Split Sort* [GV94, Fer99, CMS01]. Este algoritmo ordena n elementos, utili-

zando no máximo $O(1)$ rodadas de comunicação e a complexidade da computação local é $O(\frac{n \log n}{p})$. No caso de números inteiros, os procedimentos internos podem usar um algoritmo seqüencial de ordenação de custo linear e a complexidade local será $O(\frac{n}{p})$.

Para a duplicação recursiva, descrevemos o algoritmo BSP/CGM apresentado por Cáceres *et al* [CDF+97]. O algoritmo utiliza $O(\log p)$ rodadas de comunicação $O(\frac{n}{p})$ tempo de computação local por rodada.

O algoritmo de Cáceres *et al* [CDF+97] para calcular a árvore geradora de um grafo também foi descrito. O algoritmo utiliza $O(\log p)$ rodadas de comunicação e $O(\frac{m+n}{p})$ computação local por rodada.

O objetivo principal deste capítulo é a descrição de um algoritmo inédito no modelo BSP/CGM para computar um circuito de Euler em um grafo orientado euleriano. O algoritmo utiliza como subrotinas as técnicas básicas descritas e utiliza $O(\log p)$ rodadas de comunicação e $O(\frac{m+n}{p})$ computação local.

Capítulo 5

Implementação

5.1 Introdução

Neste capítulo, tratamos da implementação do algoritmo BSP/CGM para computação de circuitos de Euler que desenvolvemos e descrevemos no capítulo anterior. A implementação do algoritmo, denominada EulerTour, utilizou a biblioteca MPI e a linguagem C. Para gerar os grafos que serviram de entrada para o programa EulerTour, implementamos um gerador de grafos eulerianos. O código fonte do programa EulerTour e do gerador estão no Apêndice B.

5.2 Detalhes da Implementação

O algoritmo descrito no Capítulo 4 foi implementado utilizando a linguagem C e a biblioteca para troca de mensagens MPI. O MPI foi escolhido por diversos fatores, o primeiro deles é por se tratar de um padrão na programação por troca de mensagens, o que garante a portabilidade do programa. Outro fator importante são as rotinas de comunicação oferecidas pelo MPI, principalmente as rotinas de comunicação coletiva, que permitem que vários processos troquem informações através de uma única chamada.

O programa utiliza o formato SPMD onde um processo especial, denominado processo raiz, lê as m arestas do grafo de um arquivo e distribui $O(\frac{m}{p})$ arestas para cada um dos p processadores da aplicação. Cada um dos p processos executa o Algoritmo 5 (Seção 4.2 do Capítulo 4) sobre as suas $O(\frac{m}{p})$ arestas. O circuito de Euler é dado pelos vetores `edge` e `succ`, que estão ar-

mazenados nos p processadores, $\frac{m}{p}$ arestas por processador. De forma inversa ao passo inicial, cada processador p_i encaminha ao final do programa o seu conjunto de arestas, `edge` e `succ`, ao processador raiz que os armazena em um arquivo de saída.

Como vimos no Capítulo 4, [Fer99, LG00] implementaram os problemas do *list ranking*, ordenação e componentes conexos. Todos os resultados obtidos nas implementações confirmam os resultados teóricos dos algoritmos BSP/CGM para esses problemas. Em função disso, na nossa implementação, preocupamo-nos exclusivamente com os resultados relativos à computação do circuito de Euler. As rotinas BSP/CGM de ordenação, duplicação recursiva e árvore geradora, vistas no capítulo anterior, não foram implementadas. O programa utiliza algoritmos seqüenciais para essas rotinas, pois, como observamos, não se trata do objetivo do nosso estudo.

Nesta implementação, preocupamo-nos em medir o tempo de execução do algoritmo. As medidas obtidas não abrangem os tempos gastos com a distribuição inicial dos dados e o envio dos resultados ao final do programa. Também desconsideramos os tempos gastos para executar os algoritmos seqüenciais de ordenação, duplicação recursiva e árvore geradora, considerando apenas os tempos gastos com o algoritmo para obtenção do circuito de Euler do grafo.

Uma das principais dificuldades encontradas na implementação foi a definição de uma estrutura de dados que permita manter todas as informações necessárias para se recuperar qualquer informação, de qualquer processador, a partir de qualquer outro processador. Por exemplo, suponha que um processador precisa de informações sobre uma aresta que está localizada em outro processador. É importante que ele consiga essas informações sem precisar consultar todos os processadores. Outra dificuldade que encontramos foi o gerenciamento das informações contidas nas estruturas de dados, pois a cada passo do algoritmo novas estruturas precisam ser criadas ou modificadas.

Outra preocupação durante a implementação foi com a escolha das rotinas de troca de mensagens utilizadas. Optamos pelas rotinas de comunicação coletivas oferecidas pelo MPI, pois a sua utilização produz um código mais claro e melhor estruturado. Como todos os processos executam a mesma chamada à uma rotina de comunicação coletiva a distinção entre as rodadas de comunicação e computação se torna mais evidente.

As rodadas de comunicação também mereceram uma atenção especial pois como cada nova estrutura, por exemplo a árvore geradora, estava distribuída entre os processadores, sempre que surgia a necessidade de uma informação sobre a estrutura como um todo era preciso inserir novas roda-

das de comunicação. Junto com as rodadas de comunicação havia toda uma computação local para se obter as informações desejadas.

Sempre que uma rotina seqüencial era utilizada, por exemplo a duplicação recursiva, havia a necessidade de enviar todas a informações, relevantes para esta rotina, de todos os processadores para o processador raiz que executava a rotina seqüencialmente e devolvia aos processadores as novas informações. Além disso, o modelo BSP/CGM define uma complexidade para o tamanho da memória local de cada processador e para o tamanho das mensagens trocadas entre os processadores que deve ser respeitada pelo programa.

5.3 Grafos de entrada

Os grafos de entrada usados em nossos testes foram gerados de forma aleatória. Todos os grafos gerados são eulerianos. O programa utilizado gera a matriz de adjacências do grafo, *mat_adj*, decidindo com probabilidade p se uma determinada aresta está no grafo gerado. Denotamos tais grafos da forma $G_{n,p}$, onde n é o número de vértices do grafo e p é a probabilidade de uma aresta estar no grafo. Por exemplo, $G_{128,20}$ é um grafo com 128 vértices e cada aresta é sorteada com probabilidade de 20%.

O programa gera os grafos da seguinte maneira: dados dois vértices, i e j, se existe uma aresta entre eles $mat\_adj(i,j) = 1$. Caso contrário, $mat\_adj(i,j) = 0$. Para garantir que o grafo gerado é euleriano, o programa verifica se a quantidade de 1 na linha i é igual a quantidade de 1 na coluna i, $0 \leq i \leq n$. Se as quantidades forem diferentes o programa tenta inserir novas arestas ou remover arestas existentes aleatoriamente até que o grafo se torne euleriano.

Depois de gerada a matriz de adjacências do grafo, o programa armazena em um arquivo o número de vértices, o número de arestas e a lista das arestas do grafo gerado.

5.4 Características das máquinas

A experiência foi realizada no *Beowulf* do IC-UNICAMP, que possui 66 processadores *Pentium*, cada um com uma memória RAM de 256 Mbytes e 256 Mbytes para *swapping*. A comunicação entre os processadores é realizada através de uma rede de interconexão de 100 Mbits.

Um *Beowulf* é um conjunto de PCs (*Personal Computers*) interligados

por tecnologias de redes comuns e utilizando um sistema operacional UNIX. Basicamente consiste na utilização de PCs interligados de modo a constituir uma máquina virtual com capacidade de processamento paralelo. Mais informações sobre o *Beowulf* podem ser obtidas em:

- http://www.beowulf.org
- http://cesdis.gsfc.nasa.gov/linux/beowulf

5.5 Resultados dos Testes Realizados

Para obtermos os resultados mostrados nesta seção foram gerados grafos eulerianos com diferentes números de vértices, sendo que cada aresta é sorteada com probabilidade de 20%. Aqui, consideramos o tempo total de execução do algoritmo, ou seja, o tempo gasto com computação local e comunicação entre os processadores.

5.5.1 Grafos com 128 vértices

O primeiro teste realizado considerou grafos com 128 vértices. Foram gerados cinco grafos diferentes G_1, G_2, G_3, G_4 e G_5. A Tabela 5.1 mostra o número de arestas e o número de circuitos da partição de Euler de cada grafo G_i. O programa foi executado cinco vezes para cada grafo G_i e vamos considerar o menor tempo obtido nas cinco execuções (Tabela 5.2).

Grafo	arestas	circuitos
G_1	4978	30
G_2	5085	42
G_3	4961	27
G_4	5053	39
G_5	4939	37

Tabela 5.1: Número de arestas e circuitos da partição de Euler para grafos com 128 vértices

Na Figura 5.1, podemos observar que o tempo de execução do algoritmo quando aumentamos o número de processadores. Como o número de arestas é pequeno, a execução com um único processador é mais rápido. Quando

Processadores	G_1	G_2	G_3	G_4	G_5
1	0.075661	0.044399	0.062946	0.127695	0.098185
2	0.128496	0.096378	0.115442	0.179133	0.150002
4	0.124412	0.096858	0.113885	0.164554	0.143794
8	0.085460	0.081435	0.083581	0.094586	0.092203
16	0.083321	0.083291	0.083155	0.088332	0.086063
32	0.094561	0.093976	0.100651	0.098790	0.101835
64	0.302106	0.302597	0.301502	0.306478	0.305635

Tabela 5.2: Tempo de execução do algoritmo para grafos com 128 vértices

aumentamos de 2 para 4 processadores conseguimos uma melhora, e a partir de 16 processadores a comunicação acaba sendo maior que a computação local e os tempos de execução aumentam. Esta instância mostra que é difícil analisarmos o comportamento do algoritmo quando temos um número muito pequeno de arestas.

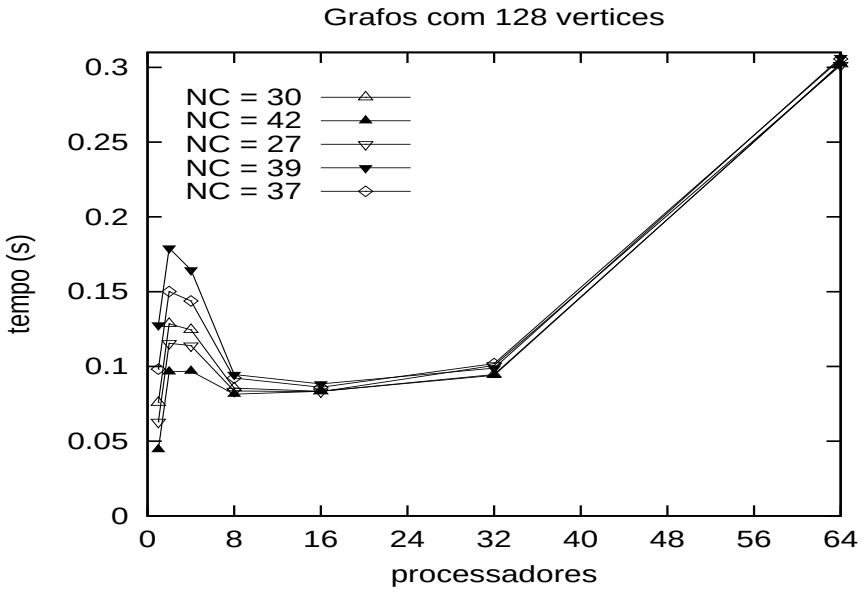


Figura 5.1: Gráfico para grafos com 128 vértices

5.5.2 Grafos com 256 vértices

No segundo teste também foram gerados cinco grafos diferentes, G_1, G_2, G_3, G_4 e G_5, cada um com 256 vértices. A Tabela 5.3 mostra o número de arestas e o número de circuitos da partição de Euler de cada grafo G_i. Para esta instância o programa também foi executado cinco vezes para cada grafo G_i e consideramos o menor tempo obtido nas cinco execuções. Os tempos obtidos estão na Tabela 5.4.

Grafo	arestas	circuitos
G_1	18356	52
G_2	18527	51
G_3	18317	52
G_4	18756	57
G_5	18372	49

Tabela 5.3: Número de arestas e circuitos da partição de Euler para grafos com 256 vértices

Processadores	G_1	G_2	G_3	G_4	G_5
1	0.590553	0.623308	0.443335	0.496561	0.509064
2	0.166752	0.179329	0.132080	0.141056	0.149729
4	0.066882	0.072514	0.062855	0.064652	0.065941
8	0.085361	0.086033	0.089652	0.087995	0.085465
16	0.086299	0.089275	0.087007	0.088913	0.092955
32	0.102530	0.128476	0.118065	0.126849	0.113098
64	0.370396	0.303024	0.304385	0.307984	0.304233

Tabela 5.4: Tempo de execução do algoritmo para grafos com 256 vértices

Nesta instância, o número de arestas aumenta significativamente, de aproximadamente 5000 da instância anterior para quase 19000, com isso temos um aumento da computação local e conseqüentemente com a utilização de 2 processadores já obtemos uma melhora no tempo de execução do algoritmo, como mostra a Figura 5.2. O *speedup* super-linear pode ser em função da memória cache, pois com esses valores não é feito *swap*. Novamente, a partir de um certo número de processadores o tempo gasto com a comunicação determina o tempo de execução total do algoritmo.

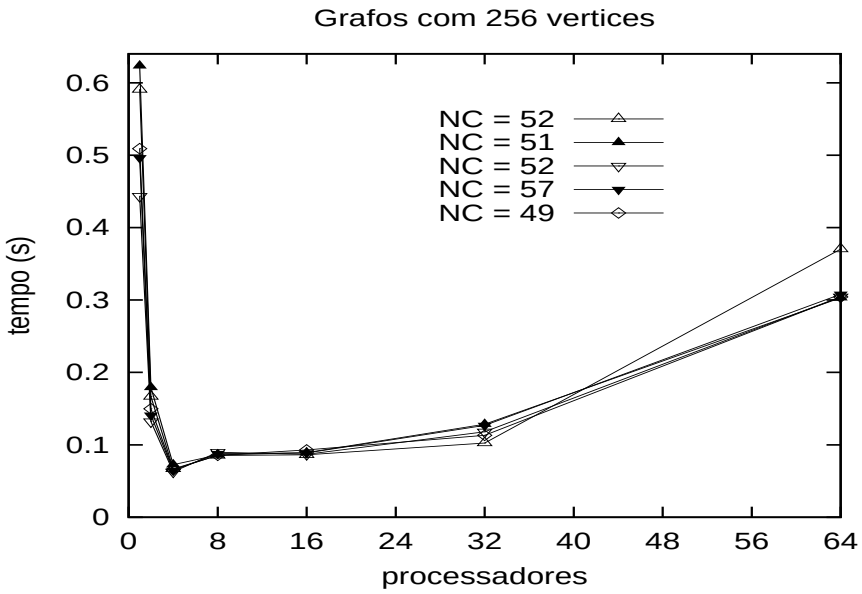


Figura 5.2: Gráfico para grafos com 256 vértices

5.5.3 Grafos com 384 vértices

O terceiro teste foi realizado com grafos de 384 vértices. Também foram gerados cinco grafos diferentes G_1, G_2, G_3, G_4 e G_5. Na Tabela 5.5 está o número de arestas e o número de circuitos da partição de Euler de cada grafo G_i. Para cada grafo G_i o programa foi executado cinco vezes e novamente consideramos o menor tempo obtido nas cinco execuções. Os tempos obtidos estão na Tabela 5.6.

