Implementação e Avaliação de Algoritmos BSP/CGM para o Fecho Transitivo e Problemas Relacionados

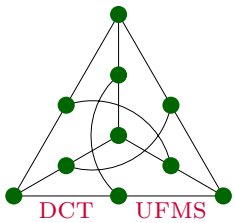
Amaury Antônio de Castro Junior

Dissertação de Mestrado

Orientação: Prof. Dr. Edson Norberto Cáceres

Área de Concentração: Ciência da Computação

Dissertação apresentada ao Departamento de Computação e Estatística (DCT) da Universidade Federal de Mato Grosso do Sul (UFMS) como parte dos requisitos para o 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
Março de 2003

Implementação e Avaliação de Algoritmos BSP/CGM para o Fecho Transitivo e Problemas Relacionados

Este exemplar corresponde à redação final da dissertação de mestrado devidamente corrigida e defendida por Amaury Antônio de Castro Junior e aprovada pela comissão julgadora.

Campo Grande/MS, 07 de abril de 2003.

Banca Examinadora:

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

"O degrau de uma escada não serve simplesmente para que alguém permaneça em cima dele, destina-se a sustentar o pé de um homem pelo tempo suficiente para que ele coloque o outro um pouco mais alto."(Thomas Huxley)

Aos meus pais, Amaury e Teresa.

Agradecimentos

Não há uma só pessoa no mundo que possa dizer que todos os obstáculos que tenha transposto ou as vitórias que tenha alcançado em sua vida foram obtidas por um esforço individual e solitário. É fato que tudo o que produzimos, não fazemos sozinhos. E, no meu caso, recebi apoio e incentivo de tantas pessoas amigas que será muito difícil expressar, em tão poucas palavras, toda a minha gratidão. Por isso, peço licença ao meu orientador e a todos os leitores desta dissertação para, de certa forma, "quebrar o protocolo"e me estender um pouco em meus agradecimentos.

Quero primeiramente, agradecer a Deus, pela oportunidade de viver e buscar neste mundo a evolução mental e espiritual. Aos meus pais, Amaury e Teresa, a quem também dedico este trabalho, e aos meus irmãos Katiuscia e André pelas lições de amor e carinho que alicerçam esta e outras conquistas de minha vida. À minha esposa Caroline e ao meu filho Pedro pelas suas presenças que enriquecem e preenchem de sentimentos a minha vida familiar, estimulando o meu crescimento como pessoa. Aos meu padrinhos, Arnaldo e Cida, pela carinho e confiança dirigidos a mim, pelas palavras de força e estímulo e pelos exemplos de fé, religiosidade e caridade. Enfim, muito obrigado a todos os meus parentes e familiares, em especial a Tia Ausdy que teve paciência e disposição de fazer a correção deste texto.

Ao meu orientador Prof. Dr. Edson Norberto Cáceres, acima de tudo pela paciência que teve comigo. Obrigado por me receber em sua casa e pelo seu esforço no sentido de possibilitar que eu chegasse até aqui. Sem a sua ajuda, com certeza isso não teria acontecido!

Aos professores e amigos do DCT/UFMS, com os quais muito aprendi. Uma grande parcela das minhas conquistas devo às oportunidades que me foram dadas por vocês. Um abraço especial ao Prof. Dr. Marcelo Henriques de Carvalho pelo seu esforço em tornar realidade este curso de mestrado, ao Prof. Henrique Mongelli, pelo pelas valiosas dicas, aos amigos Paulo Pagliosa e Luciana, pela companhia e pela carona até São Carlos em uma das viagens que fiz, aos amigos Marcelo Siqueira e Ronaldo, que mesmo distantes não deixaram de enviar conselhos e mensagens de motivação e, finalmente aos

amigos Erik, Gonda e Jona, pela força e pelas palavras de incentivo. Agradeço também aos professores e amigos do CCET/UCDB, pela oportunidade, pelas portas que me foram abertas, pela ajuda e pelo respeito que têm por mim como pessoa e como profissional, em especial aos amigos Hemerson, Mauro, Marco e Ricardo pela força e incentivo durante o desenvolvimento deste trabalho. Aos professores e técnicos do IME-USP, em especial ao Prof. Dr. Siang, com quem tive a oportunidade de conversar durante as minhas "aventuras"por São Paulo. O Prof. Siang foi muito receptivo e atencioso, desde o dia em que o conheci. Se já o admirava como pesquisador, agora, também o admiro como pessoa. Ao IC-UNICAMP por permitir a utilização do *Beowulf* para a execução dos algoritmos e obtenção dos tempos mostrados neste trabalho. Aos amigos da 1a. turma de mestrado, à amiga Cláudia, pela companhia, pelo apoio e pelas valiosas dicas, à amiga Liana, que assim como eu, produziu muito mais que uma dissertação :), ao amigo Celso, que vindo da matemática, demonstrou tanto interesse e esforço que acabou motivando a todos nós, à amiga Raquel, que mesmo distante não deixava de enviar mensagens de carinho e amizade e, finalmente ao amigo Murillo que, quando a DATAPREV deixava, nunca faltava as nossas reuniões de estudo nas tardes de sábado e domingo. Assim como eu, vocês não desistiram diante das dificuldades encontradas. Somos pioneiros e vencedores!

Quero agradecer também aos amigos do "suporte". Aqueles que mesmo distantes, estão sempre por perto. A todos os amigos da turma e da lista GRAD94, ao amigo Said, que me recebeu na sua casa em muitas das minhas viagens a São Paulo e que apesar de baixinho é um grande irmão e um grande exemplo de caráter e amizade para com todos que conhece, ao amigo Fábio Lubacheski, por sofrer junto comigo, torcendo pelo Botafogo e por ser tão pato no xadrez (aliás, Luba, quem é o mestre agora?), aos amigos Leonardo e Medina, pela companhia em uma das minhas viagens para São Paulo, aos amigos Rafael e Juninho por "suportarem"a minha presença em sua casa, aos amigos Fábio Henrique e Valguima, pelo feijão, pelas pizzas e pelo carinho com o qual sempre me receberam em sua casa e também por serem pessoas que muito admiro e respeito, ao clube da Luluzinha, Ana Lúcia, Dani, Paula e Liz, pelos momentos de descontração e oração que me proporcionaram, ao amigo Marcelo Cintra, que mesmo distante não deixava de transmitir fé, serenidade, espiritualidade e tranqüilidade.

Finalmente, peço desculpas a todos os amigos que me acompanharam e me apoiaram durante o desenvolvimento deste trabalho mas que, por descuido, não foram citados. No entanto, certamente, por serem verdadeiros amigos, partilham da minha alegria pela conquista de mais esta vitória.

Amaury Antônio de Castro Junior

Conteúdo

Resumo

Neste trabalho, descrevemos e apresentamos os resultados da implementação de um algoritmo BSP/CGM para o fecho transitivo proposto por Cáceres *et al.* Além disso, apresentamos algumas aplicações deste algoritmo na resolução de problemas relacionados em teoria dos grafos, tais como caminhos mais curtos, busca em profundidade e árvore geradora mínima.