Grafo	arestas	circuitos
G_1	40373	79
G_2	40498	69
G_3	40522	67
G_4	40976	62
G_5	40526	74

Tabela 5.5: Número de arestas e circuitos da partição de Euler para grafos com 384 vértices

Como mostra a Figura 5.3, para esta instância os tempos do algoritmo paralelo decrescem a medida que aumentamos o número de processadores,

Processadores	G_1	G_2	G_3	G_4	G_5
1	1.767295	2.721215	2.889279	2.774800	1.384056
2	0.599985	0.893345	0.912682	0.890731	0.458646
4	0.208586	0.284340	0.306130	0.274068	0.172301
8	0.157393	0.168773	0.173824	0.167250	0.151559
16	0.143216	0.150685	0.151509	0.150423	0.142476
32	0.151521	0.157534	0.163105	0.165140	0.149100
64	0.325365	0.328582	0.326214	0.339717	0.320689

Tabela 5.6: Tempo de execução do algoritmo para grafos com 384 vértices

sendo que até 16 processadores ainda há um ganho, depois disso o aumento no número de processadores não melhora o desempenho do algoritmo. Os grafos possuem aproximadamente 40000 arestas e quase 80 circuitos, assim, observamos que para grafos maiores o desempenho do algoritmo tende a ser melhor. Como temos um número grande de arestas, o tempo gasto com a computação local determina o tempo de execução total do algoritmo.

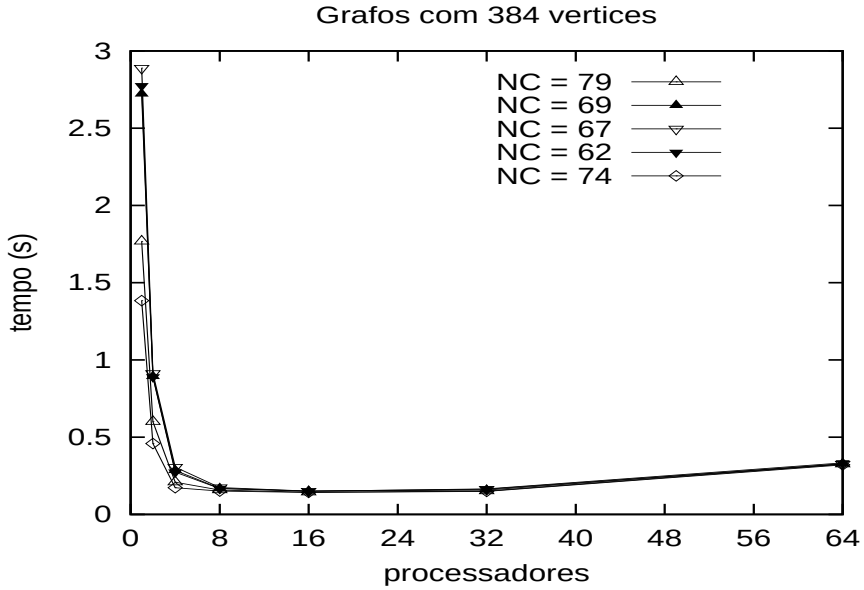


Figura 5.3: Gráfico para grafos com 256 vértices

5.5.4 Grafo com 512 vértices

O quarto teste foi realizado com grafos de 512 vértices. Também foram gerados cinco grafos diferentes G_1, G_2, G_3, G_4 e G_5. A Tabela 5.7 mostra o número de arestas e o número de circuitos da partição de Euler de cada grafo G_i. Novamente o programa foi executado cinco vezes para cada grafo G_i e consideramos o menor tempo obtido nas cinco execuções. A Tabela 5.8 mostra os tempos obtidos.

Grafo	arestas	circuitos
G_1	71654	89
G_2	71271	79
G_3	71523	80
G_4	71653	89
G_5	72485	102

Tabela 5.7: Número de arestas e circuitos da partição de Euler para grafos com 512 vértices

Processadores	G_1	G_2	G_3	G_4	G_5
1	4.117713	5.124322	4.986945	4.934334	7.631034
2	1.261497	1.681731	1.530067	1.454775	2.088567
4	0.393307	0.525984	0.484048	0.471253	0.676139
8	0.244051	0.267215	0.262339	0.264898	0.309131
16	0.218123	0.220326	0.221822	0.224551	0.231296
32	0.233427	0.229774	0.229047	0.228280	0.232303
64	0.397516	0.407809	0.388348	0.387437	0.393453

Tabela 5.8: Tempo de execução do algoritmo para grafos com 512 vértices

Para esta instância, a Figura 5.4 mostra que tempos do algoritmo paralelo também decrescem a medida que aumentamos o número de processadores. Assim como na instância anterior, depois de um determinado número de processadores a adição de novos processadores não melhora o desempenho do algoritmo. Mas observamos que, para 64 processadores, o desempenho foi melhor do que para a instância anterior.

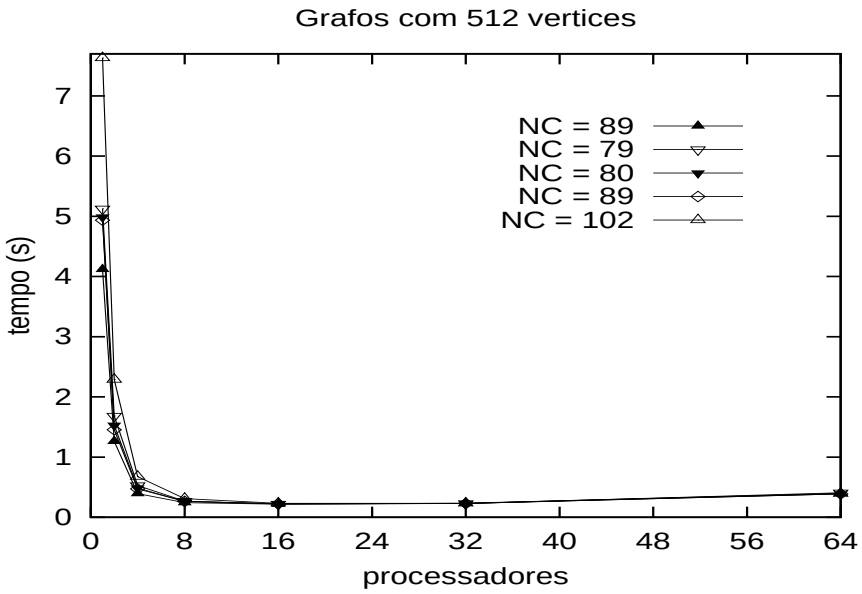


Figura 5.4: Gráfico para grafos com 512 vértices

5.5.5 Grafo com 1024 vértices

Para o quinto teste geramos um grafo com 1024 vértices e 279058 arestas. A Tabela 5.9 mostra os tempos obtidos.

Processadores	Tempo (s)
1	55.687255
2	10.895935
4	3.210515
8	2.060558
16	0.763391
32	0.659695
64	0.623849

Tabela 5.9: Tempo de execução do algoritmo para grafo com 1024 vértices e 279058 arestas

Para esta instância, os tempos do algoritmo paralelo também decrescem à medida que aumentamos o número de processadores, como podemos observar na Figura 5.5, pelos mesmos motivos da instância anterior. Note ainda que os ganhos são bastante significativos quando passamos de 1 para 2 pro-

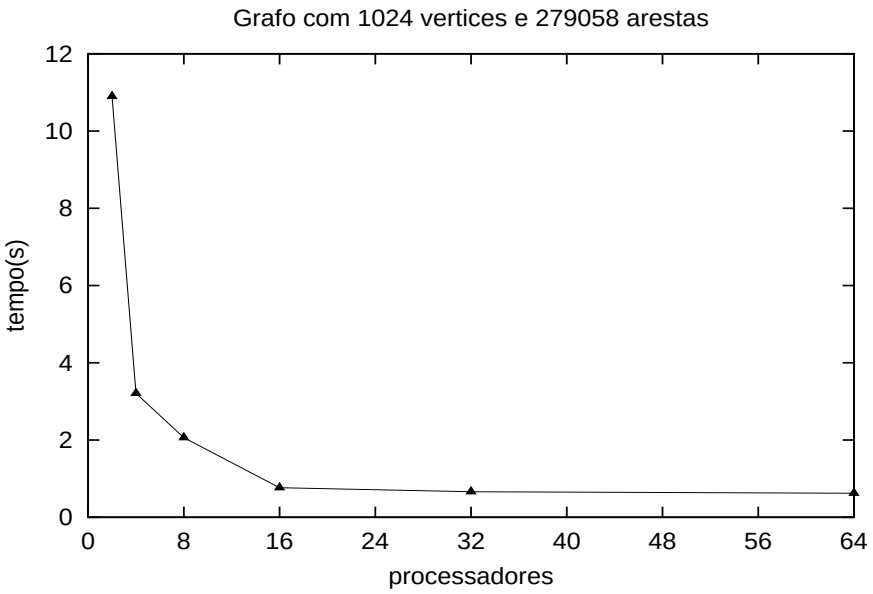


Figura 5.5: Gráfico para 1024 vértices e 279058 arestas

cessadores e, depois, de 2 para 4 processadores. A partir de 8 processadores, os ganhos passam a ser menos significativos e tendem a se uniformizar. Isto ocorre porque, embora o tempo de computação local seja dividido em pedaços menores, o número de processadores que precisam comunicar-se aumenta e a comunicação volta a influenciar no tempo total de execução, diminuindo os ganhos.

5.5.6 Grafo com 1280 vértices

Para o sexto teste consideramos um grafo com 1280 vértices. Foram geradas 434213 arestas. A Tabela 5.10 mostra os tempos medidos.

Analisando o gráfico da Figura 5.6 percebemos um comportamento análogo ao da instância anterior. Notamos ainda que a partir de 8 processadores os ganhos são proporcionalmente melhores que os da instância anterior. Isso porque ao aumentarmos o tamanho do grafo aumentamos também o impacto do tempo de processamento no tempo total de execução.

Processadores	Tempo (s)
1	186.558166
2	95.390303
4	61.482433
8	15.810464
16	5.660367
32	3.116062
64	2.49476

Tabela 5.10: Tempo de execução do algoritmo para grafo com 1280 vértices e 434213 arestas

Processadores	Tempo (s)
1	265.330792
2	147.326562
4	38.471446
8	10.775232
16	4.346431
32	2.812743
64	2.421682

Tabela 5.11: Tempo de execução do algoritmo para grafo com 1600 vértices e 673500 arestas

5.5.7 Grafo com 1600 vértices

No sétimo teste geramos um grafo com 1600 vértices e 673500 arestas. A Tabela 5.11 mostra os tempos medidos.

Na Figura 5.7 podemos observar o tempo de execução quando aumentamos o número de processadores. O gráfico mostra um ganho significativo quando passamos de 2 para 4 processadores e de 4 para 8 processadores, e que os ganhos obtidos foram proporcionalmente melhores que os da instância anterior. Como vimos anteriormente, isso ocorre porque ao aumentarmos o tamanho do grafo, aumentamos também o impacto do tempo de processamento sobre o tempo total de execução. Podemos observar que os ganhos obtidos nesta instância quando passamos de 16 para 32 processadores e de 32 para 64 processadores são proporcionalmente melhores que os ganhos obtidos na instância anterior. Neste caso, também, o ganho é devido ao aumento do tamanho do grafo.

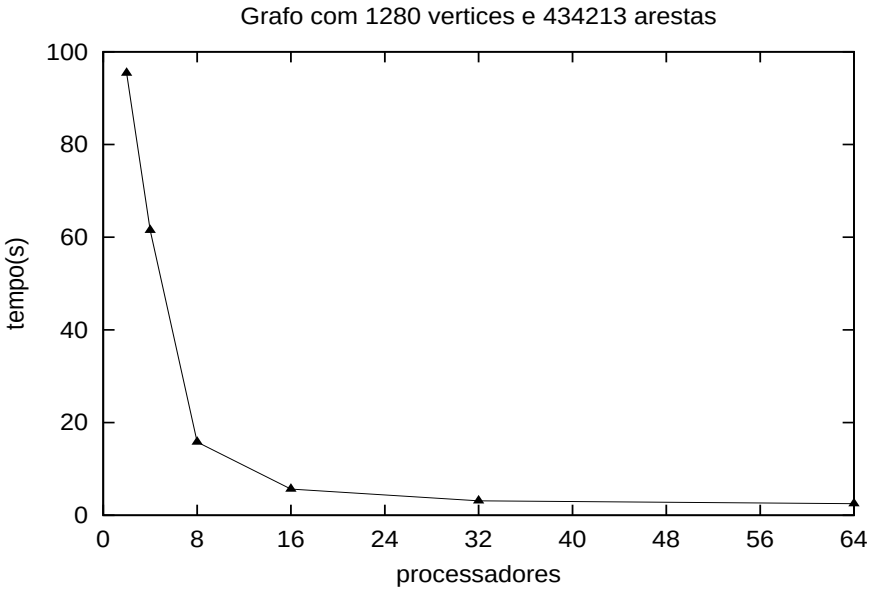


Figura 5.6: Gráfico para 1280 vértices e 434213 arestas

5.6 Conclusão

Implementamos o algoritmo BSP/CGM para computar um circuito de Euler em um grafo euleriano. O algoritmo foi implementado em MPI e executado no *Beowulf* do Instituto de Computação da UNICAMP com 66 nós. O programa foi executado utilizando grafos variando de 128 vértices e 4939 arestas a 1600 vértices e 673500 arestas. Esses grafos foram gerados aleatoriamente pelo programa descrito no Apêndice B.

Para grafos pequenos (128, 256, 384 e 512 vértices), geramos cinco instâncias de cada grafo e fizemos cinco medidas para cada caso (1, 2, 4, 8, 16, 32 e 64 processadores). Tomamos a melhor medida do processador mais lento entre as cinco 5 execuções.

Pelos resultados obtidos nesses grafos, pudemos observar que a comunicação acaba afetando o desempenho do algoritmo, sendo que o aumento de processadores, em alguns casos, acaba aumentando o tempo final de execução do algoritmo. Nos casos onde o aumento de processadores implica em redução do tempo de processamento fica difícil fazer uma estimativa real da eficiência ou do *speedup* em função da utilização da memória *cache*.

Para grafos maiores (1024, 1280 e 1600 vértices) o número de arestas chega

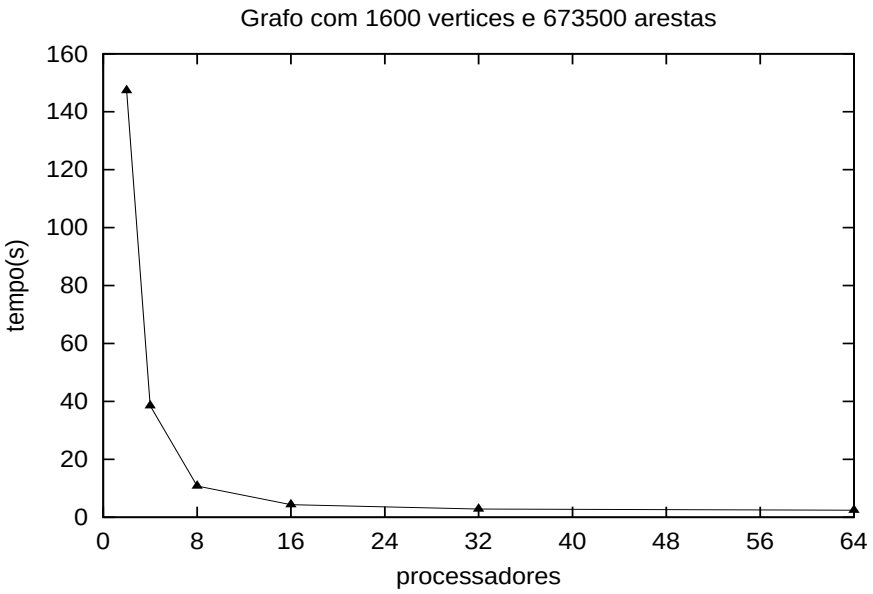


Figura 5.7: Gráfico para 1600 vértices e 673500 arestas

a quase 675000, fazendo com que o trabalho a ser executado seja maior que o tempo gasto na comunicação. Nesses casos, em função do tamanho do grafo, quando executados em um único processador o algoritmo usa "*swap*"o que acaba comprometendo a avaliação do seu tempo com relação ao executado com mais processadores. Isso explica a ocorrência *speedups* aparentemente super-lineares.

Capítulo 6

Conclusão

A obtenção de um circuito de Euler em um grafo é um problema clássico de teoria dos grafos. As diversas possibilidades de aplicação desse problema, tais como componentes conexos, emparelhamento máximo e decomposição em orelha, e suas importantes características teóricas motivaram esse estudo.

Vários algoritmos, seqüenciais e paralelos, foram propostos para resolver este problema. Em geral, os algoritmos seqüenciais utilizam como rotina a busca em profundidade que é um problema difícil de ser paralelizado. Por esta razão, os algoritmos paralelos buscam uma estrutura alternativa para a busca em profundidade.

Entre os algoritmos paralelos está o proposto por Atallah-Vishkin [GR88] e o proposto por Cáceres *et al* [CDSS92], ambos foram desenvolvidos para o modelo de computação paralela PRAM. O algoritmo de Atallah-Vishkin utiliza uma árvore geradora como alternativa para a busca em profundidade. O algoritmo de Cáceres *et al* propõe uma estrutura mais simples, denominada suporte, como alternativa.

Embora o modelo PRAM seja muito conhecido ele não é adequado para as máquinas paralelas reais, ou seja, o modelo não consegue captar algumas características importantes destas máquinas, como o alto custo da comunicação entre os processadores. Conseqüentemente, muitos algoritmos teoricamente eficientes para o modelo PRAM não produzem o desempenho esperado quando implementados em máquinas paralelas reais.

Para resolver este problema foram propostos os modelos realísticos de computação paralela, que buscam uma maior proximidade entre o desempenho teórico e prático dos algoritmos desenvolvidos. Entre os modelos realísticos estão o BSP proposto por Valiant [Val90] e o CGM proposto por

Dehne *et al* [DFRC93]. Segundo Götz [Göt98], o modelo CGM facilita o desenvolvimento de algoritmos por ser mais simples (simplifica os custos de comunicação) e levemente mais poderoso que o modelo BSP. Além disso, um algoritmo CGM pode ser transferido para o modelo BSP diretamente, sem a necessidade de mudanças.

Não encontramos na literatura algoritmos no modelo de computação paralela BSP/CGM para o problema. Baseado no algoritmo proposto por Cáceres *et al* [CDSS92], desenvolvemos um algoritmo paralelo no modelo BSP/CGM para obtenção de um circuito de Euler em grafos eulerianos, utilizando p processadores. O algoritmo BSP/CGM tem tempo de computação local $O(\frac{m+n}{p})$, consumindo $O(\frac{m+n}{p})$ memória utilizando $O(\log p)$ rodadas de comunicação, na qual são trocados no máximo $O(\frac{m}{p})$ dados. O algoritmo BSP/CGM foi implementado na linguagem C utilizando a biblioteca para troca de mensagens MPI e executado em um *Beowulf* com 66 processadores *Pentium*.

Na implementação foram utilizados algoritmos seqüenciais para as rotinas de ordenação, *list ranking* e árvore geradora. Assim, os tempos de execução do algoritmo obtidos consideram apenas os custos do algoritmo de circuitos de Euler, ou seja, a complexidade dos algoritmos de ordenação, *list ranking* e árvore geradora são ignorados, o que não influencia no nosso objetivo de validar o modelo BSP/CGM. Além disso, citamos trabalhos que implementaram e apresentaram resultados experimentais para esses problemas.

Os dados experimentais obtidos mostraram um bom desempenho do algoritmo, principalmente para grafos maiores, onde o tempo gasto com a computação local determina o tempo de execução total do algoritmo. Os dados também mostraram que a partir de um determinado número de processadores, a adição de novos processadores não melhora o desempenho do algoritmo.

Conseguimos um algoritmo inédito para obter circuitos de Euler em grafos para modelos de memória distribuída de granularidade grossa. Além disso a implementação do algoritmo validou o modelo BSP/CGM, como mostram os resultados apresentados neste trabalho.

A contribuição de nosso trabalho consiste na apresentação de um algoritmo BSP/CGM para computar um circuito de Euler num grafo euleriano. Além disso, implementamos o referido algoritmo cujo código pode vir a fazer parte de uma biblioteca de algoritmos BSP/CGM. Uma outra contribuição é que os Capítulos 2 e 4 podem ser usados como material didático auxiliar em disciplinas de Algoritmos Paralelos e Distribuídos.

Apêndice A

MPI

A.1 Introdução

O MPI (*Message Passing Interface*) não é uma linguagem de programação e sim uma biblioteca de definições e funções que podem ser usadas em programas C, C++ ou Fortran. Neste apêndice, descrevemos o funcionamento do MPI para a distribuição LAM.

LAM é um *deamon* baseado na implementação MPI. Inicialmente o programa `lamboot` distribui os *deamons* do LAM baseado em uma lista de máquinas fornecidas pelo usuário. Esses *deamons* permanecem inativos na lista de máquinas remotas até que eles recebam uma mensagem para carregar o executável do MPI para iniciar a execução. Só podemos executar programas MPI enquanto os *deamons* estiverem atuando em *background*. Para desativar os *deamons* basta executar o programa `lamhalt`.

Como exemplo considere o seguinte arquivo, chamado `hostfile`, que contém uma lista de máquinas onde devem ser executados os *deamons* do MPI.

```
host0
host1
host2
host3
```

Para iniciar o *deamons* nessas quatro máquinas basta executar o comando `lamboot -v hostfile`. De maneira análoga para desativar os *deamons* execute o comando `lamhalt` ou o comando `wipe -v hostfile`.

Para compilar um programa que utiliza a linguagem C e a biblioteca `mpi.h` utilize o programa `mpicc`. Por exemplo, se o programa a ser compilado se chama `prog.c` e programa executável correspondente irá se chamar `prog`, utilize

```
mpicc -o prog prog.c
```

Depois que o programa executável for gerado ele poderá ser executado através do comando `mpirun`. Por exemplo, o programa `prog` pode ser executado por 4 processos distintos em uma única máquina. Para tanto utilize

```
mpirun -c 4 prog
```

Também podemos executar o programa `prog` em 4 máquinas diferentes, criando um processo em cada máquina. Neste caso utilize

```
mpirun n0-4 prog
```

As próximas seções apresentam uma descrição das principais rotinas implementadas pelo LAM MPI. Uma descrição mais detalhadas destas e de outras rotinas pode ser obtidas em [Pac97].

A.2 Programas MPI

Todo programa MPI deve incluir o arquivo `mpi.h` que possui as definições e declarações necessárias para compilar um programa MPI. Antes que qualquer outra função MPI seja utilizada, a função `MPI_Init` deve ser chamada. Os parâmetros desta função são ponteiros para os parâmetros da função `main`, `argc` e `argv`. Isso permite que o sistema realize qualquer configuração especial para que a biblioteca MPI possa ser utilizada. Ao final de um programa MPI uma chamada a função `MPI_Finalize` deve ser feita. Assim, um típico programa MPI possui o seguinte formato:

```
#include <mpi.h>

main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);
```

```
   /* Chamadas a funcoes MPI */

   MPI_Finalize();
}
```

A.3 Identificação das Tarefas

Os programas MPI geralmente utilizam o modelo SPMD. O fluxo de controle em um programa SPMD depende da identificação do processo. Uma tarefa obtém o seu identificador (`rank`) chamando a função `MPI_Comm_rank`, que retorna a identificação de um processo em seu segundo parâmetro. O primeiro parâmetro é um comunicador. Um comunicador define um conjunto de processos que podem enviar mensagens uns para os outros. O comunicador `MPI_COMM_WORLD` é pré-definido pelo MPI e consiste de todas as tarefas na aplicação MPI.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

É comum também que as tarefas necessitem saber quantas tarefas estão sendo executadas em uma aplicação. Essa informação é obtida chamando-se a função `MPI_Comm_size`, que retorna o número de tarefas em um comunicador no seu segundo parâmetro. O primeiro parâmetro é um comunicador.

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

A.4 Enviando e Recebendo Mensagens

Em ambientes de troca de mensagens, as tarefas coordenam suas atividades através do envio e do recebimento de mensagens. Uma mensagem é um vetor de elementos de um determinado tipo de dados. Uma mensagem é enviada para uma tarefa específica através da função `MPI_Send`. A função `MPI_Recv` recebe uma mensagem de uma determinada tarefa. A sintaxe dessas funções é:

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tabela A.1: Tipos de dados pré-definidos pelo MPI e os correspondentes tipos em C.

```
MPI_Send(message, count, datatype, destination, tag,
communicator);
MPI_Recv(message, count, datatype, source, tag,
communicator, status);
```

O conteúdo da mensagem está armazenada em `message`. Os parâmetros `count` e `datatype` permitem que o sistema determine quanto espaço é necessário para armazenar a mensagem: a mensagem contém uma seqüência de `count` valores do tipo `datatype`. Os tipos de dados definidos pelos MPI estão na Tabela A.1.

O parâmetro `destination` é o `rank` da tarefa que irá receber a mensagem, e o parâmetro `source` é o `rank` da tarefa que está enviando a mensagem. Existe uma constante pré-definida, `MPI_ANY_SOURCE`, que pode ser usada se uma tarefa deseja receber um mensagem de qualquer tarefa ao invés de receber de uma tarefa específica.

O parâmetro `tag` é um inteiro que pode ser utilizado para distinguir diferentes tipos de mensagens que uma tarefa pode enviar ou receber. Existe uma constante pré-definida, `MPI_ANY_TAG`, que pode ser usada pela função `MPI_Recv` quando a tarefa deseja receber uma mensagem independente da `tag` que ela possui.

O último parâmetro da função `MPI_Recv` retorna informações sobre o dado recebido, tais como, tarefa que enviou a mensagem, tamanho da mensagem recebida, entre outros.

A.5 Comunicação Coletiva

Uma comunicação que envolve todos os processos de um comunicador é chamada de comunicação coletiva. O MPI possui várias funções para comunicação coletiva que serão descritas nesta seção.

A.5.1 Broadcast

O *broadcast* é uma operação de comunicação coletiva onde uma única tarefa envia o mesmo dado para todas as tarefas do comunicador. Esta operação é implementada em MPI pela função `MPI_Bcast`:

```
MPI_Bcast(message, count, datatype, root, comm);
```

A função envia uma cópia do dado em `message`, armazenado na tarefa `root`, para todas as tarefas do comunicador `comm`. Observe que o dado enviado por `root` será armazenado, pelas demais tarefas, em `message`. `MPI_Bcast` deve ser chamada por todas as tarefas com os mesmos argumentos para `root` e `comm`. Os parâmetros `count` e `datatype` possuem as mesmas funções descritas anteriormente.

A.5.2 Reduce e Allreduce

A operação *reduce* permite que todas as tarefas de um comunicador contribua com um dado que é combinado usando uma operação binária. A função MPI que implementa a operação reduce é:

```
MPI_Reduce(operand, result, count, datatype, operator, root,
comm);
```

A função `MPI_Reduce` combina os valores armazenados em `operand` usando a operação `operator` e armazena o resultado em `result` da tarefa `root`. A função deve ser chamada por todas as tarefas do comunicador `comm`, e `count`,

Operação	Significado
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Soma
MPI_PROC	Multiplicação
MPI_LAND	E lógico
MPI_LOR	OU lógico
MPI_LXOR	XOR lógico
MPI_MAXLOC	Máximo e localização do máximo
MPI_MINLOC	Mínimo e localização do mínime

Tabela A.2: Operações pré-definidas pelo MPI para a função reduce.

`datatype`, `operator` e `root` devem ser os mesmos para todas as tarefas. O parâmetro `operator` pode assumir um dos valores pré-definidos listados na Tabela A.2.

Como exemplo, suponha que cada tarefa possui uma variável que armazena um contador, e que desejamos somar os contadores de todas as tarefas para obter um contador global:

```
MPI_Reduce(&cont, &cont_global, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Todas as tarefas chamam a função `MPI_Reduce` com os mesmos argumentos, embora `cont_global` só faça sentido para o processo 0, que acumula os contadores.

A função `MPI_Allreduce` é usada da mesma maneira que a função `MPI_Reduce`. A única diferença é que o resultado é armazenado em todos os processos, ao invés de ser armazenado apenas na tarefa raiz. Sintaxe:

```
MPI_Allreduce(operand, result, count, datatype, operator, root,
comm);
```

A.5.3 Gather e Allgather

Gather é uma operação de comunicação coletiva onde a tarefa raiz recebe dados de todos os outros processos do comunicador. É implementada pelo MPI através da função `MPI_Gather`. Sintaxe:

```
MPI_Gather(send_data, send_count, send_type, recv_data,
recv_count, recv_type, root, comm);
```

A função `MPI_Gather` coleta o dado armazenado em `send_data` de cada tarefa do comunicador `comm` e armazena os dados coletados em `recv_data` na ordem dos `ranks` das tarefas. Assim, o dado da tarefa 0 é seguido pelo dado da tarefa 1, que é seguido do dado da tarefa 2 e assim por diante. Os parâmetros `root` e `comm` devem ser idênticos em todas as tarefas do comunicador `comm`.

A função `MPI_Allgather` é usada da mesma maneira que a função `MPI_Gather`. A única diferença é que o conteúdo de `send_data` de cada tarefa é armazenado em `recv_data` de cada tarefa, ao invés de ser armazenado apenas em `recv_data` da tarefa `root`. Sintaxe:

```
MPI_Gather(send_data, send_count, send_type, recv_data,
recv_count, recv_type, comm);
```

A.5.4 Scatter

Scatter é uma operação de comunicação coletiva onde a tarefa raiz envia um conjunto de dados distintos para cada processo do comunicador. É implementada pelo MPI através da função `MPI_scatter`. Sintaxe:

```
MPI_Scatter(send_data, send_count, send_type, recv_data,
recv_count, recv_type, root, comm);
```

A função `MPI_Scatter` divide o dado referenciado por `send_data`, na tarefa `root`, em p segmentos, cada segmento contendo `send_cont` elementos do tipo `send_type`. O primeiro segmento é enviado para a tarefa 0, o segundo para a tarefa 1 e assim sucessivamente. Os parâmetros `root` e `comm` devem ser idênticos em todas as tarefas do comunicador `comm`.

A.5.5 Alltoall e Alltoallv

A operação troca total (*total exchange*) é uma operação de comunicação coletiva onde cada tarefa envia um conjunto de dados distintos para todas as outras tarefas do comunicador. Existem duas formas de realizar a troca

total em MPI: `MPI_Alltoall` e `MPI_Alltoallv`. A primeira forma é utilizada quando desejamos enviar a mesma quantidade de dados para todos os processos. Sintaxe:

```
MPI_Alltoall(send_buffer, send_count, send_type, recv_buffer,
recv_count, recv_type, comm);
```

Esta é uma operação de comunicação coletiva então todas as tarefas do comunicador `comm` devem chamar a função. A função faz com que a tarefa t envie `send_count` elementos de tipo `send_type` para todas as tarefas. O primeiro bloco de `send_count` elementos vai para a tarefa 0, o segundo bloco para a tarefa 1, e assim sucessivamente. A tarefa t também irá receber `recv_count` elementos do tipo `recv_type` de todas as demais tarefas. Os elementos da tarefa 0 são recebidos no início de `recv_buffer`. Os elementos da tarefa 1 são recebidos imediatamente após os elementos da tarefa 0, e assim por diante.

A função `MPI_Alltoallv` é usada da mesma maneira que a função `MPI_Alltoall`. A única diferença é que cada tarefa envia/recebe quantidades diferentes de dados para/de cada tarefa.