Estes algoritmos foram implementados em C, usando a interface LAM/MPI e executados no *Beowulf* do IC-UNICAMP, contendo 66 processadores. Os resultados obtidos são melhores que os descritos na literatura. Para os problemas relacionados, as implementações que usam a estrutura do algoritmo de Warshall para o fecho transitivo apresentam melhores tempos, quando comparadas a algumas implementações paralelas para os mesmos problemas.

Abstract

In this work we describe and present the results of the implementation of a transitive closure BSP/CGM algorithm proposed by Cáceres *et al.* We introduce some applications of this algorithm in the resolution of related problems in graph theory, such as shortest paths, breadth-first search and minimum spanning tree.

These algorithms were implemented in C, using LAM/MPI interface and were executed on the *Beowulf* of IC-UNICAMP, containing 66 processors. The results obtained are better than described in the literature. For the related problems, the implementations that use the Warshall's algorithm for transitive closure present better times, when compared them some existent parallel implementations or the same problems.

Capítulo 1

Introdução e Preliminares

1.1 Introdução

Atualmente, apesar do avanço da tecnologia e da redução dos custos, as máquinas seqüenciais de Von Neumann possuem um poder computacional limitado pelas características da arquitetura e da tecnologia utilizadas. Além disso, o rápido crescimento e a grande disponibilidade de máquinas e ambientes multiprocessados têm motivado o surgimento de questões sobre o projeto de algoritmos paralelos eficientes e a utilização de novas arquiteturas e técnicas de programação para resolver problemas complexos que processam um grande volume de dados.

No entanto, ao contrário do modelo RAM (*Random Access Machine*), utilizado pelas máquinas de Von Neumann, a computação paralela ainda não encontrou um modelo único e amplamente aceito para a análise e implementação de algoritmos paralelos. Dessa forma, a solução paralela proposta para um dado problema é totalmente dependente do modelo paralelo para o qual foi desenvolvida. O principal modelo teórico de computação paralela, o modelo PRAM[15], foi proposto com o intuito de desempenhar um papel semelhante ao modelo RAM para as máquinas paralelas. O modelo PRAM utiliza uma memória compartilhada e na análise da complexidade dos algoritmos para este modelo, não são considerados os custos de comunicação.

Além do modelo PRAM, ainda existem os modelos de memória distribuída, baseados em arquiteturas específicas, como o anel e o hipercubo que não serão detalhados neste trabalho. Nestes modelos, a comunicação entre os processadores é implementada através de uma rede de interconexão. Dessa forma, um processador pode acessar um dado na memória em tempo constante, somente se o dado estiver disponível em sua memória local. Caso contrário, é necessária a comunicação entre processos através da rede para

obtê-lo. Estes passos de comunicação tomam tempo e devem ser descritos como parte do algoritmo.

Portanto, se quisermos um algoritmo paralelo para um dado modelo de memória distribuída, podemos simular o algoritmo PRAM ou projetar um algoritmo paralelo para a arquitetura específica do modelo. A primeira solução é simples e elegante, mas normalmente não produz algoritmos eficientes. Por outro lado, a segunda solução pode ser mais eficiente, mas pode tornar a tarefa de desenvolvimento e implementação muito complexa.

Após diversas experiências com máquinas e arquiteturas paralelas, os pesquisadores concluíram que, atualmente, a comunicação é o principal gargalo da computação paralela. Baseado nesta observação, os modelos realísticos[8, 10, 34] surgiram como uma opção para o desenvolvimento de algoritmos paralelos para resolver problemas complexos que, além de lidarem com um grande volume de dados, utilizem muita comunicação. Uma classe de problemas que possui estas características é a dos problemas em teoria dos grafos.

Um desses problemas consiste em computar o fecho transitivo de um grafo, descrito na Seção 1.2. Segundo Nuutila[22], a computação eficiente do fecho transitivo de um grafo dirigido é exigida em muitas aplicações como, por exemplo, na análise de fluxo e dependência em grafos que representam sistemas paralelos e distribuídos, na construção de *parsing automata* no projeto de compiladores e como um subproblema importante na avaliação de consultas recursivas em bancos de dados. Diversos algoritmos seqüenciais e paralelos foram propostos para este problema. Nosso estudo concentra-se no algoritmo paralelo proposto por Cáceres *et al*[6], que utiliza o modelo BSP/CGM.

Além disso, Leighton[17] apresenta alguns algoritmos paralelos para modelos de memória distribuída, que utilizam o fecho transitivo como subrotina para a resolução de outros problemas em grafos, entre eles o problema dos componentes conexos, dos caminhos mais curtos, da busca em largura e da árvore geradora mínima. Estas idéias podem ser adaptadas para a implementação destes algoritmos no modelo BSP/CGM, demonstrando, através de resultados empíricos, o uso deste modelo na implementação de algoritmos paralelos para os problemas citados acima.

Nas seções que se seguem, apresentamos alguns conceitos básicos em teoria dos grafos e processamento paralelo necessários para o entendimento dos problemas e técnicas utilizadas no desenvolvimento deste trabalho. Além disso, são brevemente descritos os principais modelos de computação paralela e ambientes de troca de mensagens. Por último, apresentamos, de forma sucinta, como esta dissertação está organizada.

1.2 Grafos

Na literatura, alguns conceitos básicos sobre teoria dos grafos diferem um pouco de um autor para outro. Portanto, aqui estão reunidas as definições adaptadas das referências [7, 14, 32] e que serão importantes para o entendimento do trabalho.

Definição 1 *Um* ***grafo*** $G = (V, E)$ *é um conjunto finito não-vazio* V *de* ***vértices*** *e um conjunto* E *de* ***arestas*** *representadas por um par* $\{v, w\}$ *não-ordenado de vértices distintos. Os vértices* v *e* w *são as* ***extremidades*** *da aresta, sendo denominados* ***adjacentes****. A aresta* $\{v, w\}$ *é dita* ***incidente*** *a ambos os vértices* v *e* w*. Se houver mais de uma aresta ligando o mesmo par de vértices, essas arestas são ditas* ***arestas múltiplas****. Se* E *é representado por pares ordenados de arestas* $\{v, w\}$*, então* G *é dito* ***dirigido*** *e a aresta é dita* ***orientada*** *de* v *a* w*.*

Exceto quando especificado, será utilizada a notação $|V| = n$ e $|E| = m$.

Definição 2 *Um* ***grafo com pesos nas arestas*** *é uma tripla* (V, E, W) *onde* (V, E) *é um grafo e* W *é uma função de* E *em* $\mathbb{Z}^+$*. Para uma aresta* $e \in E$*,* $W(e)$ *é denominado* ***peso*** *de* e*.*

Em alguns problemas em grafos, o peso corresponde aos custos ou aspectos negativos de uma aresta, enquanto em outros eles correspondem às capacidades ou benefícios das arestas. Em algumas situações é permitido o uso de pesos negativos ou valores reais. Nestes casos, deve-se tomar muito cuidado com a escolha do algoritmo, pois a corretude de alguns deles depende da restrição aos valores inteiros e não negativos.