```
MPI_Alltoallv(send_buffer, send_count[], send_displ[], send_type,
recv_buffer, recv_count[], recv_displ[], recv_type, comm);
```

`Send_cout` e `recb_count` são vetores contendo a quantidade de elementos enviados/recebidos para/de cada tarefa. `Send_displ` contém a posição do vetor `send_buffer` a partir da qual `send_conts` elementos devem ser enviados para a tarefa correspondente. De maneira análoga `recv_displ` contém a posição de `recv_buffer` a partir da qual `recv_counts` elementos devem ser recebidos da tarefa correspondente. Observe que os vetores `send_count`, `send_displ`, `recv_count` e `recv_displ` possuem p posições, onde p é o número de tarefas no comunicador `comm`.

A.6 Medindo Tempos

Se você tem acesso exclusivo ao processador que está executado o seu programa, é bastante simples medir o tempo de execução de um programa. Basta sincronizar os processos no início do código que você deseja cronometrar, iniciar um relógio em cada processo, sincronizar os processos no final do código, parar o relógio e computar o tempo.

Para sincronizar os processos utilizamos o função `MPI_Barrier`, que faz com que cada processo do comunicador `comm` fique bloqueado até que todos os processos do comunicador alcancem a chamada desta função.

```
int MPI_Barrier( comm );
```

Para medir o tempo o MPI oferece a função `MPI_Wtime` que retorna um valor que representa o número de segundos que se passaram desde o último ponto em que ela foi chamada. Assim, um programa MPI pode determinar o tempo transcorrido da seguinte maneira:

```
double start, finish, total;

MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

/* Trecho do programa a ser cronometrado */

MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();

total = finish - start;
```

Apêndice B

Código Fonte

Neste apêndice, listamos os códigos fontes dos programas EulerTour e Gerador. O programa EulerTour implementa o algoritmo BSP/CGM para computar circuitos de Euler em grafos descrito no capítulo 4. Foi escrito em C utilizando a biblioteca para troca de mensagens MPI. O programa Gerador foi utilizado para gerar os grafos eulerianos que servem como entrada para o programa EulerTour.

B.1 O Programa EulerTour

```
/*-------------------------------------------------------[]
 Programa: EulerTour.c
 Programador: Claudia Nasu/Edson Norberto Caceres
 Data: 23/08/2002

 O Dialogo: Este programa recebe um digrafo euleriano repre-
 sentado atraves de suas arestas e envia m/p arestas para
 cada tarefa. O programa computa o circuito de Euler do
 digrafo.
[]-------------------------------------------------------*/
#include<mpi.h>
#include<stdio.h>
#include<math.h>
#include<string.h>

// Declaracao das constantes globais
```

```
#define TAMANHO 16 //numero de vertices do grafo

#define TAMMAX 32 //numero de arestas do grafo

#define TAMMAXA 64 //no. max. de arestas da arvore geradora

#define TAMMAXV 32 //no. max. de vertices da arvore geradora

#define NTAREFAS 4 //no. max. de tarefas

// Definicao dos tipos de dados utilizados pelo programa
typedef unsigned long int Matriz[TAMMAX][2];

typedef unsigned long int Vetor[TAMMAXV];

typedef unsigned long int MatrizA[TAMMAXA][2];

typedef enum{false, true} boolean;

typedef struct{ unsigned long int u, v[2];
                unsigned long int rank, rot_u, rot_v;
                unsigned long int indice, grau;
                unsigned long int bp;
                unsigned long int sp;
                unsigned long int st;
               }Bipart; // Grafo bipartido auxiliar

typedef struct{ unsigned long int aresta[2], suc[2];
                unsigned long int rank, indice;
              }Stitch; // Suporte

typedef struct{ unsigned long int v;
                unsigned long int ini, fim;
              }Lista; // sub-lista

typedef struct{ unsigned long int v;
                unsigned long int grau;
              }Vertice;

// declaracao das variaveis globais
Matriz MatrizEdge; // arestas do grafo
```

```
Matriz MatrizSuc; // sucessores das arestas do grafo

Matriz MatrizAresta; // ordenadas pelo vertice origem

Matriz MatrizSucessor; // ordenadas pelo vertice destino

Matriz CycleRep; // representantes dos ciclos

Matriz CycleRepAux; // auxiliar

unsigned long int Tarefas[TAMANHO][NTAREFAS];

Bipart *Grafo, *GrafoAux, aux;  // grafo bipartido

Stitch temp, *Aux; // stitch

Lista *Sub; // sublistas das costuras

Vertice *Vert; // vertices do grafo

unsigned long int *L, *inicio, *fim, *pos;

unsigned long int *sbuf; // tamanho do buffer de envio

unsigned long int *send_displ; // pos. do buffer de envio

unsigned long int *rbuf; // tamanho do buffer de recebimento

unsigned long int *rec_displ;// pos. do buffer de recebimento

unsigned long int *buf_send; // buffer de envio

unsigned long int *buf_rec; // buffer de recebimento

int rank, size;  // id. da tarefa e numero de tarefas

unsigned long int tam_a, tam_v, tam_din1, tam_din2;

unsigned long int num_ver, num_ares, size_e;
```

```
unsigned long int ncyc, numelem, tamS, tamR;

unsigned long int tam_g, tam_b, tam_sp;

unsigned long int i, j, k, l, h, cont, rounds = 0;

unsigned long int *P; // variavel auxiliar

int root = 0; // tarefa raiz

double start, finish, total = 0.0; // medida de tempo

double *buff; // variavel auxiliar

MatrizA MatrizDadosB;

unsigned long int MatrizIndices[TAMMAXV][2];

Vetor Vertices; // Conjunto de vertices

Vetor Cor; // indica a marcacao dos vertices

Vetor Fila; // Fila dos vertices visitados

Vetor Pai; // Vertice anterior no percurso

unsigned long int ArestasArvore[TAMMAXA];

unsigned long int vertice, nvert, narest;

/*-------------------------------------------------------[]
                  Funcao principal
[]-------------------------------------------------------*/

int main(int argc, char *argv[]) {
// Declaracao das variaveis locais
 FILE *Arq;
 boolean compl;
 char ch, file_name[30];
 MPI_Status status;
// declaracao das funcoes locais
```

```
 int ComparE(const unsigned long int*,const unsigned long int*);
 int ComparB(const void *, const void *);
 int ComparS(const void *, const void *);
 int ComparBipart(const void *, const void *);

// Passo 1. Inicializacao
 MPI_Init(&argc, &argv); // iniciacao do MPI
 MPI_Comm_size(MPI_COMM_WORLD, &size); // numero de tarefas
 MPI_Comm_rank(MPI_COMM_WORLD, &rank); // id. da tarefa
 tam_a = (unsigned long int) TAMMAX/size;
 if(rank == root)
    strcpy(file_name, "Euler");
 else
    strcpy(file_name, "/home/tmp/Euler");
// Passo 1.2 Inicialize a Matriz Dados
 for(i = 0; i < TAMMAX; i++) {
    MatrizEdge[i][0] = MatrizEdge[i][1] = 0;
    MatrizSuc[i][0]  = MatrizSuc[i][1]  = 0;
 }
// Passo 2. Leia os dados a partir do arquivo de entrada
 if (rank == root) {
  Arq = fopen("Egrafo256a.txt", "r");
  fscanf(Arq, "%ld%ld", &num_ver, &num_ares);
  for(i = 0; i < num_ares; i++)
   fscanf(Arq,"%ld%ld",&MatrizEdge[i][0],&MatrizEdge[i][1]);
  for(i = 0; i < num_ares; i++)
   fscanf(Arq,"%ld%ld",&MatrizSuc[i][0],&MatrizSuc[i][1]);
  fclose(Arq); // Fecha o arquivo de dados
// Passo 2.1. Crie o vetor de ponteiros P
// Passo 2.1.1 Dimensione o vetor P
   P = (unsigned long int *) malloc(TAMMAX *
        sizeof(unsigned long int));
// Passo 2.1.2 Determine a posicao de cada aresta Suc em Edge
   for( i = 0; i < num_ares; i++ )
     for( j = 0; j < num_ares; j++ )
       if( MatrizSuc[i][0] == MatrizEdge[j][0] &&
           MatrizSuc[i][1] == MatrizEdge[j][1] ) {
            P[i] = j;
            break;
       }// fim if
// Passo 3. Calcule os representantes de ciclos
```

```
// Passo 3.1. Copie o vetor Suc para o vetor Aux
   for( i = 0; i < num_ares; i++ ) {
     CycleRepAux[i][0] = MatrizSuc[i][0];
     CycleRepAux[i][1] = MatrizSuc[i][1];
   }// fim for
// Passo 3.2. Execute duplicacao recursiva
//            para determinar representante de ciclo
   for(i=0;i<(unsigned long int)log(num_ares)/log(2);i++)
   {
     for(j=0; j<num_ares; j++ ) // Para cada aresta
     {
// Passo 3.2.1. Representante eh a menor aresta do ciclo
       if( !(CycleRepAux[j][0] < CycleRepAux[P[j]][0] ||
            (CycleRepAux[j][0] == CycleRepAux[P[j]][0] &&
             CycleRepAux[j][1] < CycleRepAux[P[j]][1])) ) {
                CycleRepAux[j][0] = CycleRepAux[P[j]][0];
                CycleRepAux[j][1] = CycleRepAux[P[j]][1];
       }// fim if
       P[j] = P[P[j]]; // Proxima aresta do ciclo
     }// fim for j
   }// fim for i
// Passo 3.2.2. Libere o espaco atribuido a P
   free(P);
 } // fim if rank == 0
// Passo 4. Envie os dados as tarefas filhos
// Passo 4.1. Inicialize Aresta e Sucessor de cada tarefa
 for (j = 0; j < tam_a; j++) {
    MatrizAresta[j][0] = MatrizAresta[j][1] = 0;
    MatrizSucessor[j][0] = MatrizSucessor[j][1] = 0;
 }
// Inicio da tomada de tempo
 MPI_Barrier(MPI_COMM_WORLD);
 start = MPI_Wtime();
// Passo 4.2. Distribua os dados entre os processadores
 MPI_Scatter(MatrizEdge, 2*tam_a, MPI_UNSIGNED_LONG,
             MatrizAresta, 2*tam_a, MPI_UNSIGNED_LONG,
             root, MPI_COMM_WORLD);
 MPI_Scatter(MatrizSuc, 2*tam_a,  MPI_UNSIGNED_LONG,
             MatrizSucessor, 2*tam_a, MPI_UNSIGNED_LONG,
             root, MPI_COMM_WORLD);
 MPI_Scatter(CycleRepAux, 2*tam_a,  MPI_UNSIGNED_LONG,
```

```
            CycleRep, 2*tam_a, MPI_UNSIGNED_LONG,
            root, MPI_COMM_WORLD);
 tam_din1 = tam_a;
 tam_din2 = tam_a;
 for( i = 0; i < tam_a; i++ ) { // Descarta arestas (0,0)
   if(MatrizAresta[i][0] == 0 && MatrizAresta[i][1] == 0)
      tam_din1--;
   if(MatrizSucessor[i][0] == 0 && MatrizSucessor[i][1] == 0)
      tam_din2--;
 } // fim for
// Passo 5. Calcule o numero de circuitos
 numelem = 0; // No. de representantes de ciclos da tarefa
// Passo 5.1. Encontrar o no. de representantes em cada tarefa
 for( i = 0; i < tam_din1; i++ )
   if( MatrizAresta[i][0] == CycleRep[i][0] &&
       MatrizAresta[i][1] == CycleRep[i][1] )
         numelem++;//Incrementa no. de representantes de ciclos
// Passo 5.2 Raiz acumula o numero de ciclos
// Passo 5.2..1. Envia numero de representantes para raiz
 MPI_Reduce(&numelem,&ncyc,1,MPI_UNSIGNED_LONG,MPI_SUM,
            root,MPI_COMM_WORLD);
 rounds++;// Incrementa o no. de rodadas de comunicacao
// Passo 5.3. Raiz envia ncyc para demais tarefas
 MPI_Bcast(&ncyc,1,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 6. Computar o grafo bipartido auxiliar
// Passo 6.1. Dimensione a matriz Grafo
 Grafo = (Bipart *) malloc(TAMMAX*sizeof(Bipart));
// Passo 6.2. Construa o grafo bipartido auxiliar
 tam_g = 1; // Numero de arestas do grafo bipartido
 j = 1;
// Passo 6.2.1 Define a primeira aresta do grafo auxiliar
 Grafo[0].u = MatrizAresta[0][1]; // Vertice
 Grafo[0].v[0] = CycleRep[0][0];  // Representante de ciclo
 Grafo[0].v[1] = CycleRep[0][1];  // Representante de ciclo
 Grafo[0].bp = Grafo[0].st = 0;
 Grafo[0].indice = 0; // indice da aresta que a originou
// Passo 6.3. Eliminar as replicas presentes na tarefa
 for( i = 1; i < tam_din1; i++ ) {
   l = 0;
   for( k = 0; k < j; k++ ) // Procura replicas
```

```
    if( MatrizAresta[i][1] == Grafo[k].u &&
        CycleRep[i][0] == Grafo[k].v[0] &&
        CycleRep[i][1] == Grafo[k].v[1] ) {
          l = 1; // Aresta jah pertence ao grafo bipartido
          break;
    } // fim if
    if( l == 0 ) { // Se nao ha replica, inclua a aresta
      Grafo[j].u = MatrizAresta[i][1];
      Grafo[j].v[0] = CycleRep[i][0];
      Grafo[j].v[1] = CycleRep[i][1];
      Grafo[j].bp = Grafo[j].st = 0;
      Grafo[j].indice = i;
      tam_g++;// Incrementa no. de arestas do grafo bipartido
      j++;
    } // fim if
 }// fim for i
// Passo 6.4. Ordene as arestas do grafo bipartido
 qsort(Grafo, tam_g, sizeof(Bipart), ComparBipart);
// Passo 6.5 Elimine as replicas das outras tarefas
// Passo 6.5.1  Dimensione o vetor send_displ e inicialize
 sbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
 for( i = 0; i < size; i++ )
   sbuf[i] = 0; // Inicializa sbuf
//Passo 6.5.2 Determina no. arestas que incidem no ultimo vert.
 cont = 0;
 for(i=0; i<tam_g; i++)
   if( Grafo[i].u == Grafo[tam_g-1].u )
      cont++; // incrementa o numero de arestas
// Passo 5.5.3 Dimensione o vetor de envio
 buf_send = (unsigned long int *)malloc(((3*cont) + 1) *
             sizeof(unsigned long int));
// Passo 5.5.4 Prepara buffer de envio contendo as arestas
 k = 0;
 for( i = 0; i < tam_g; i++ ) {
   if( Grafo[i].u == Grafo[tam_g-1].u ) {
      buf_send[k++] = Grafo[i].u;
      buf_send[k++] = Grafo[i].v[0];
      buf_send[k++] = Grafo[i].v[1];
   }
 }
```

```
// Passo 6.5.5 Se existe tarefa p+1 envie as arestas de p
 if( (rank + 1) < size )
   sbuf[rank+1]=cont;// No. arestas a serem enviadas para p
// Passo 6.5.6 Compute o vetor das posicoes de envio
 send_displ = (unsigned long int *) malloc(size*
               sizeof(unsigned long int));
 send_displ[0] = 0;
 sbuf[0] = 3*sbuf[0];
 for (i = 1; i < size; i++) {
   send_displ[i] = send_displ[i-1] + sbuf[i-1];
   sbuf[i] = 3*sbuf[i];
 }
// Passo 6.5.7 Dimensione o vetor de recebimento
 rbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
// Passo 6.5.8  Envie/receba o no. de vertices
 MPI_Alltoall(sbuf, 1, MPI_UNSIGNED_LONG, rbuf, 1,
               MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
 rounds++;// Incrementa no. de rodadas de comunicacao
// Passo 6.5.9 Dimensione o vetor de armazenamento
 rec_displ = ( unsigned long int*)malloc(size*
               sizeof(unsigned long int));
// Passo 6.5.10  Compute as posicoes de recebimento
 rec_displ[0] = 0;
 tamR = rbuf[0];
 for( i = 1; i < size; i++ ) {
   rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
   tamR += rbuf[i];
 }
// Passo 6.5.11 Dimensione o vetor de armazenamento
 buf_rec = ( unsigned long int *) malloc((1 + tamR)*
             sizeof(unsigned long int));
// Passo 6.5.12 Envie/receba os vertices para/de as tarefas
 MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
 buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 6.5.13 Libere espaco alocado para buffers
 free(buf_send);
 free(sbuf);
 free(send_displ);
 free(rbuf);
```

```
 free(rec_displ);
// Passo 6.6 Verifique se esta aresta estah replicada
 for( j = 0; j < tam_g; j++ ) {
   for( k = 0; k < tamR; k += 3 ) {
// Passo 6.6.1 Elimine aresta replicada, se houver
     if( Grafo[j].u == buf_rec[k] &&
         Grafo[j].v[0] == buf_rec[k+1] &&
         Grafo[j].v[1] == buf_rec[k+2] ) {
       for( i = j+1; i < tam_g; i++ ) {
           Grafo[i-1].u = Grafo[i].u;
           Grafo[i-1].v[0] = Grafo[i].v[0];
           Grafo[i-1].v[1] = Grafo[i].v[1];
           Grafo[i-1].indice = Grafo[i].indice;
           Grafo[i-1].bp = Grafo[i].bp;
           Grafo[i-1].st = Grafo[i].st;
       }// fim for
       tam_g--; // Desconsidere a replica eliminada
     }// fim if
   } // fim for k
 } // fim for j
 free(buf_rec);// Libera espaco alocado para rec_displ
// Passo 7. Eliminar os vertices de grau < 2
// Passo 7.0. Raiz envia num_ver para demais tarefas
 MPI_Bcast(&num_ver,1,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
 rounds++;// Incrementa no. de rodadas de comunicacao
// Passo 7.1. Calcule o numero de vertices de cada tarefa
 tam_v = (unsigned long int)TAMANHO/size;
 Vert = (Vertice *)malloc(tam_v*sizeof(Vertice));
// Passo 7.2 Inicialize Vert com os vertices de cada tarefa
 for( i = 0; i < tam_v; i++ ) {
  if( (i+rank*tam_v) <  num_ver ){// Se eh um rotulo valido
     Vert[i].v=(i+rank*tam_v);//Atribui rotulos aos vertices
     Vert[i].grau = 0; // Inicializa o grau do vertice
   }
   else Vert[i].v = num_ver; // Nao existe este vertice
 }
// Passo 7.3 Detemine o grau de cada vertice
// Passo 7.3.1 Dimensione o vetor send_displ e inicialize
 sbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
 buf_send = ( unsigned long int *)malloc(2*tam_g *
```

```
              sizeof(unsigned long int));
 for( i = 0; i < size; i++ )
    sbuf[i] = 0; // Inicializa sbuf
 for( i = 0; i < 2*tam_g; i++ )
    buf_send[i] = 0; // Inicializa buf_send
// Passo 7.3.2 Inicialize o grau de cada vertice
 j = 0;
 buf_send[0] = Grafo[0].u;
 for( i = 0; i < tam_g; i++ ) {
   if( Grafo[i].u == buf_send[j] )
      buf_send[j+1]++;
   else {
      j += 2;
      buf_send[j] = Grafo[i].u;
      buf_send[j+1]++;
   } // fim if-else
 }// fim for
 j += 2; // tamanho de buf_send
// Passo 7.3.3 Calcule o no. de vert. a ser enviado
 for( i = 0; i < j; i += 2 )
   sbuf[(int)buf_send[i]/tam_v]++;
// Passo 7.4 Compute o vetor das posicoes de envio
 send_displ = ( unsigned long int *)malloc(size *
                sizeof(unsigned long int));
 send_displ[0] = 0;
 sbuf[0] = 2*sbuf[0];
 for (i = 1; i < size; i++) {
    send_displ[i] = send_displ[i-1] + sbuf[i-1];
    sbuf[i] = 2*sbuf[i];
 }
// Passo 7.4.1 Dimensione o vetor de recebimento
 rbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
// Passo 7.4.2 Envie/receba o no. de vertices
 MPI_Alltoall(sbuf,1,MPI_UNSIGNED_LONG,rbuf,1,
              MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
 rounds++;//Incrementa no. rodadas de comunicacao
// Passo 7.5. Dimensione o vetor de armazenamento
 rec_displ = ( unsigned long int *) malloc(size *
               sizeof(unsigned long int));
// Passo  7.6. Compute as posicoes de recebimento
```

```
 rec_displ[0] = 0;
 tamR = rbuf[0];
 for( i = 1; i < size; i++ ) {
    rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
    tamR += rbuf[i];
 }
// Passo 7.7 Dimensione o vetor de armazenamento
 buf_rec = ( unsigned long int *) malloc(tamR *
             sizeof(unsigned long int));
// Passo 7.8. Envie/receba os vertices
 MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
 buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 7.9  Libere o espaco alocado para os buffers
 free(buf_send);
 free(sbuf);
 free(send_displ);
 free(rbuf);
 free(rec_displ);
// Passo 7.10. Calcule o grau de cada vertice da tarefa
 for( i = 0; i < tamR; i += 2 )
   for( j = 0; j < tam_v; j++ )
      if( buf_rec[i] == Vert[j].v ) {
         Vert[j].grau += buf_rec[i+1];
         break;
      } // fim if
// Passo 7.10.1  Libere o espacao atribuido a buf_rec
 free(buf_rec);
// Passo 8.Elimine vertices com grau < 2 do grafo auxiliar
// Passo 8.1.Elimine vertices com grau < 2 do processador
 tam_b = 0;
 for(  i = 0; i < tam_g; i++ )
   for( j = 0; j < tam_v; j++ ) {
     if( Grafo[i].u == Vert[j].v && Vert[j].grau >= 2 ) {
       Grafo[i].bp = 1;//Aresta pertence ao grafo bipartido
       tam_b++;
       break;
     }
   } // fim for
// Passo 8.2 Elimine vertices com grau < 2 dos outros proc.
// Passo 8.2.1 Cada proc. envia pedido para processadores
```

```
// que guardam graus dos vert. que aparecem em suas arestas
// Passo 8.2.1.1 Dimensione o vetor send_displ e inicialize
 sbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
 buf_send = ( unsigned long int *)malloc(tam_g *
              sizeof(unsigned long int));
 for( i = 0; i < size; i++ )  sbuf[i] = 0;
 for( i = 0; i < tam_g; i++ ) buf_send[i] = 0;
// Passo 8.2.1.2 Inicie os pedidos para as demais tarefas
 j = 0;
 for( i = 0; i < tam_g; i++ )
  if(Grafo[i].u < Vert[0].v ||Grafo[i].u > Vert[tam_v-1].v)
      buf_send[j++] = Grafo[i].u;
// Passo 8.2.1.3 Calcule no. vert. a ser enviado
 for( i = 0; i < j; i++ )
    sbuf[(int)buf_send[i]/tam_v]++;
// Passo 8.2.2  Compute o vetor das posicoes de envio
 send_displ = ( unsigned long int *) malloc(size *
                sizeof(unsigned long int));
 send_displ[0] = 0;
 sbuf[0] = sbuf[0];
 for (i = 1; i < size; i++)
    send_displ[i] = send_displ[i-1] + sbuf[i-1];
// Passo 8.2.3  Dimensione o vetor de recebimento
 rbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
// Passo 8.2.4. Envie/receba o numero de vertices
 MPI_Alltoall(sbuf, 1,  MPI_UNSIGNED_LONG, rbuf, 1,
             MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
 rounds++; // Incrementa no. de rodadas de comunicacao
// Passo 8.2.5 Dimensione o vetor de armazenamento
 rec_displ = ( unsigned long int *) malloc(size *
               sizeof(unsigned long int));
// Passo 8.2.6. Compute as posicoes de recebimento
 rec_displ[0] = 0;
 tamR = rbuf[0];
 for (i = 1; i < size; i++) {
   rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
   tamR += rbuf[i];
 }
// Passo 8.2.7 Dimensione o vetor de armazenamento
```

```
 buf_rec = ( unsigned long int *) malloc(tamR *
             sizeof(unsigned long int));
// Passo 8.2.8. Envie/receba os vertices para/de as tarefas
 MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
  buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comuncicacao
// Passo 8.2.9 Libere espaco alocado para buffers
 free(buf_send);
 free(sbuf);
 free(send_displ);
// Passo 8.3. Determina grau dos vert. pedidos e envia
// Passo 8.3.1 Dimensione o vetor send_displ e inicialize
 sbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
 buf_send = ( unsigned long int*) malloc( 2*tamR*
              sizeof(unsigned long int));
 for( i = 0; i < size; i++ )   sbuf[i] = 0;
 for( i = 0; i < 2*tamR; i++ ) buf_send[i] = 0;
// Passo 8.3.2 Determina o grau dos vertices pedidos
 k = 0;
 for( i = 0; i < tamR; i++ )
   for( j = 0; j < tam_v; j++ )
     if( Vert[j].v == buf_rec[i] ) {
         buf_send[k++] = buf_rec[i];
         buf_send[k++] = Vert[j].grau;
     } // fim if
// Passo 8.3.3 Calcule no. vert. a ser enviado
 for( i = 0; i < size; i++ ) // a cada tarefa
    sbuf[i] = rbuf[i];
// Passo 8.3.4 Libere espaco alocado aos buffers
 free(rbuf);
 free(rec_displ);
 free(buf_rec);
// Passo 8.3.5. Compute o vetor das posicoes de envio
 send_displ = ( unsigned long int *)malloc(size *
                sizeof(unsigned long int));
 send_displ[0] = 0;
 sbuf[0] = 2*sbuf[0];
 for (i = 1; i < size; i++) {
    send_displ[i] = send_displ[i-1] + sbuf[i-1];
    sbuf[i] = 2*sbuf[i];
```

```
 }
// Passo 8.3.6  Dimensione o vetor de recebimento
 rbuf = ( unsigned long int *) malloc(size *
          sizeof(unsigned long int));
// Passo 8.3.7 Envie/receba o no. de vertices
 MPI_Alltoall(sbuf,1,MPI_UNSIGNED_LONG,rbuf,1,
            MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
 rounds++;// Incrementa no. de rodadas de comunicacao
// Passo 8.3.7 Dimensione o vetor de armazenamento
 rec_displ = ( unsigned long int *) malloc(size*
               sizeof(unsigned long int));
// Passo 8.3.8 Compute as posicoes de recebimento
 rec_displ[0] = 0;
 tamR = rbuf[0];
 for (i = 1; i < size; i++) {
    rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
    tamR += rbuf[i];
 }
// Passo 8.3.9 Dimensione o vetor de armazenamento
 buf_rec = ( unsigned long int *) malloc(tamR *
             sizeof(unsigned long int));
// Passo 8.3.10  Envie/receba os vertices
 MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
  buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
 rounds++; // Incrementa no. de rodadas de comunicacao
// Passo 8.3.11 Libere o espaco alocado aos buffers
 free(buf_send);
 free(sbuf);
 free(send_displ);
 free(rbuf);
 free(rec_displ);
// Passo 8.4 Elimine os vertices com grau < 2
 for( i = 0; i < tam_g; i++ )
   for( j = 0; j < tamR; j += 2 )
     if(Grafo[i].u==buf_rec[j]&&buf_rec[j+1]>=2
        && Grafo[i].bp == 0) {// Se vert. possui grau >= 2
           Grafo[i].bp = 1;// Pertence ao grafo bipartido
           tam_b++;
     } // fim if
// Determine a arvore geradora do grafo auxiliar
// Passo 9. Move todas as arestas para root
```

```
// Passo 9.1 Dimensione o vetor de envio buf_send
   buf_send = ( unsigned long int *) malloc(4*tam_b *
                sizeof(unsigned long int));
// Passo 9.2 Prepare o vetor de envio
 k = 0;
 for(  i = 0; i < tam_g; i++ )
   if( Grafo[i].bp  ) {
       buf_send[k++] = Grafo[i].u;
       buf_send[k++] = Grafo[i].v[0];
       buf_send[k++] = Grafo[i].v[1];
       buf_send[k++] = rank;
    } // fim if
// Passo 9.3. Dimensione o vetor rbuf
 if( rank == root )
   rbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
// Passo 9.4. Envie as quantidades de arestas
 MPI_Gather(&tam_b,1,MPI_UNSIGNED_LONG,rbuf,1,MPI_UNSIGNED_LONG,
             root, MPI_COMM_WORLD);
 rounds++; // Incrementa no. de rodadas de comunicacao
 if( rank == root ) {
// Passo 9.5. Dimensione o vetor rec_displ
   rec_displ = ( unsigned long int *) malloc(size *
                 sizeof(unsigned long int));
// Passo 9.6. Compute no. de arestas a serem recebidos
   rec_displ[0] = 0;
   rbuf[0] = 4*rbuf[0];
   tamR = rbuf[0];
   for (i = 1; i < size; i++) {
     rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
     rbuf[i] = 4*rbuf[i];
     tamR += rbuf[i];
   }// fim for
// Passo 9.7. Dimensione o vetor de Recebimento
   buf_rec = ( unsigned long int *) malloc(tamR *
               sizeof(unsigned long int));
 }// fim if rank == root
// Passo 9.8. Envie as arestas  para a raiz
 MPI_Gatherv(buf_send,4*tam_b,MPI_UNSIGNED_LONG,buf_rec,
  rbuf,rec_displ,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao
```

```
// Passo 9.8.1. Libere o espaco atribuido a buf_send
 free(buf_send);
// Termina tomada de tempo
 MPI_Barrier(MPI_COMM_WORLD);
 finish = MPI_Wtime();
 total += (finish - start);
// Passo 10. Atribua o buf_rec a variavel GrafoAux
 if( rank == root ) {
// Passo 10.1. Dimensione a matriz GrafoAux
   GrafoAux = (Bipart *)malloc((tamR/4)*sizeof(Bipart));
// Passo 10.2. Transfira os dados recebido para GrafoAux
   j = 0;
   for( i = 0; i < tamR; i += 4 ){
     GrafoAux[j].u = buf_rec[i];
     GrafoAux[j].v[0] = buf_rec[i+1];
     GrafoAux[j].v[1] = buf_rec[i+2];
     GrafoAux[j].rank = buf_rec[i+3];
     j++;
   }// fim for
// Passo 10.3. Libere o espaco atribuido a rec_displ, buf_rec
   free(rec_displ);
   free(buf_rec);
// Passo 11. Ordene as arestas do grafo bipartido pelo grau
// Passo 11.1. Dimensione o vetor L e inicialize-o
   L = ( unsigned long int *) malloc(TAMANHO *
         sizeof(unsigned long int));
   for( i = 0; i < TAMANHO; i++ )  L[i] = 0;
// Passo 11.2. Determine o grau de cada vertice
   for( i = 0; i < tamR/4; i++ )
      L[GrafoAux[i].u]++;
// Passo 11.3. Atualize o grau de cada vertice em GrafoAux
   for( i = 0; i < tamR/4; i++ )
      GrafoAux[i].grau = L[GrafoAux[i].u];
// Passo 11.4. Libere o espaco atribuido a L
   free(L);
// Passo 11.5. Ordene as arestas do grafo auxiliar pelo grau
   qsort(GrafoAux, tamR/4, sizeof(Bipart), ComparB);
// Passo 12. Contrua a arvore geradora
// Passo 12.1. Rotular vertices do grafo bipartido 1 a k
   k = 1; // Primeiro rotulo
   GrafoAux[0].rot_u = k++;//Primeiro vertice recebe rotulo 1
```