Definição 3 *Um* ***caminho*** *ligando os vértices* v_1 *a* v_k *em* G *é uma seqüência de arestas distintas* $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$*. Dizemos que este caminho contém as arestas* $\{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$ *e os vértices* $v_1, \ldots, v_k$*. O caminho é* ***simples*** *se* $v_1, v_2, \ldots, v_{k-1}, v_k$ *são distintos. O caminho é um* ***ciclo****, ou* ***circuito*** *se* $v_1 = v_k$*. Um grafo que não contém ciclos é dito* ***acíclico****.*

Podemos representar um caminho pela sua seqüência de vértices que o compõe $v_1, v_2, \ldots, v_k$.

Definição 4 *O* ***peso*** *ou* ***comprimento de um caminho*** *é definido como sendo o somatório dos pesos das arestas no caminho. Se o caminho é chamado* P*, denotamos seu peso por* $W(P)$*. Um caminho de um vértice* v *a* w *é dito um* ***caminho mais curto*** *de* v *a* w *se não existe nenhum outro caminho de* v *a* w *com peso menor. Observe que o caminho mais curto não é necessariamente único.*

Definição 5 *Um **subgrafo** $G' = (V', E')$ de um grafo G é um grafo tal que $V' \subseteq V$ e $E' \subseteq E$. G' é **gerador** se $V' = V$.*

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 satisfaça P. Ou seja, S' não está propriamente contido em nenhum subconjunto de S que satisfaça P.

Definição 6 *Um grafo $G = (V, E)$ é **conexo** se existe um caminho entre cada par de vértices de G. Denominam-se **componentes conexos** de um grafo G aos subgrafos maximais de G que sejam conexos.*

Definição 7 *Uma **árvore** T é um grafo conexo e acíclico. Em uma árvore existe um único caminho entre cada par de vértices. Uma **árvore enraizada** (T, r) é uma árvore com um vértice especial r, denominado **raiz**. Dados dois vértices quaisquer, v e w, de uma árvore enraizada (T, r), nós dizemos que v é um **ancestral** de w e w é um **descendente** de v (denotado por $v \xrightarrow{*} w$) se v pertence ao caminho de r a w. Cada vértice é ancestral e descendente de si mesmo. Se $v \xrightarrow{*} w$ e $\{v, w\}$ é uma aresta de T, nós dizemos que v é o **pai** de w e w é um **filho** de v (denotado por $v \rightarrow w$).*

Definição 8 *Quando o subgrafo gerador é uma árvore, ele é dito **árvore geradora** de G. Todo grafo conexo G possui uma árvore geradora.*

Definição 9 *Dado um grafo dirigido $G = (V, E)$, encontrar um **fecho transitivo** de G consiste em construir um grafo $G' = (V, E')$ com a aresta $(i, j) \in E'$ se, e somente se, existir um caminho dirigido do vértice i ao vértice j em G.*

O fecho transitivo G' de um grafo $G = (V, E)$ pode ser visto como uma estrutura de dados capaz de determinar, de forma eficiente, se, dado um par de vértices $x, y \in V$, o vértice y pode ser alcançado a partir de x. Após o pré-processamento para a construção do fecho transitivo, este tipo de consulta pode ser respondido em tempo constante, através de um simples acesso à matriz de adjacências de G'. O fecho transitivo pode ser utilizado também para mapear o problema da propagação de resultados. Por exemplo, considere uma tabela de propagação modelada através de um grafo, onde os seus vértices representam as células da tabela e existe uma aresta da célula i para a célula j se o resultado da célula j depende da célula i. Quando o valor de uma célula qualquer é alterado, todos os valores das células alcançáveis, a partir desta, devem ser atualizados. A identificação destas células é possível pelo fecho do grafo.

Nas próximas duas subseções, descrevemos brevemente as principais estruturas de dados para representação de grafos e as duas principais estratégias de percurso em grafos.

1.2.1 Representação de Grafos

Nesta seção, vamos descrever as duas formas de representação computacional mais utilizadas para problemas que envolvem grafos: a **matriz de adjacências** e a **lista de adjacências**.

Seja $G = (V, E)$ um grafo, dirigido ou não, com $|V| = n$, $|E| = m$ e $V = \{v_1, v_2, \ldots, v_n\}$. G pode ser representado por uma matriz $n \times n$, denominada matriz de adjacências, na qual $A = (a_{ij})$ é definida por:

$$a_{ij} = \begin{cases} 1 \text{ se } v_i v_j \in E \\ 0 \text{ caso contrário} \end{cases}, \text{para } 1 \leq i, j \leq n$$

Observe que a matriz de adjacências de um grafo não dirigido é simétrica. Se $G = (V, E, W)$ é um grafo com peso nas arestas, dirigido ou não, os pesos podem ser armazenados diretamente na matriz de adjacências, com uma simples modificação na definição anterior:

$$a_{ij} = \begin{cases} W(v_i v_j) \text{ se } v_i v_j \in E \\ c \text{ caso contrário} \end{cases}, \text{para } 1 \leq i, j \leq n$$

onde c é uma constante cujo valor dependerá da interpretação dos pesos e do problema a ser resolvido. Se os pesos representam os custos, $c = \infty$ (infinito) ou um outro valor muito alto deve ser escolhido para c, pois assume-se que o custo de percorrer uma aresta que não existe é muito alto. Se os pesos representam possibilidades, uma escolha de $c = 0$ é mais apropriada, visto que não se pode percorrer uma aresta que não existe. Para exemplificar esta representação, observe as Figuras 1.1(a), 1.1(b), 1.2(a) e 1.2(b).

Uma outra forma de representação é a lista de adjacências, que armazena, para cada vértice v, uma lista de vértices adjacentes a v. As informações armazenadas em uma lista de adjacências podem variar de acordo com o problema. No entanto, para a grande maioria dos problemas em grafos, cada nó de uma lista de adjacências possui, pelo menos, um campo *vértice* que contém o índice do vértice e um campo *ponteiro* que armazena o endereço do próximo elemento da lista. Cada nó representa uma aresta de um grafo. Por exemplo, suponha que os vértices do grafo foram numerados da seguinte forma $V = \{1, 2, \ldots, n\}$.

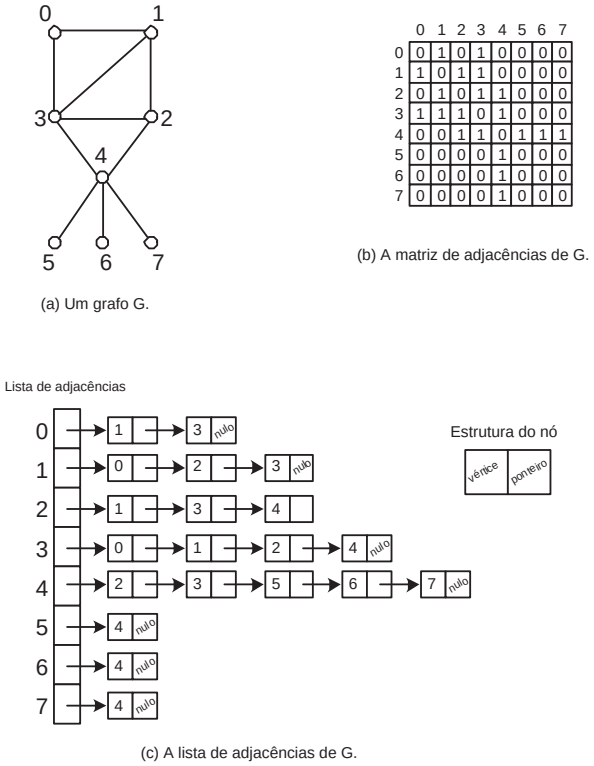


Figura 1.1: Representação de um grafo não dirigido.

Então, se um nó que contém o valor 2 em seu campo *vértice* faz parte da lista de adjacências do 7, ele representa a aresta $(7, 2)$. É utilizado um vetor de ponteiros de tamanho n, sendo que cada posição corresponde a um vértice do grafo e aponta o início da lista de adjacências deste vértice. Esta estrutura de dados para um grafo é exemplificada pela Figura 1.1(c). Observe que cada aresta é representada duas vezes, ou seja, se (v, w) é uma aresta, existe um nó para w na lista de adjacências de v e um nó para v na lista de adjacências de w. No caso de grafos dirigidos, cada aresta é representada exatamente uma vez. Se o grafo, seja ele dirigido ou não, possui peso nas arestas, um campo peso é incluído em cada nó da lista. A Figura 1.2(c) ilustra a estrutura de dados para um grafo dirigido com peso nas arestas.

Para o desenvolvimento deste trabalho, foi escolhida a matriz de adjacências como forma de representação dos grafos. Esta escolha justifica-se pelo fato de que, nos algoritmos estudados e implementados, a verificação de adjacência entre pares de vértices é realizada com muita freqüência. Dessa forma, se utilizássemos uma lista de adjacências, cada verificação exigiria uma busca na lista de adjacências do respectivo vértice, o que levaria a um

tempo de processamento maior. Com a matriz de adjacências, a verificação de adjacência entre pares de vértices é realizada em tempo constante.

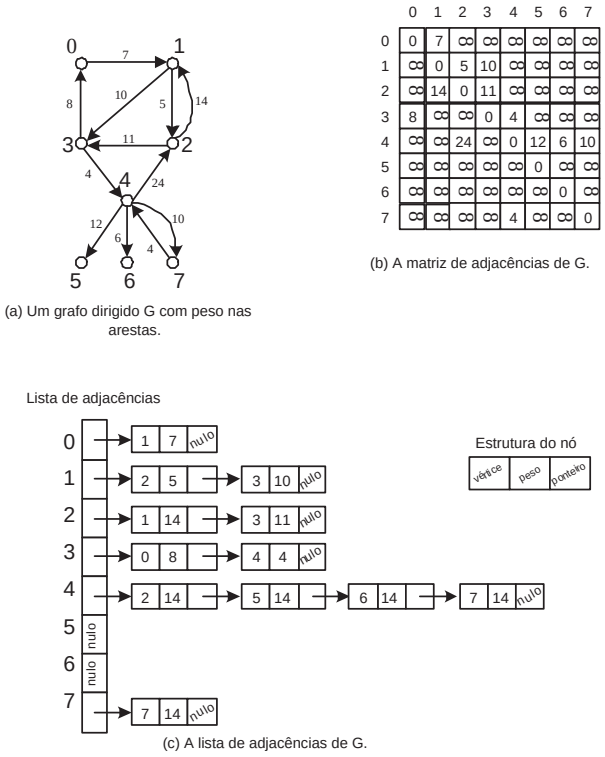


Figura 1.2: Representação de um grafo dirigido.

1.2.2 Percurso em Grafos

A maioria dos algoritmos desenvolvidos para resolver problemas em grafos, dirigidos ou não, percorrem as suas arestas, visitando cada um de seus vértices. Em alguns dos algoritmos considerados adiante, a ordem na qual os vértices são visitados é parte fundamental do método usado para resolver o problema. Algumas estratégias utilizadas no processamento de vértices e arestas produzem métodos amplamente utilizados e eficientes para resolver problemas em grafos. Para finalizar esta seção, vamos descrever duas destas estratégias: a **busca em profundidade** e a **busca em largura**.

A busca em profundidade é uma generalização do percurso em pré-ordem das árvores. O vértice de início pode ser determinado pelo problema ou ser escolhido aleatóriamente. Dessa forma, partindo de v, um caminho é seguido

enquanto for possível, visitando todos os vértices encontrados no caminho. Quando é encontrado um vértice w, tal que todos os seus vizinhos já tenham sido visitados, retornamos sobre as arestas que compõem o caminho encontrado, buscando algum vértice vizinho a um dos que já pertencem ao caminho que ainda não tenha sido visitado. Caso seja encontrado algum vértice, a busca prossegue nesta direção, obedecendo à mesma regra de parada. Esse processo é repetido até que todos os vértices do grafo tenham sido visitados.

Na busca em largura, os vértices são visitados à medida que a distância do vértice inicial v aumenta, onde a distância corresponde ao número de arestas em um caminho mais curto. A idéia da busca em largura é um vértice x a uma distância d de v e percorrer todas as arestas incidentes em x, visitando todos os vértices a uma distância $d+1$ de v, repetindo este processo até que não exista mais nenhum vértice que não tenha sido visitado.

Vale lembrar que, para ambos os métodos descritos, a escolha do próximo vértice a ser visitado é determinada pelos detalhes de implementação, por exemplo, a forma pela qual os vértices são numerados e organizados na representação do grafo G.

1.3 Processamento Paralelo

O processamento paralelo consiste na técnica de divisão de tarefas a serem computadas em partes menores que possam ser executadas em um computador paralelo. Um computador paralelo consiste de um conjunto de processadores que podem trabalhar para resolver um problema computacional cooperativamente. Esta definição é suficientemente abrangente para incluir desde supercomputadores paralelos, que possuem centenas ou milhares de processadores, até um *cluster* de PCs, implementado em um ambiente de rede. O objetivo principal do processamento paralelo é a redução do tempo necessário para a execução de programas. Além disso, podemos destacar alguns objetivos secundários, tais como possibilitar uma maior quantidade de memória primária disponível e resolver problemas mais complexos e que manipulam um grande volume de dados.

1.3.1 Métricas de Desempenho em Sistemas Paralelos

Um sistema paralelo é a combinação do algoritmo com a arquitetura paralela para a qual ele foi desenvolvido. Sabemos que um programa seqüencial é avaliado com base no seu tempo de execução e que esse tempo depende do tamanho de suas entradas. O tempo de execução de um algoritmo paralelo não depende somente de suas entradas, mas também da arquitetura utilizada

e do número de processadores utilizados. As seguintes medidas podem ser utilizadas para a avaliação de sistemas paralelos:

Tempo de Execução (T_p)

Por definição, o tempo de execução seqüencial (T_s) de um programa é o tempo obtido pelo melhor algoritmo seqüencial desde o início até o final da computação em um computador seqüencial. O tempo de execução paralelo (T_p) é o tempo entre o início do processamento até o instante em que o último processador finaliza a execução.