```
   for( i = 1; i < tamR/4; i++ ){
     if( GrafoAux[i].u == GrafoAux[i-1].u )
        GrafoAux[i].rot_u = GrafoAux[i-1].rot_u;
     else GrafoAux[i].rot_u = k++; // Proximo rotulo k+1
   }// fim for
// Passo 12.2. Rotular os vertices de k a n
   GrafoAux[0].rot_v = k++;//Primeiro vertice recebe rotulo k
   for( i = 1; i < tamR/4; i++ ){
     for( j = 0; j < i; j++ )
       if(GrafoAux[i].v[0] == GrafoAux[j].v[0] &&
          GrafoAux[i].v[1] == GrafoAux[j].v[1] ) {
            GrafoAux[i].rot_v = GrafoAux[j].rot_v;
            break;
       }// fim if
      if( j >= i ) GrafoAux[i].rot_v = k++;//Prox. rotulo
   }// fim for
   nvert = k-1; // Numero de vertices do grafo
// Passo 13. Calcule a arvore geradora do grafo auxiliar
// Passo 13.1. Inicialize a Dados, Fila e Vertices
   for( i = 0; i < TAMMAXA; i++ ) {
      MatrizDadosB[i][0] = 0;
      MatrizDadosB[i][1] = 0;
   } // fim for
   for( i = 0; i < TAMMAXV; i++ ){
      Fila[i] = -1;
      Vertices[i] = i+1;//Rotulo dos vertices de 1 a TAMMAXV
   }
// Passo 13.1.1. Copie as arestas para MatrizDados
   for( i = 0; i < tamR/4; i++ ){
       MatrizDadosB[i][0] = GrafoAux[i].rot_u;
       MatrizDadosB[i][1] = GrafoAux[i].rot_v;
   }
// Passo 13.1.2 Duplique as arestas do grafo auxiliar
//              para eliminar a orientacao
   k = tamR/4;
   for( i = 0; i < tamR/4; i++ ) {
     MatrizDadosB[k][0] = MatrizDadosB[i][1];
     MatrizDadosB[k][1] = MatrizDadosB[i][0];
     k++;
   }
   narest = k; // Numero de arestas do grafo
```

```
// Passo 13.2. Calcule a a Lista de adjacencias
// Passo 13.2.1. Ordene as arestas pela primeira coordenada
   qsort( MatrizDadosB, narest, 2*sizeof(int), ComparE );
// Passo 13.3. Inicializa a Matriz de indices como vazia
   for( i = 0; i < nvert; i++ ) {
      MatrizIndices[i][0] = -1;
      MatrizIndices[i][1] = -1;
   }
// Passo 13.4. Calcule os indices
   j = 0;
   for( i = 0; i < narest; i++ ) {
     if( MatrizDadosB[i][0] == Vertices[j] ) {
        if( MatrizIndices[Vertices[j]-1][0] == -1 )
            MatrizIndices[Vertices[j]-1][0] = i;
     }else {
        i--;
         MatrizIndices[Vertices[j]-1][1] = i;
         j++;
     } // fim if/else
   } // fim for
   MatrizIndices[Vertices[j]-1][1] = i-1;
// Passo 13.5. Calcule a arvore geradora
// Passo 13.5.1. Marque os demais vert. como nao visitados
   for( l = 0; l < nvert; l++ ) {
      Cor[l] = 0;
      Pai[l] = -1;
   }
// Passo 13.5.2. Visite o vertice raiz
   Cor[Vertices[0]-1] = 1;
// Passo 13.6. Inicialize a Fila
   i = 0; // comeco da fila
   j = 0; // fim da fila
   Fila[j] = Vertices[0];
   j++;
// Passo 13.7. Visite os demais vertices
   while( i != j ) {
      vertice = Fila[i];
      for(k = MatrizIndices[vertice-1][0];
          k <= MatrizIndices[vertice-1][1]; k++)
      {
         if( Cor[MatrizDadosB[k][1]-1] == 0 ) {
```

```
            Cor[MatrizDadosB[k][1]-1] = 1;
            Pai[MatrizDadosB[k][1]-1] = vertice-1;
            Fila[j] = MatrizDadosB[k][1];
            j++;
         } // fim if
      } // fim for
      Cor[vertice-1] = 2;
      i++;
   }// fim while
// Passo 13.8.Determine as arestas da arvore geradora
   for( i = 0; i < narest; i++ )
      ArestasArvore[i] = 0;
   for( i = 0; i < narest; i++ ) {
      if(MatrizDadosB[i][0]-1==Pai[MatrizDadosB[i][1]-1])
           ArestasArvore[i] = 1;
   } // fim for
// Passo 14. Atualize GrafoAux
// Passo 14.1. Inicialize o campo st (arvore geradora)
   for( i = 0; i < tamR/4; i++ )
      GrafoAux[i].st = 0;
// Passo 14.2. Marque as arestas da arvore geradora
   for( i = 0; i < narest; i++ ) {
    for( j = 0; j < tamR/4; j++ ) {
     if((MatrizDadosB[i][0] == GrafoAux[j].rot_u &&
         MatrizDadosB[i][1] == GrafoAux[j].rot_v &&
         ArestasArvore[i]) ||
         MatrizDadosB[i][0] == GrafoAux[j].rot_v &&
         MatrizDadosB[i][1] == GrafoAux[j].rot_u &&
         ArestasArvore[i]))
             GrafoAux[j].st = 1;
    } // fim for
   } // fim for
// Passo 14.3. Calcule o grau de cada vert. na arv. geradora
// Passo 14.3.1. Dimensione o vetor L e inicialize-o
   L = ( unsigned long int *) malloc(TAMANHO *
         sizeof(unsigned long int));
   for( i = 0; i < TAMANHO; i++ ) L[i] = 0;
// Passo 14.4. Determine o grau de cada vertice
   for( i = 0; i < tamR/4; i++ )
     if( GrafoAux[i].st ) L[GrafoAux[i].u]++;
// Passo 14.5. Atualize o grau de cada vert.em GrafoAux
```

```
   for( i = 0; i < tamR/4; i++ )
     GrafoAux[i].grau = L[GrafoAux[i].u];
// Passo 14.6. Libere o espaco atribuido a L
   free(L);
// Passo 15. Envie as arestas de volta para as demais tarefas
// Passo 15.1 Prepare o vetor de envio
   buf_send = ( unsigned long int *) malloc( 5*(tamR/4) *
                sizeof(unsigned long int));
// Passo 15.1.1. Dimensione o vetor send_displ
   send_displ = ( unsigned long int *)malloc(size *
                  sizeof(unsigned long int));
// Passo 15.1.2. Compute o no. de arestas a serem enviados
   send_displ[0] = 0;
   rbuf[0] = 5*(rbuf[0]/4);
   for (i = 1; i < size; i++) {
      send_displ[i] = send_displ[i-1] + rbuf[i-1];
      rbuf[i] = 5*(rbuf[i]/4);
   }// fim for
// Passo 15.1.3. Inicialize o vetor de envio
   for( i = 0; i < tamR/4; i++ ) {
    buf_send[send_displ[GrafoAux[i].rank]++]=GrafoAux[i].u;
    buf_send[send_displ[GrafoAux[i].rank]++]=GrafoAux[i].v[0];
    buf_send[send_displ[GrafoAux[i].rank]++]=GrafoAux[i].v[1];
    buf_send[send_displ[GrafoAux[i].rank]++]=GrafoAux[i].st;
    buf_send[send_displ[GrafoAux[i].rank]++]=GrafoAux[i].grau;
   }
// Passo 15.1.4. Libere o espaco atribuido a GrafoAux
   free(GrafoAux);
   free(send_displ);
// Passo 15.2. Monte o vetor de envio
   sbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
   for(i = 0; i < size; i++) sbuf[i] = rbuf[i]/5;
// Passo 15.2.1. Libere o espaco atribuido a rbuf
   free(rbuf);
// Passo 15.3. Calcule send_disp e sbuf
   send_displ = ( unsigned long int *) malloc(size *
                  sizeof(unsigned long int));
   send_displ[0] = 0;
   sbuf[0] = 5*sbuf[0]; // envie u, v[0], v[1], st e grau
   for (i = 1; i < size; i++) {
```

```
     send_displ[i] = send_displ[i-1] + sbuf[i-1];
     sbuf[i] = 5*sbuf[i];
   } // fim for
 }// fim if rank == root (fim parte sequencial)
// Inicia tomada de tempo
 MPI_Barrier(MPI_COMM_WORLD);
 start = MPI_Wtime();
// Passo 16. Envie/recebe para/das as demais tarefas
// Passo 16.1. Dimensione o vetor de recebimento
 buf_rec = ( unsigned long int *) malloc(5*tam_b*
             sizeof(unsigned long int));
// Passo 16.2. Receba as arestas da arvore geradora
 MPI_Scatterv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
  buf_rec,5*tam_b,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 16.3. Libere o espaco atribuido a sbuf e send_displ
 if( rank == root ) {
    free(buf_send);
    free(sbuf);
    free(send_displ);
 }
// Passo 16.4. Atualize as informacoes recebidas em Grafo
 for( i = 0; i < 5*tam_b; i += 5 ) {
   for( j = 0; j < tam_g; j++ ) {
     if( Grafo[j].bp )
       if(buf_rec[i]==Grafo[j].u && buf_rec[i+1]==Grafo[j].v[0]
          && buf_rec[i+2] == Grafo[j].v[1] ) {
             Grafo[j].st = buf_rec[i+3];
             Grafo[j].grau = buf_rec[i+4];
             break;
          }//fim if
     } // fim for j
   }// fim for i
// Passo 16.5. Libere o espaco atribuido rec_buf
 free(buf_rec);
// Passo 17. Calcule o suporte
// Passo 17.1 Elimine os vert. de grau<2 da arvore geradora
 tam_sp = 0;// no. de arestas da arv. geradora com grau>=2
 for( i = 0; i < tam_g; i++ ) {
   if( Grafo[i].grau >= 2 && Grafo[i].st ) {
      Grafo[i].sp = 1;
```

```
      tam_sp++;
   } // fim if
   else Grafo[i].sp = 0;
 }
// Passo 17.2. Verifique o numero de arestas com grau >= 2
// Passo 17.2.1. Obtem no. total de arestas com grau >= 2
 MPI_Allreduce( &tam_sp, &tamS, 1, MPI_UNSIGNED_LONG,
                MPI_SUM, MPI_COMM_WORLD);
 rounds++; // Incrementa o no. de rodadas de comunicacao

 if( tamS <= tam_a ){//arestas cabem em um unico processador
// Passo 18. Enviar SP para a raiz e resolve sequencialmente
// Passo 18.1 Dimensione o vetor de envio buf_send
   buf_send = ( unsigned long int *) malloc(6*tam_sp*
                sizeof(unsigned long int));
// Passo 18.2 Prepare o vetor de envio
   k = 0;
   for(  i = 0; i < tam_g; i++ )
     if( Grafo[i].sp  ) {
       buf_send[k++] = MatrizAresta[Grafo[i].indice][0];
       buf_send[k++] = MatrizAresta[Grafo[i].indice][1];
       buf_send[k++] = MatrizSucessor[Grafo[i].indice][0];
       buf_send[k++] = MatrizSucessor[Grafo[i].indice][1];
       buf_send[k++] = Grafo[i].indice;
       buf_send[k++] = rank;
     } // fim if
// Passo 18.3. Dimensione o vetor rbuf
   if( rank == root )
      rbuf = ( unsigned long int *) malloc(size *
               sizeof(unsigned long int));
// Passo 18.4. Envie as quant. de arestas do grafo bipart.
   MPI_Gather(&tam_sp,1,MPI_UNSIGNED_LONG,rbuf,1,
           MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
   rounds++; //Incrementa no. de rodadas de comunicacao
   if( rank == root ) {
// Passo 18.5. Dimensione o vetor rec_displ
     rec_displ = ( unsigned long int *)malloc(size *
                   sizeof(unsigned long int));
// Passo 18.6. Compute o numero de arestas a serem recebidos
     rec_displ[0] = 0;
     rbuf[0] = 6*rbuf[0];
```

```
     tamR = rbuf[0];
     for (i = 1; i < size; i++) {
        rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
        rbuf[i] = 6*rbuf[i];
        tamR += rbuf[i];
     }// fim for
// Passo 18.7. Dimensione o vetor de Recebimento
     buf_rec = ( unsigned long int *) malloc(tamR *
                 sizeof(unsigned long int));
   }// fim if rank == root
// Passo 18.8. Envie as arestas  para a raiz
   MPI_Gatherv(buf_send,6*tam_sp,MPI_UNSIGNED_LONG,buf_rec,
      rbuf,rec_displ,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
   rounds++; // Incrementa o no. de rodadas de comunicacao
//Passo 18.9. Libere o espaco atribuido a buf_send
   free(buf_send);
// Passo 18.10. Resolve stitch sequencialmente
   if( rank == root ) {
// Passo 18.10.1 Dimensione o vetor Aux
     Aux = (Stitch *)malloc((tamR/6)*sizeof(Stitch));
// Passo 18.11. Inicializa Aux com dados recebidos
     j = 0;
     for( i = 0; i < tamR; i += 6 ) {
        Aux[j].aresta[0] = buf_rec[i];
        Aux[j].aresta[1] = buf_rec[i+1];
        Aux[j].suc[0] = buf_rec[i+2];
        Aux[j].suc[1] = buf_rec[i+3];
        Aux[j].indice = buf_rec[i+4];
        Aux[j].rank = buf_rec[i+5];
        j++;
     } // fim for i
// Passo 18.12. Libere espaco atribuido a rec_displ e buf_rec
     free(rec_displ);
     free(buf_rec);
// Passo 18.13. Ordene as aresta pelo vertice destino
     qsort(Aux, tamR/6, sizeof(Stitch), ComparS);
// Passo 18.14. Realize a operacao de costura
     j = 0; // inicio da sublista
     for( i = 0; i < (tamR/6)-1; i++ ) {
        if( i == j ) {
           k = Aux[i].suc[0];
```

```
            l = Aux[i].suc[1];
          } // se a proxima aresta pertence a mesma sublita
          if( Aux[i].aresta[1] == Aux[i+1].aresta[1] ) {
             Aux[i].suc[0] = Aux[i+1].suc[0]; // Troca o sucessor
             Aux[i].suc[1] = Aux[i+1].suc[1];
           } else { // Nova sublista
             Aux[i].suc[0] = k;
             Aux[i].suc[1] = l;
             j = i+1;    // Inicio da nova sublista
           } // fim if-else
        } // fim for
        Aux[i].suc[0] = k;
        Aux[i].suc[1] = l;
// Passo 18.15. Envie as arestas costuradas de volta
// Passo 18.15.1. Monte o vetor de envio
        sbuf = ( unsigned long int *) malloc(size*
                 sizeof(unsigned long int));
        for( i = 0; i < size; i++ ) sbuf[i] = rbuf[i]/6;
// Passo 18.15.2. Libere o espaco atribuido a rbuf
        free(rbuf);
// Passo 18.15.3. Dimensione os vetores send_displ e P
        send_displ = ( unsigned long int *) malloc(size*
                       sizeof(unsigned long int));
        L = ( unsigned long int *) malloc(size *
              sizeof(unsigned long int));
// Passo 18.15.4. Calcule send_disp e sbuf
        send_displ[0] = 0;
        L[0] = 0;
        tamS = sbuf[0] = 3*sbuf[0];
        for (i = 1; i < size; i++) {
          send_displ[i] = send_displ[i-1] + sbuf[i-1];
          L[i] = send_displ[i];
          sbuf[i] = 3*sbuf[i];
          tamS += sbuf[i];
        } // fim for
// Passo 18.15.5. Dimensione o vetor de envio
        buf_send = ( unsigned long int *) malloc(tamS*
                     sizeof(unsigned long int));
// Passo 18.15.6. Aloque as informacoes em buf_send
        for( i = 0; i < tamR/6; i++ ) {
          buf_send[L[Aux[i].rank]++] = Aux[i].indice;
```