Neste trabalho, consideramos o tempo de execução seqüencial (T_s) como sendo o tempo de execução da implementação paralela sobre um único processador.

Speedup (S)

Na avaliação de um sistema paralelo, geralmente estamos interessados em conhecer o ganho de desempenho (aumento de velocidade) obtido pela paralelização do programa, quando comparado com sua versão seqüencial. O *speedup* (S) é a medida que representa o ganho de se resolver o problema em paralelo.

$$S = \frac{T_s}{T_p}$$

Normalmente, em um sistema com p processadores, o *speedup* é menor do que p, dificilmente atingindo a situação ideal, que seria $S = p$. Um *speedup* acima de p é chamado de ***speedup* superlinear** e raramente é observado.

Eficiência (E)

Como em um sistema paralelo, os processadores não podem gastar 100% do tempo em computação, devido ao tempo gasto com algumas operações intrínsecas à computação paralela, tais como comunicação, sincronização dos processos, entre outras, o *speedup* ideal não é obtido. No entanto, através da medida de eficiência, podemos ter uma idéia da fração do tempo em que os processadores foram efetivamente utilizados. Ela é obtida pela divisão do *speedup* pelo número de processadores.

$$E = \frac{S}{p}$$

Como normalmente o *speedup* é menor que p, a eficiência, nestes casos, fica entre 0 e 1.

Custo (C)

O custo de resolver um problema em paralelo é igual ao produto do tempo de execução paralelo pelo número de processadores utilizados.

$$C = T_p \times P$$

Um dos fatores determinantes no desempenho de algoritmos paralelos se relaciona à minimização dos custos de comunicação entre processadores e, conseqüentemente, ao conceito de **nível de paralelismo** ou **granularidade**. Uma aplicação paralela consiste de diversas tarefas, sendo executadas em processadores distintos. Estas tarefas comunicam-se entre si para garantir o progresso consistente da aplicação. A intensidade da comunicação entre as tarefas de uma dada aplicação paralela estabelece o nível de paralelismo ou granularidade daquela aplicação. Aplicações que demandam intensa comunicação são ditas de **granulariade fina**. As aplicações que requerem pouca comunicação são de **granularidade grossa**.

A granularidade de uma aplicação paralela estabelece requerimentos para a plataforma na qual a aplicação é executada. A plataforma de execução de uma aplicação paralela consiste do número de processadores usados pela aplicação e também do meio físico utilizado para conectar estes processadores. Aplicações paralelas de granularidade fina demandam o uso de plataformas dedicadas, tais como supercomputadores massivamente paralelos, para que o tempo gasto com a comunicação não anule os ganhos decorrentes do paralelismo. Já as aplicações paralelas de granularidade grossa, ao contrário, podem efetivamente utilizar processadores compartilhados e interconectados por redes locais ou de longa distância, executando uma grande carga de processamento.

Atualmente, na área de algoritmos paralelos, muitas pesquisas têm se desenvolvido no sentido de prover soluções que comprovem a utilidade dos modelos de granularidade grossa, conhecidos como realísticos, para a implementação de algoritmos paralelos. Tal atenção deve-se, principalmente, à intensa busca por um modelo de computação paralela, que satisfaça os requisitos de simplicidade, compatibilidade e portabilidade, sendo amplamente aceito para o projeto, a análise e a implementação de algoritmos paralelos. Na seção seguinte, descrevemos alguns dos principais modelos realísticos de computação paralela.

1.4 Modelos Realísticos de Computação Paralela

O principal objetivo do projeto e da implementação de algoritmos paralelos é obter um melhor desempenho com relação à versão seqüencial. No modelo de computação seqüencial de Von Neumann (RAM - *Random Access Machine*) - Máquina de Acesso Aleatório -, que assume a existência de uma única unidade central de processamento e uma memória de acesso aleatório, é possível estabelecer uma relação entre os desempenhos das implementações e dos seus respectivos algoritmos através das medidas de complexidade de tempo baseadas em análises assintóticas. Neste modelo, estas medidas são capazes de refletir corretamente o desempenho dos algoritmos seqüenciais, servindo como referências para as implementações. No entanto, na computação paralela, a mesma relação entre algoritmos e implementações ainda não encontrou um modelo apropriado. Apesar de sua importância conceitual e teórica, o modelo PRAM não consegue capturar com exatidão a noção de paralelismo. Existem diversos fatores que devem ser considerados no projeto de algoritmos paralelos com o intuito de obter o melhor desempenho possível dentro das restrições do problema a ser solucionado. A presença de diversos elementos de processamento torna a definição de um modelo de computação paralela consideravelmente mais complexa. As características não incorporadas ao modelo PRAM durante o desenvolvimento dos algoritmos, tais como custo adicional para referência à memória global e latência, têm grande impacto no desempenho das implementações.

Como já foi dito, a escolha do modelo de computação mais adequado para o projeto de algoritmos paralelos tem sido foco de muita atenção nesta área durante os últimos anos, colocando-nos diante de um conflito inerente. Por um lado, gostaríamos de ter um modelo de alto nível que abstraia os detalhes dos sistemas paralelos para os quais os algoritmos são implementados, tornando os algoritmos projetados para tais modelos simples para descrever e analisar e facilmente portáveis para diversas plataformas. Por outro lado, ignorando detalhes importantes dos sistemas paralelos, a análise dos algoritmos pode não refletir adequadamente o desempenho de suas respectivas implementações. Conseqüentemente, para facilitar a análise de desempenho pode ser essencial o uso de modelos de baixo nível que considerem mais detalhes. Entretanto, algoritmos eficientes para modelos de baixo nível podem ser mais difíceis de projetar e analisar. Além disso, tais algoritmos podem não ser facilmente portáteis de um sistema paralelo para outro.

A busca por um modelo adequado necessita da compreensão e da incorporação, em tal modelo, de características intrínsecas à computação paralela,

bem como de ignorar aquelas características secundárias, superáveis através da tecnologia. Este modelo precisaria, de certa forma, balancear simplicidade com precisão e abstração com praticidade. Como alternativas para a solução destes problemas surgem os chamados *modelos realísticos*, que buscam estabelecer padrões amplamente aceitos que reflitam as dificuldades inerentes do próprio paralelismo.

Goodrich[12] discutiu um modelo clássico de alto nível para o projeto de algoritmos paralelos, o PRAM, e mencionou trabalhos alternativos, sobre modelos de baixo nível, conhecidos como *bridging models* (modelos "ponte"ou realísticos), como o BSP, o LogP e o CGM. As recentes propostas destes últimos modelos motivaram pesquisas teóricas e experimentais para comprovar suas capacidades. Ao mesmo tempo, persiste a busca por modelos alternativos que sejam satisfatórios.

Nesta seção, descrevemos os principais modelos realísticos de computação paralela. Por fim, é realizada uma comparação entre os modelos descritos e justificada a escolha do modelo CGM para a implementação dos algoritmos estudados.

1.4.1 Modelo BSP

O modelo **BSP** (*Bulk Synchronous Parallel*) foi proposto por Valiant [34], em 1990. Além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros. O objetivo principal deste modelo é servir de modelo ponte entre as necessidades de *hardware* e *software* na computação paralela. Segundo Valiant[34], essa ponte é uma das características fundamentais do sucesso do modelo seqüencial de Von Neumann.

O modelo **BSP** consiste de um conjunto de p processadores com memória local, comunicando-se através de algum meio de interconexão, gerenciados por um **roteador** e com facilidades de sincronização global. Um algoritmo BSP consiste numa seqüência de **superpassos** separados por **barreiras de sincronização**, como mostra a Figura 1.3. Um **superpasso** consiste de uma combinação de passos de computação, usando dados disponibilizados localmente no início do superpasso, e passos de comunicação, através de instruções de envio e recebimento de mensagens. Neste modelo uma h**-relação** em um superpasso corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador. Os valores obtidos em resposta a uma mensagem enviada em um superpasso somente poderão ser usados no próximo superpasso.

Os parâmetros do modelo BSP são os seguintes:

- n: tamanho do problema;
- p: número de processadores disponíveis, cada qual com sua memória local;
- L: o tempo mínimo entre dois passos de sincronização. Também chamado de parâmetro de periodicidade ou **latência** de um superpasso;
- g: é a capacidade computacional dividida pela capacidade de comunicação de todo o sistema, ou seja, a razão entre o número de operações de computação realizadas em uma unidade de tempo e o número de operações de envio e recebimento de mensagens. Este parâmetro descreve a taxa de eficiência de computação e comunicação do sistema.

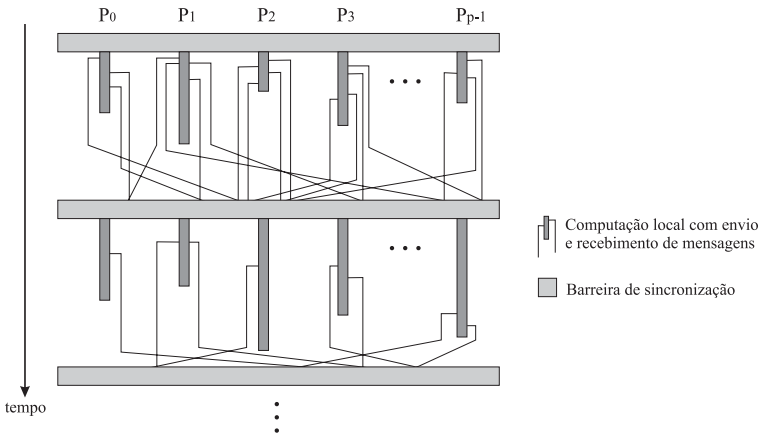


Figura 1.3: O Modelo BSP [13]

Os dois últimos parâmetros, L e g, são utilizados para computar o **custo de comunicação** de um algoritmo BSP. O parâmetro L representa o **custo de sincronização**, de tal forma que cada operação de sincronização contribui com L unidades de tempo para o tempo total de execução. O parâmetro L, também pode ser visto como sendo a **latência** da comunicação, pois os dados recebidos somente podem ser acessados no próximo superpasso. A capacidade de comunicação de uma rede de computadores está relacionada ao parâmetro g. Através deste parâmetro, o tempo gasto pela troca de dados entre os processadores pode ser estimado. Se o número máximo de mensagens enviadas por algum processador durante uma troca simples é h, então seriam necessárias até gh unidades de tempo para a conclusão da troca.

Na prática, o valor de g é determinado empiricamente, para cada máquina paralela, através da execução de *benchmarks* apropriados [16].

Logo, o **tempo total de execução** de um superpasso de um algoritmo BSP é igual a $w_i + gh_i + L$, onde $w_i = \max\{L; t_1; \ldots; t_p\}$ e $h_i = \max\{L; c_1; \ldots; c_p\}$, t_j e c_j são respectivamente, o número de operações de computações executadas e o número de mensagens recebidas e/ou enviadas pelo processador j no superpasso i. O **custo total** de um algoritmo é dado pela soma dos custos de cada um dos superpassos. Considerando T o número de superpassos, seja

$$W = \sum_{i=0}^{T} w_i \text{ e } H = \sum_{i=0}^{T} h_i$$

a soma de todos os valores de w e h de cada superpasso. Então, o **custo total** de um algoritmo BSP é $W + gH + LT$. O valor de W representa o total de computações locais e o valor de H representa o volume total de comunicação.

1.4.2 Modelo LogP

O modelo LogP [8] foi proposto a partir de um esforço de diversos grupos de pesquisadores de áreas teóricas, de hardware e de software, no sentido de produzir um modelo de computação paralela amplamente aceito. O objetivo, semelhante ao do modelo BSP, é o de incorporar os atributos das máquinas reais existentes.

O nome do modelo LogP surgiu do agrupamento das letras que representam os parâmetros considerados por ele. O modelo LogP possui os seguintes parâmetros:

- n: tamanho do problema;
- L: limite superior de latência, ou tempo de espera necessário para o envio de uma mensagem de sua origem até seu destino;
- o: o *overhead*, definido como o intervalo de tempo que um processador permanece comprometido com o envio ou o recebimento de cada mensagem. Durante este período, o processador não pode realizar nenhuma outra operação;
- g: o *gap*, definido como o tempo mínimo entre duas operações de envio de mensagem consecutivas ou duas operações de recebimento de mensagem consecutivas em um mesmo processador;

- p: número de processadores disponíveis, cada qual com sua memória local. Um **ciclo** é definido como uma unidade de tempo para operações locais.

Além disso, este modelo assume que a rede possui uma capacidade limitada, tal que, no máximo, $\lceil \frac{L}{g} \rceil$ mensagens podem estar em trânsito entre quaisquer dois processadores ao mesmo tempo. Se um processador tenta transmitir uma mensagem que exceda este limite, ele terá que aguardar até que a mensagem possa ser enviada sem exceder o limite.

O modelo LogP é assíncrono, isto é, os processadores trabalham de modo assíncrono e o parâmetro L é usado como medida de latência. Devido a variações na latência, as mensagens enviadas por um processador podem não chegar ao seu destino na mesma ordem em que foram enviadas.