```
        buf_send[L[Aux[i].rank]++] = Aux[i].suc[0];
        buf_send[L[Aux[i].rank]++] = Aux[i].suc[1];
      } // fim for
// Passo 18.15.7.Libere o espaco atribuido a Aux e L
      free(Aux);
      free(L);
   }// fim rank == root
// Passo 19. Envie/recebe para/das as demais tarefas
// Passo 19.1. Dimensione o vetor de recebimento
   buf_rec = ( unsigned long int *) malloc(3*tam_sp*
               sizeof(unsigned long int));
// Passo 19.2. Receba as arestas da arvore geradora
   MPI_Scatterv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
    buf_rec,3*tam_sp,MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);
   rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 19.3. Libere o espaco atribuido a sbuf e send_displ
   if( rank == root ) {
      free(buf_send);
      free(sbuf);
      free(send_displ);
   }
// Passo 19.4. Atualize o circuito de Euler
   for( i = 0; i < 3*tam_sp; i += 3 ) {
     MatrizSucessor[buf_rec[i]][0] = buf_rec[i+1];
     MatrizSucessor[buf_rec[i]][1] = buf_rec[i+2];
   }
// Passo 19.5. Libere o espaco atribuido a buf_rec
   free(buf_rec);
 } // fim if tamS <= tam
 else { // Arestas nao cabem em um unico processador
// Passo 20. Ordenar as arestas de SP
// Passo 20.1. Dimensionar o vetor Aux
   Aux = (Stitch *)malloc((tam_sp+1)*sizeof(Stitch));
// Passo 20.2 Inicialize Aux com arestas a serem costuradas
   j = 0;
   for(  i = 0; i < tam_g; i++ )
     if( Grafo[i].sp  ) {
       Aux[j].aresta[0] = MatrizAresta[Grafo[i].indice][0];
       Aux[j].aresta[1] = MatrizAresta[Grafo[i].indice][1];
       Aux[j].suc[0] = MatrizSucessor[Grafo[i].indice][0];
       Aux[j].suc[1] = MatrizSucessor[Grafo[i].indice][1];
```

```
       Aux[j].indice = Grafo[i].indice;
       Aux[j].rank = rank;
       j++;
     } // fim if
// Passo 20.3. Ordene as arestas em SP
   qsort(Aux, tam_sp, sizeof(Stitch), ComparS);
// Passo 20.4. Calcular o inicio e o fim da cada sublista
// Passo 20.4.1. Dimensionar o vetor Sub
   Sub = (Lista *) malloc((tam_sp+1)*sizeof(Lista));
// Passo 20.4.2. Encontrar ini/fim das subl. e realizar costura
   h = 0;
   Sub[0].v = Aux[0].aresta[1]; // Vert. da primeira sublista
   Sub[0].ini = 0; // Inicio da primeira sublista
   j = 0; // inicio da sublista
   for( i = 0; i < tam_sp-1; i++ ) {
     if( i == j ) {
        k = Aux[i].suc[0];
        l = Aux[i].suc[1];
        Sub[h].ini = i; // Inicio da nova sublista
        Sub[h].v = Aux[i].aresta[1];//Vert. da nova sublista
      }
      if( Aux[i].aresta[1] == Aux[i+1].aresta[1] ) {
         Aux[i].suc[0] = Aux[i+1].suc[0];//Troca o sucessor
         Aux[i].suc[1] = Aux[i+1].suc[1];
      }else { // Nova sublista
         Aux[i].suc[0] = k;
         Aux[i].suc[1] = l;
         j = i+1; // Inicio da nova sublista
         Sub[h++].fim = i; // Fim da sublista anterior
      } // fim if-else
   } // fim for
   if( tam_sp ){//se a tarefa possui alguma aresta de sp
      Aux[i].suc[0] = k;
      Aux[i].suc[1] = l;
      Sub[h].fim = i;// Fim da ultima sublista
   }
   h++; // h eh o Numero de sublistas
// Passo 20.5. Envie inicio/fim das sublistas para tarefa
//           que armazena o vertice correspondente
// Passo 20.5.1. Dimensione o tamanho de sbuf e inicialize
   sbuf = ( unsigned long int *) malloc(size *
```

```
            sizeof(unsigned long int));
   for( i = 0; i < size; i++ ) sbuf[i] = 0;
// Passo 20.5.2. Descobre quantos itens serao enviados
   for( i = 0; i < h; i++ )
     sbuf[Sub[i].v/tam_v]++;
// Passo 20.5.3. Dimensione os vetores send_displ e P
   send_displ = ( unsigned long int *)malloc(size *
                  sizeof(unsigned long int));
   L = ( unsigned long int *) malloc(size *
         sizeof(unsigned long int));
// Passo 20.5.4. Calcule send_disp e sbuf
   send_displ[0] = 0;
   L[0] = 0;
   tamS = sbuf[0] = 4*sbuf[0];
   for( i = 1; i < size; i++ ) {
     send_displ[i] = send_displ[i-1] + sbuf[i-1];
     L[i] = send_displ[i];
     sbuf[i] = 4*sbuf[i];
     tamS += sbuf[i];
   } // fim for
// Passo 20.5.5. Dimensione o vetor de envio
   buf_send = ( unsigned long int *) malloc(tamS *
                sizeof(unsigned long int));
// Passo 20.6. Coloque  as informacoes em buf_send
   for( i = 0; i < h; i++ ) {
     buf_send[L[Sub[i].v/tam_v]++] = rank;
     buf_send[L[Sub[i].v/tam_v]++] = Sub[i].v;
     buf_send[L[Sub[i].v/tam_v]++] = Sub[i].ini;
     buf_send[L[Sub[i].v/tam_v]++] = Sub[i].fim;
   } // fim for
// Passo 20.7.  Dimensione o vetor de recebimento
   rbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
// Passo 20.8. Envie/receba o numero de vertices
   MPI_Alltoall(sbuf,1,MPI_UNSIGNED_LONG,rbuf,1,
                MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
   rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 20.9. Dimensione o vetor de armazenamento
   rec_displ = ( unsigned long int *) malloc(size *
                 sizeof(unsigned long int));
// Passo 20.10. Compute as posicoes de recebimento
```

```
   rec_displ[0] = 0;
   tamR = rbuf[0];
   for( i = 1; i < size; i++ ) {
      rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
      tamR += rbuf[i];
   } // fim for i
// Passo 20.11. Dimensione o vetor de armazenamento buf_rec
   buf_rec = ( unsigned long int *) malloc(tamR *
               sizeof(unsigned long int));
// Passo 20.12. Envie/receba inicio/fim das sublistas
   MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
    buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
   rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 20.13. Libere espaco alocado para buffers auxiliares
   free(L);
   free(sbuf);
   free(send_displ);
   free(buf_send);
// Passo 20.14. Para cada vert. calcular o inicio/fim global
//              de cada sublista
// Passo 20.14.1. Dimensione os vetores e inicialize-os
   inicio = ( unsigned long int *)malloc( tam_v*
              sizeof(unsigned long int));
   fim = ( unsigned long int *) malloc(tam_v *
           sizeof(unsigned long int));
   for( i = 0; i < tam_v; i++ ) {
      inicio[i] = rank+1;
      fim[i] = 0;
   } // fim for
// Passo 20.14.2.Determine o ini/fim global de cada subl.
   for( i = 0; i < tamR; i += 4 ) {
     if( inicio[buf_rec[i+1]\%tam_v] > buf_rec[i] )
       inicio[buf_rec[i+1]\%tam_v] = buf_rec[i];
     if( fim[buf_rec[i+1]\%tam_v] < buf_rec[i] )
       fim[buf_rec[i+1]\%tam_v] = buf_rec[i];
   } // fim for
// Passo 20.14.3. Dimensione o vetor L e inicialize-o
   L = ( unsigned long int *) malloc(tam_v *
         sizeof(unsigned long int));
   for( i = 0; i < tam_v; i++ )  L[i] = 0;
// Passo 20.15 Determina quais processadores possuem
```

```
//             parte de cada sublista
   for( i = 0; i < tamR; i += 4 )
     Tarefas[buf_rec[i+1]%tam_v][L[buf_rec[i+1]%tam_v]++] =
     buf_rec[i];
// Passo 20.16. Determina os vizinhos esq/dir de cada tarefa
   for( i = 0; i < tamR; i += 4 ) {
     for( j = 0; j < L[buf_rec[i+1]%tam_v]; j++ ) {
       if( buf_rec[i] == Tarefas[buf_rec[i+1]%tam_v][j] ) {
          if( buf_rec[i] == inicio[buf_rec[i+1]%tam_v] )
             buf_rec[i+2] = fim[buf_rec[i+1]%tam_v];
          else if( (j-1) >= 0 ) {
            buf_rec[i+2] = Tarefas[buf_rec[i+1]%tam_v][j-1];
            if( buf_rec[i] == fim[buf_rec[i+1]%tam_v] )
              buf_rec[i+3] = inicio[buf_rec[i+1]%tam_v];
            else if((j+1) < L[buf_rec[i+1]%tam_v])
              buf_rec[i+3] = Tarefas[buf_rec[i+1]%tam_v][j+1];
          }
       }// fim if
     } // fim for j
   } // fim for i
// Passo 20.17 Libere o espaco atribuido para L, inicio e fim
   free(L);
   free(inicio);
   free(fim);
// Passo 20.18. Dimensione os vetores de envio
   buf_send =( unsigned long int*) malloc(tamR *
               sizeof(unsigned long int));
   for( i = 0; i < tamR; i++ ) buf_send[i] = buf_rec[i];
   sbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
   send_displ = ( unsigned long int *) malloc(size *
                  sizeof(unsigned long int));
   for( i = 0; i < size; i++) {
     sbuf[i] = rbuf[i];
     send_displ[i] = rec_displ[i];
   }
// Passo 20.18.2. Libere espaco alocado aos buffers aux.
   free(rbuf);
   free(rec_displ);
   free(buf_rec);
// Passo 20.19.  Dimensione o vetor de recebimento
```

```
   rbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
// Passo 20.20. Envie/receba o no. de vertices
   MPI_Alltoall(sbuf,1,MPI_UNSIGNED_LONG,rbuf,1,
                MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
   rounds++; // Incrementa no. de rodadas de comunicacao
// Passo 20.21. Dimensione o vetor de armazenamento
   rec_displ = ( unsigned long int *) malloc(size *
                sizeof(unsigned long int));
// Passo 20.22. Compute as posicoes de recebimento
   rec_displ[0] = 0;
   tamR = rbuf[0];
   for( i = 1; i < size; i++) {
     rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
     tamR += rbuf[i];
   } // fim for i
// Passo 20.23. Dimensione o vetor de armazenamento
   buf_rec = ( unsigned long int *) malloc(tamR *
              sizeof(unsigned long int));
// Passo 20.24. Envie/receba viz. esq./dir. das sublistas
   MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
    buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
   rounds++; // Incrementa o no. de rodadas de comunicacao
// Passo 20.25. Libere espaco alocado para os buffers
   free(sbuf);
   free(send_displ);
   free(buf_send);
// Passo 21.  Atualize as costuras do ini/fim das sublistas
// Passo 21.1 Verifique quais sublistas nao comecam/terminam
//            na tarefa
   inicio = ( unsigned long int *) malloc( (4 * h)*
              sizeof(unsigned long int));
   l = 0;
   for( i = 0; i < tamR; i += 4 ) {
     if( buf_rec[i+2] != rank ) {
      for( j = 0; j < h; j++ )
        if( Sub[j].v == buf_rec[i+1] ) {
          inicio[l++] = buf_rec[i+2];
          inicio[l++] = Sub[j].v;
          inicio[l++] = Aux[Sub[j].fim].suc[0];
          inicio[l++] = Aux[Sub[j].fim].suc[1];
```

```
        } // fim if
      } // fim if
   }  // l eh o numero de costuras em outras tarefas
// Passo 21.2. Libere espaco alocado para buffers
   free(rbuf);
   free(rec_displ);
   free(buf_rec);
// Passo 21.3. Envie costuras das sublistas para tarefa
//             que armazena o vertice correspondente
// Passo 21.3.1. Dimensione o tamanho de sbuf e inicialize
   sbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
   for( i = 0; i < size; i++ ) sbuf[i] = 0;
// Passo 21.3.2. Descobre quantos itens serao enviados
   for( i = 0; i < l; i += 4 )
     sbuf[inicio[i]]++;//Incrementa no. de sublistas do proc.
// Passo 21.3.3. Dimensione os vetores send_displ e L
   send_displ = ( unsigned long int *) malloc(size*
                  sizeof(unsigned long int));
   L = ( unsigned long int *) malloc(size *
         sizeof(unsigned long int));
// Passo 21.3.4. Calcule send_disp e sbuf
   send_displ[0] = 0;
   L[0] = 0;
   tamS = sbuf[0] = 3*sbuf[0];
   for( i = 1; i < size; i++ ) {
      send_displ[i] = send_displ[i-1] + sbuf[i-1];
      L[i] = send_displ[i];
      sbuf[i] = 3*sbuf[i];
      tamS += sbuf[i];
   } // fim for
// Passo 21.3.5. Dimensione o vetor de envio
   buf_send = (unsigned long int *) malloc( (tamS + 1)*
               sizeof(unsigned long int));
// Passo 21.3.6. Coloque  as inf. de Sub em buf_send
   for( i = 0; i < l; i += 4 ) {
      buf_send[L[inicio[i]]++] = inicio[i+1]; // vertice
      buf_send[L[inicio[i]]++] = inicio[i+2]; // suc[0]
      buf_send[L[inicio[i]]++] = inicio[i+3]; // suc[1]
   } // fim for
// Passo 21.3.8. Libere o espaco atribuido para L e inicio
```

```
   free(L);
   free(inicio);
// Passo 21.4.  Dimensione o vetor de recebimento
   rbuf = (unsigned long int *) malloc(size *
           sizeof(unsigned long int));
// Passo 21.5. Envie/receba o numero de costuras
   MPI_Alltoall(sbuf,1,MPI_UNSIGNED_LONG,rbuf,1,
                MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
   rounds++; // Incrementa no. de rodadas de comunicacao
// Passo 21.6. Dimensione o vetor de armazenamento
   rec_displ = ( unsigned long int *) malloc(size *
                 sizeof(unsigned long int));
// Passo 21.7. Compute as posicoes de recebimento
   rec_displ[0] = 0;
   tamR = rbuf[0];
   for (i = 1; i < size; i++) {
     rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
     tamR += rbuf[i];
   } // fim for i
// Passo 21.8. Dimensione o vetor de armazenamento
   buf_rec = (unsigned long int *) malloc((tamR + 1)*
              sizeof(unsigned long int));
// Passo 21.9. Envie/receba costuras das sublistas
   MPI_Alltoallv(buf_send,sbuf,send_displ,MPI_UNSIGNED_LONG,
    buf_rec,rbuf,rec_displ,MPI_UNSIGNED_LONG,MPI_COMM_WORLD);
   rounds++; // Incrementa no. de rodadas de comunicacao
// Passo 21.9.1. Libere espaco alocado para buffers
   free(sbuf);
   free(send_displ);
   free(buf_send);
// Passo 21.10. Realize as costuras necessarias
   for( i = 0; i < tamR; i += 3 )
     for( j = 0; j < h; j++ ) {
       if( buf_rec[i] == Sub[j].v ){//vertice a ser costurado
         Aux[Sub[j].fim].suc[0] = buf_rec[i+1];
         Aux[Sub[j].fim].suc[1] = buf_rec[i+2];
       }// fim if
     }// fim for i
// Passo 21.10.1. Libere espaco alocado aos buffers
   free(Sub);
   free(buf_rec);
```

```
   free(rbuf);
   free(rec_displ);
// Passo 21.11. Atualize as costuras em Edge e Suc
   for( i = 0; i < tam_sp; i++ ) {
     MatrizSucessor[Aux[i].indice][0] = Aux[i].suc[0];
     MatrizSucessor[Aux[i].indice][1] = Aux[i].suc[1];
   }//fim for
// Passo 21.12.Libere o espaco atribuido para Aux
   free(Aux);
 } // fim else tamS <= tam
// Passo 22. Libere o espaco atribuido a Vert e Grafo
 free(Vert);
 free(Grafo);
 // Final da tomada de tempo
 MPI_Barrier(MPI_COMM_WORLD);
 finish = MPI_Wtime();
 total += (finish - start);
 printf("tempo processador %d: %lf", rank, total);
// Passo 23 Envio os tempos para o processador raiz
// Passo 23.1 Dimensione o vetor rbuf
 if( rank == root )
   buff = (double *)malloc(size*sizeof(double));
// Passo 23.2 Envie o tempo de execucao para a raiz
 MPI_Gather(&total,1,MPI_DOUBLE,buff,1,MPI_DOUBLE,
            root,MPI_COMM_WORLD);
 if( rank == root ) {
// Passo 23.3 Armazene o tempo de execucao em arquivo
// Passo 23.3.1 Abre o arquivo
   Arq = fopen("TemposEuler.txt", "a");
// Passo 23.3.2 Escreve no arquivo
   fprintf(Arq,"Numero de ciclos da particao: %ld", ncyc);
   fprintf(Arq,"Rodadas de comunicacao: %ld", rounds);
   fprintf(Arq,"Vertices:%ld Arestas:%ld",num_ver,num_ares);
   for( i = 0; i < size; i++ ) {
     fprintf(Arq,"Tarefa: %d  ", i);
     fprintf(Arq,"Tempo: %lf ", buff[i]);
   }
// Passo 23.3.3 Feche o arquivo
   fclose(Arq);
// Passo 23.4 Libere o espaco alocado para buff
   free(buff);
```

```
 } // fim if
// Passo 24. Envio o circuito de Euler para a raiz
// Passo 24.1. Dimensione o vetor rbuf
 if( rank == root )
   rbuf = ( unsigned long int *) malloc(size *
            sizeof(unsigned long int));
// Passo 24.2. Envie as quant. de arestas local do grafo
 MPI_Gather(&tam_din2,1,MPI_UNSIGNED_LONG,rbuf,1,
            MPI_UNSIGNED_LONG,root,MPI_COMM_WORLD);

 if( rank == root ) {
// Passo 24.3. Dimensione o vetor rec_displ
   rec_displ = ( unsigned long int *) malloc(size *
                 sizeof(unsigned long int));
// Passo 24.4. Compute o no. arestas a serem recebidos
   rec_displ[0] = 0;
   rbuf[0] = 2*rbuf[0];
   for( i = 1; i < size; i++ ) {
     rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
     rbuf[i] = 2*rbuf[i];
   } // fim for
 }// fim if rank == root
// Passo 24.5. Envie as arestas  para a raiz
 MPI_Gatherv(MatrizSucessor,2*tam_din2,MPI_UNSIGNED_LONG,
  MatrizSuc,rbuf, rec_displ,MPI_UNSIGNED_LONG,
  root,MPI_COMM_WORLD);
// Passo 24.6. Libere o espacao atribuido a rbuf e rec_displ
 if( rank == root ) {
    free(rbuf);
    free(rec_displ);
// Passo 24.7. Armazene o resultado em arquivo
// Passo 24.7.1. Armazene o resultado em arquivo
// Passo 24.7.1.1. Abre o arquivo
   Arq = fopen("grafo16_r.txt", "w");
// Passo 24.7.2.1. Escreve no arquivo
   for( i = 0; i < num_ares; i++ ) {
     fprintf(Arq,"(%ld, ", MatrizEdge[i][0]);
     fprintf(Arq,"%ld) ", MatrizEdge[i][1]);
   } // fim fo
   fprintf(Arq, "\n");
   for( i = 0; i < num_ares; i++ ) {
```

```
     fprintf(Arq,"(%ld, ", MatrizSuc[i][0]);
     fprintf(Arq,"%ld) ", MatrizSuc[i][1]);
   } // fim for
// Passo 24.7.3. Feche o arquivo
   fclose(Arq);
 } // fim if
// Passo 25. Finalize o MPI
 MPI_Finalize();
 return 0;
} // fim funcao main

/*-------------------------------------------------------[]
 Funcao CompartBipart:Compara duas arestas do grafo bipartido
                      quanto a suas arestas
[]-------------------------------------------------------*/

int ComparBipart(const void *a, const void *b) {
  Bipart *a1 = (Bipart*)a;
  Bipart *a2 = (Bipart*)b;
  int resp = 0;
  if( a1->u < a2->u )
     resp = -1;
  else if( a1->u > a2->u )
     resp = 1;
  else {
    if( a1->v[0] < a2->v[0] ) resp = -1;
    else if( a1->v[0] > a2->v[0] ) resp = 1;
    else {
      if( a1->v[1] < a2->v[1] ) resp = -1;
      else if( a1->v[1] > a2->v[2] ) resp = 1;
      else resp = 0;
    }
  }
  return( resp ); // Compara duas arestas do grafo bipartido
} // fim funcao ComparBipart

/*-------------------------------------------------------[]
  Funcao ComparB: Compara duas arestas do grafo bipartido
                  quanto ao seus graus
[]-------------------------------------------------------*/
```

```
int ComparB(const void *a, const void *b) {
   Bipart *u = (Bipart*)a;
   Bipart *v = (Bipart*)b;
   return( u->grau > v->grau ? -1 : u->grau != v->grau );
} // fim funcao ComparB

/*-------------------------------------------------------------[]
    ComparaS: Compara duas arestas do suporte
[]-------------------------------------------------------------*/

int ComparS(const void *u, const void *v) {
   Stitch *Lhs = (Stitch*)u;
   Stitch *Rhs = (Stitch*)v;
   int Leftkey_1 = Lhs->aresta[1];
   int Rightkey_1 = Rhs->aresta[1];
   int Leftkey_2 = Lhs->aresta[0];
   int Rightkey_2 = Rhs->aresta[0];
   int Leftkey, Rightkey;
   if (Leftkey_1 != Rightkey_1) {
      Leftkey = Leftkey_1;
      Rightkey = Rightkey_1;}
   else {
      Leftkey = Leftkey_2;
      Rightkey = Rightkey_2; }
   return( Leftkey < Rightkey ? -1 : Leftkey != Rightkey );
} // fim funcao ComparS

/*-------------------------------------------------------------[]
    Funcao ComparE: Compara duas arestas do grafo
[]-------------------------------------------------------------*/

int ComparE( const unsigned long int *Lhs,
             const unsigned long int *Rhs)
{
   int Leftkey_1 = Lhs[0];
   int Rightkey_1 = Rhs[0];
   int Leftkey_2 = Lhs[1];
   int Rightkey_2 = Rhs[1];
   int Leftkey, Rightkey;
   if (Leftkey_1 != Rightkey_1) {
      Leftkey = Leftkey_1;
```

```
      Rightkey = Rightkey_1;}
   else {
      Leftkey = Leftkey_2;
      Rightkey = Rightkey_2; }
   return( Leftkey < Rightkey ? -1 : Leftkey != Rightkey );
} // fim funcao ComparE
```

B.2 O Programa Gerador

```
/*-------------------------------------------------------------[]
Programa: Gerador.c

Programador: Claudia Nasu/Edson Norberto Caceres
Data: 23/08/2002

Dialogo: Este programa gera grafos eulerianos aleatoriamente
[]-------------------------------------------------------------*/
// Declaracao das bibliotecas utilizadas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 1024 // Numero maximo de arestas do grafo

int mat_adj[MAX][MAX];// Matriz de adjacencias do grafo

// Funcao principal
int main( int argc, char *argv[] )
{
  FILE *ArqS;
  int i, j, dif, somal, somac, aux, flag,
      nvertices, narestas;

// Passo 1. Verifique se houve erro nos argumentos
  if( argc != 3 ) {
   printf("Gerador <numero de vertices> <nome do arquivo>");
   exit(0);
  }
  nvertices = atoi(argv[1]);
```

```
// Passo 2. Inicialize a semente do srand
  srand ((unsigned)time(0));
// Passo 3. Gere a matriz de adjacencias do grafo
  for( i = 0; i < nvertices; i++ )
    for( j = 0; j < nvertices; j++ ) {
      if( i == j )
        mat_adj[i][j] = -1; // grafo nao possui lacos
      else if( mat_adj[i][j] != -1 ) {
// Inclui uma aresta no grafo com probabilidade de 20%
         aux = rand()%4;
         if( aux == 1 ) // inclui aresta entre i e j
             mat_adj[i][j] = 1;
         else // nao inclui aresta entre i e j
             mat_adj[i][j] = 0;
      } // fim else-if
    } // fim for j
// Passo 4. Verifique se o grafo gerado eh euleriano
   flag = 1;
   while( flag ) {
     flag = 0;
     for( i = 0; i < nvertices; i++ ) {
       somal = 0;
       somac = 0;
       for( j = 0; j < nvertices; j++ ) {
        if( mat_adj[i][j] > 0 )
           somal += mat_adj[i][j];
        if( mat_adj[j][i] > 0 )
           somac += mat_adj[j][i];
       } // fim for
       if( somal < somac ) {
         // quantas arestas faltam na linha
         dif = somac - somal;
// Procura nas colunas posteriores
// procura um coluna naquela linha para modificar
         for( j = i; dif > 0 && j < nvertices; j++ ) {
           if( mat_adj[i][j] == 0 ) {
             mat_adj[i][j] = 1;
             dif--;
           } // fim if
         } // fim for
// Nao havia colunas posteriores em 0 suficiente
```

```
        if( dif > 0 ) {
// Procura nas colunas anteriores
// procura um coluna naquela linha para modificar
          for( j = 0; dif > 0 && j < i; j++ ) {
           if( mat_adj[i][j] == 0 ) {
              mat_adj[i][j] = 1;
              dif--;
              flag = 1;
           } // fim if
          } // fim for
        } // fim if
// Nao havia colunas ant/post suficientes
        if( dif > 0 ) {
// Elimina arestas nas colunas
          for( j = 0; dif > 0 && j < nvertices; j++ ) {
            if( mat_adj[j][i] == 1 ) {
              mat_adj[j][i] = 0;
              dif--;
              if( j < i ) flag = 1;
            } // fim if
          } // fim for
        } // fim if
      } // fim if
      else if( somac < somal ){
        dif = somal - somac;
// Procura nas linhas posteriores
// procura um linha naquela coluna para modificar
        for( j = i; dif > 0 && j < nvertices; j++ ) {
          if( mat_adj[j][i] == 0 ) {
            mat_adj[j][i] = 1;
            dif--;
          } // fim if
        } // fim for
// Nao havia linhas posteriores suficientes
        if( dif > 0 ) {
// Procura nas linhas anteriores
// procura um linha naquela coluna para modificar
          for( j = 0; dif > 0 && j < i; j++ ) {
            if( mat_adj[j][i] == 0 ) {
              mat_adj[j][i] = 1;
              dif--;
```

```
              flag = 1;
            } // fim if
          } // fim for
        } // fim if
// Nao havia linhas ant/post suficiente
        if( dif > 0 ) {
// Elimina arestas na linha
          for( j = 0; dif > 0 && j < nvertices; j++ ) {
            if( mat_adj[i][j] == 1 ) {
               mat_adj[i][j] = 0;
               dif--;
               if( j < i ) flag = 1;
            } // fim if
          } // fim for
        }// fim if
      } // fim if-else
    } // fim for i
  } // fim while
// Passo 5. Calcule o numero de arestas do grafo
  narestas = 0;
  for( i = 0; i < nvertices; i++ )
     for( j = 0; j < nvertices; j++ )
// Se existe uma aresta entre os vertices i e j
       if( mat_adj[i][j] == 1 )
         narestas++;
// Passo 6. Armazene as arestas do grafo gerado em arquivo
  ArqS = fopen(argv[2], "w");
// Passo 6.1. Ordene pelo vertice destino - Edge
  fprintf(ArqS, "%d \n", nvertices);
  fprintf(ArqS, "%d \n", narestas);
  for( i = 0; i < nvertices; i++ )
    for( j = 0; j < nvertices; j++ )
      if( mat_adj[j][i] == 1 )
        fprintf(ArqS, "%d  %d\n", j, i);
// Passo 6.2. Ordene pelo vertice origem - Sucessor
  for( i = 0; i < nvertices; i++ )
    for( j = 0; j < nvertices; j++ )
      if( mat_adj[i][j] == 1 )
        fprintf(ArqS, "%d  %d\n", i, j);
// Passo 6.3. Fecha o arquivo
     fclose(ArqS);
```

```
}
```

Referências Bibliográficas

[AG94] G. S. Almasi e A. Gottlieb. *Highly Parallel Computing*. The Benjamin-Cummings Publishing Company, 1994.

[Bar96] V. C. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.

[BJvOR99] O. Bonorden, B. Juurlink, I. von Otte, e I. Rieping. The paderborn university BSP (PUB) library - design, implementation and performance. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, Abril 1999.

[BM76] J. A. Bondy e U. S. R. Murty. *Graph Theory with Applications*. North-Holland, 1976.

[CD99] A. Chan e F. Dehne. A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4):533–538, 1999.

[CDF+97] E. N. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, e S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *24th International Colloquium on Automata, Languages, and Programming (ICALP 97). Lecture Notes in Computer Science*, 125:390–400, 1997.

[CDSS92] E. N. Cáceres, N. Deo, S. Sastry, e J. L. Szwarcfiter. On finding euler tours in parallel. *Parallel Processing Letters*, 3(3):223–231, 1992.

[CMS01] E. N. Cáceres, H. Mongelli, e S. W. Song. Algoritmos paralelos usando CGM/PVM/MPI: Uma introdução. *Anais do XXI Congresso da Sociedade Brasileira de Computação, Jornadas de Atualização de Informática*, 2:219–280, 2001.

[CO93] G. Chartrand e O. R. Oellermann. *Applied and Algorithmic Graph Teory*. McGraw-Hill, 1993.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[CV88] R. Cole e U. Vishkin. Approximate parallel scheduling. part i: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17(1), 1988.

[Deh99] F. Dehne. Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[DFC+02] F. Dehne, A. Ferreira, E. N. Cáceres, S. W. Song, e A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002.

[DFRC93] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *Proceedings of the 9th ACM Symposium on Computational Geometry*, páginas 298–307, 1993.

[Fer99] A. Ferreira. Discrete computing with PC clusters: an algorithmc approach. *International School on Advanced Algorithmic Techniques for Parallel Computation with applications*, Setembro 1999.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, e U. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[GLS99] W. Gropp, E. Lusk, e A. Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, segunda edição, 1999.

[Goo96] M. T. Goodrich. Communication efficient parallel sorting. *Proceeding of the 28th ACM Symposium on Theory of Computing*, páginas 247–256, 1996.

[Goo99] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM J. COMPUT.*, 29(2):416–432, 1999.

[Göt98] S. Götz. *Communication-Efficient Parallel Algoritms for Minimum Spanning-Tree Computation*. Tese de Doutoramento, Universitat-Gesamthochschule Parderborn, Maio 1998.

[GR88] A. Gibbons e W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[GV94] A. V. Gerbessiotis e L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, páginas 251–267, 1994.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[LG99] I. G. Lassous e J. Gustedt. List ranking on a coarse grained multiprocessor. Relatório Técnico 3640, Institut National de Recherche en Informatique et en Automatique, Março 1999.

[LG00] I. G. Lassous e J. Gustedt. List ranking on PC clusters. Relatório Técnico 3869, Institut National de Recherche en Informatique et en Automatique, Janeiro 2000.

[MT95] H. Mongelli e R. Terada. Algoritmos paralelos para solução de problemas lineares. Relatório Técnico RT-MAC-9506, Instituto de Matemática e Estatística da Universidade de São Paulo, Junho 1995.

[Pac97] P. S. Pacheco. *Parallel Programming with MPI*. Morgan-Kaufmann, 1997.

[Rei93] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan-Kaufmann Publishers, 1993.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, e J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[Son99] S. Song. Parallel graph algorithms. *International School on Advanced Algorithmic Techniques for Parallel Computation with Applications*, Setembro 1999.

[SV82] Y. Shiloch e U. Vishkin. An O(log n) parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[Szw88] J. L. Szwarcfiter. *Grafos e Algoritmos Computacionais*. Editora Campus, 1988.

[TV85] R. E. Tarjan e U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Agosto 1990.

[WA99] B. Wilkinson e M. Allen. *Parallel Programing: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.