Segundo Culler *et al* [8], não podemos descrever completamente todas as máquinas reais usando um pequeno conjunto de parâmetros. No entanto, a escolha dos parâmetros representa um compromisso entre capturar fielmente as características de execução em máquinas reais e fornecer uma ferramenta razoável para o projeto e análise de algoritmos. Apesar disso, os parâmetros não são igualmente importantes em todas as situações. Em alguns casos, é possível ignorar um ou mais parâmetros. Por exemplo, em algoritmos que fazem pouca comunicação de dados, pode-se ignorar a largura de banda e os limites de capacidade. Por outro lado, em algoritmos que enviam mensagens muito longas, estas mensagens podem ser quebradas em blocos que são enviados pela rede, de tal modo que o tempo de transmissão da mensagem seja dominado pelos *gaps* entra as mensagens e a latência possa ser desconsiderada. Em algumas máquinas, o *overhead* domina o *gap*, então o g pode ser desprezado.

1.4.3 Modelo CGM

O modelo **CGM** (*Coarse Grained Multicomputer*) foi proposto por Dehne *et al* [10]. Nesse modelo, os processadores podem estar conectados por qualquer meio de interconexão. O termo "granularidade grossa" (*coarse grained*) vem do fato de que o tamanho do problema é consideravelmente maior que o número de processadores, ou seja, $n/p \gg p$.

Um **algoritmo CGM** consiste de uma seqüencia de **rodadas** (*rounds*), alternando fases bem definidas de computação local e comunicação global, como mostra a Figura 1.4. Normalmente, durante uma rodada de computação é utilizado o melhor algoritmo seqüencial para o processamento dos dados disponibilizados localmente.

O CGM é semelhante ao modelo BSP, no entanto é definido em apenas dois parâmetros:

1. n: tamanho do problema;
2. p: número de processadores disponíveis, cada um com uma memória local de tamanho $O(n/p)$.

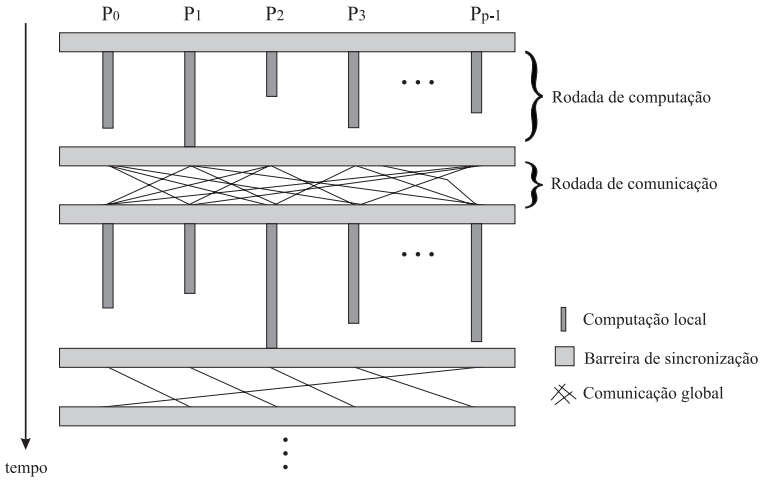


Figura 1.4: O Modelo CGM [13]

Em uma rodada de comunicação uma h-relação (com $h = O(n/p)$) é roteada, isto é, cada processador envia $O(n/p)$ dados e recebe $O(n/p)$ dados. No modelo CGM, o custo de comunicação é modelado pelo número total de rodadas de comunicação.

O custo de um algoritmo CGM é a soma dos tempos obtidos em termos do número total de rodadas de computação local (análogo ao W do modelo BSP) e o número de superpassos (análogo ao T do modelo BSP), que equivale ao número total de rodadas de comunicação.

Um algoritmo CGM é um caso especial de um algoritmo BSP onde todas as operações de comunicação de um superpasso são feitas na h-relação. Conforme observado por Dehne [9], os algoritmos CGM, quando implementados em multiprocessadores atualmente disponíveis, se comportam bem e obtêm *speedups* similares àqueles previstos em suas análises. Para estes algoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local.

1.4.4 Comparação entre os Modelos Realísticos

Todos os modelos apresentados aqui buscam estabelecer padrões amplamente aceitos que reflitam as dificuldades inerentes do próprio paralelismo. Todos eles tentam reduzir os custos de comunicação de modo muito semelhante e buscam caracterizar uma máquina paralela através de um conjunto de parâmetros. Nesta seção, faremos uma comparação entre as principais características dos três modelos apresentados e justificaremos a escolha do modelo CGM para o desenvolvimento do nosso trabalho.

LogP vs. BSP

Segundo Bilardi *et al* [3], embora os modelos BSP e LogP possam simular um ao outro de modo eficiente, o modelo BSP parece ser mais aceitável como um modelo teórico para o projeto e a análise de algoritmos paralelos e como um paradigma para a escrita de programas paralelos que sejam escaláveis e portáveis entre diversas plataformas de hardware. A principal vantagem do modelo BSP reside no fato de que ele define seus parâmetros em termos globais, aumentando o nível de abstração. Sob o modelo BSP, um projetista descreve o desempenho de um algoritmo paralelo em função de p, L, g e do tamanho do problema n. Para um dado problema, um conjunto de algoritmos pode ser proposto, variando-se somente um ou mais parâmetros que determinam o desempenho. Por exemplo, um algoritmo pode ser mais adequado para máquinas com um L pequeno, enquanto outro para máquinas com um g ou n pequenos. Para uma dada máquina BSP, cujos parâmetros são conhecidos ou mensuráveis, deve ser escolhido o algoritmo cujo desempenho (computação e comunicação) sobre a máquina específica seja o melhor possível.

BSP vs. CGM

Ambos os modelos, BSP e CGM, tentam reduzir os custos de comunicação de modo muito semelhante e buscam caracterizar uma máquina paralela através de um conjunto de parâmetros, sendo que dois destes se assemelham, o número de operações de computação local e o número de superpassos . No entanto, existem algumas diferenças. Uma delas, segundo Götz [13], é que o modelo CGM simplifica o projeto e o desenvolvimento de algoritmos por ser um modelo mais simples e levemente mais poderoso que o modelo BSP.

O foco principal do modelo CGM reside na redução do número de superpassos. Entretanto, isto parece ser adequado somente se a latência domina o custo de comunicação. No caso do modelo BSP, o foco está em minimizar a soma dos diferentes atributos, dependendo da máquina.

Além disso, um algoritmo CGM pode ser transferido para o modelo BSP sem mudanças. Se um algoritmo CGM executa W operações de computação local, T_{cp} superpassos de computação e T_{cm} superpassos de comunicação, então o algoritmo BSP correspondente terá custo $O(W + ghT_{cm} + L(T_{cp} + T_{cm}))$[5].

Em resumo, os modelos BSP e CGM apresentam muitas similaridades. Entretanto, o modelo CGM simplifica os custos de comunicação, facilitando o projeto e a análise de algoritmos.

1.5 Ambientes de Troca de Mensagens

O paradigma de troca de mensagens ou passagem de mensagens (*Message Passing*) consiste em um conjunto de métodos que torna possível a criação, a gerência, a comunicação e a sincronização entre processos quando não existe memória compartilhada.

Para permitir que linguagens como C e Fortran incorporassem esse paradigma, foram definidas extensões para essas linguagens, geralmente na forma de bibliotecas, chamadas de **ambientes de troca de mensagens**.

Nesta seção, apresentamos uma breve introdução a dois exemplos desses ambientes: o PVM (*Parallel Virtual Machine*) e o MPI (*Message Passing Interface*).

1.5.1 PVM - *Parallel Virtual Machine*

O PVM é um conjunto integrado de ferramentas de software e bibliotecas que emulam um sistema computacional concorrente heterogêneo, flexível e de propósito geral[2].

O projeto PVM foi iniciado em 1989, no *Oak Ridge National Laboratory*. A primeira versão (PVM 1.0), foi desenvolvida por Vaidy Sunderam e Al Geist, sendo utilizada apenas pelo laboratório e não disponibilizada para outras instituições. A segunda versão (PVM 2.0), contou com o auxílio da Universidade do Tennessee no desenvolvimento e uma atualização em Março de 1991, ano em que o PVM começou a ser utilizado em aplicações científicas. Após a verificação de alguns problemas, o código foi completamente reescrito, gerando a terceira versão do sistema PVM (PVM 3.0), que começou a ser distribuído como um software de domínio público, fato que contribuiu significativamente para a sua divulgação e difusão. A partir daí, várias atualizações foram feitas, sendo que a versão mais recente é a PVM 3.4.

Segundo Beguelin *et al*[2], o PVM é baseado nos seguintes princípios:

- Coleção de máquinas (*host*[1] *pool*) configurada pelo usuário: as aplicações são executadas em um conjunto de máquinas selecionadas de maneira dinâmica pelo usuário;

- Transparência de acesso ao hardware: a aplicação enxerga o hardware como uma coleção de elementos de processamento virtuais, sendo possível a atribuição de tarefas para as arquiteturas mais apropriadas;

- Computação baseada em processos: a unidade de paralelismo do PVM é uma tarefa[2] que alterna sua execução seqüencial entre computação e comunicação, sendo possível a execução de mais de uma tarefa em um elemento de processamento virtual;

- Passagem de mensagens: a coleção de tarefas que estão sendo executadas cooperam entre si, enviando e recebendo mensagens entre elas, sendo que o tamanho dessas mensagens é limitado apenas pelos recursos do sistema (memória disponível);

- Suporte a ambientes heterogêneos: o sistema PVM dá suporte à heterogeneidade em nível de arquiteturas de computadores, redes de comunicação e aplicações. Mensagens de máquinas com diferentes representações de dados podem ser trocadas e corretamente interpretadas.

O sistema PVM consiste basicamente em duas partes. A primeira parte é o PVM *daemon*[3], que reside em todas as máquinas que fazem parte da máquina virtual. Quando o usuário necessita executar uma aplicação utilizando o PVM, ele precisa iniciar o processo PVM *daemon*, através da linha de comando ou da aplicação, nas máquinas que serão utilizadas. Vários usuários podem configurar suas máquinas virtuais próprias, sem que uma interfira na máquina virtual de outro usuário[2]. A segunda parte consiste na biblioteca de comunicação PVM (*Libpvm*), que deve ser incluída nas aplicações que são desenvolvidas. Essa biblioteca disponibiliza as rotinas para comunicação, gerenciamento dinâmico e sincronização entre processos.

[1] O termo host é utilizado para designar qualquer um dos elementos de processamento (por exemplo uma estação de trabalho) que compõe a máquina virtual.

[2] Na terminologia PVM, uma tarefa é uma abstração similar aos processos no sistema UNIX, que representa o processo em execução e a estrutura necessária para efetivar a troca de mensagens.

[3] De acordo com a terminologia UNIX, *daemon* é um processo que é executado em background, atendendo requisições e/ou disponibilizando determinados serviços. Possui finalidade e funcionalidade similares aos processos servidores no modelo cliente/servidor.

Criação e Execução de Processos

O modelo computacional do PVM considera que uma aplicação é composta de várias tarefas, sendo que cada uma delas é responsável pela execução de uma parte do trabalho a ser efetuado. Assume-se também que qualquer tarefa pode enviar mensagens para outra tarefa e não existe um limite para o tamanho e quantidade dessas mensagens.

Todas as tarefas que estão sendo executadas no PVM possuem um identificador inteiro único, chamado de identificador de tarefas (TID - *task identifier*), fornecido para a aplicação pelo PVM *daemon* assim que a tarefa é criada, através da função `pvm_spawn()`. Mensagens são enviadas e recebidas especificando as tarefas envolvidas pelo seu TID. O usuário pode identificar as tarefas por meio de números naturais variando de 0 (zero) a $p - 1$, onde p é o número de tarefas envolvidas na computação.

Como já foi dito, um processo PVM *daemon* é executado em cada elemento de processamento virtual que compõe a máquina virtual. Atuando como roteador e controlador de mensagens, o PVM *daemon* fornece um ponto de contato, autenticação, controle de processos e detecção de falhas.

O primeiro PVM *daemon* executado é chamado de **mestre**, enquanto os outros são chamados de **escravos**. Durante a execução normal, não existem grandes diferenças estruturais entre os dois. A única diferença é que apenas o mestre pode efetuar as operações de gerenciamento, como iniciar outros *daemons* e anexá-los à configuração atual da máquina paralela virtual.

No caso de falha em um escravo, o mesmo é marcado como morto pelo mestre, que em intervalos de tempo regulares envia mensagens aos escravos para verificar se esses estão funcionando corretamente. Se o mestre falhar, toda a máquina virtual pára a sua execução.

Para armazenar informações a respeito da máquina virtual, o PVM *daemon* mantém algumas estruturas de dados. Entre elas estão as tabelas de *hosts* e a tabela de tarefas, que armazenam informações a respeito das tarefas que estão sendo executadas.

Comunicação e Sincronização de Processos

A *Libpvm* consiste em um conjunto de funções (biblioteca) que implementa a interface entre a aplicação e o PVM. Através da *Libpvm*, a aplicação pode conectar-se ao seu respectivo PVM *daemon*, e, conseqüentemente, unir-se à máquina virtual. Na biblioteca de comunicação são implementadas também todas as funções para gerenciamento da troca de mensagens: rotinas para criação de *buffers*, codificação, envio e recebimento. Através dessas rotinas, uma tarefa pode comunicar-se com outras tarefas.

O processo de envio de uma mensagem no PVM envolve três fases principais:

- A criação do *buffer* de envio, realizada através da função `pvm_initsend()`;
- A preparação da mensagem, conhecida também como etapa de empacotamento, que é realizada por rotinas específicas, tais como `pvm_pkint()` e `pvm_upk_int()`, para inteiros;
- O envio e o recebimento efetivo da mensagem para a outra tarefa, realizado através de funções como `pvm_send()` (para envio) e `pvm_recv()` (para recebimento).

A tarefa receptora recebe a mensagem através de uma função que retira a mensagem do *buffer*, residente no *host* receptor da mensagem. As funções para recebimento de mensagens podem aceitar quaisquer mensagens, qualquer mensagem de um *host* específico, qualquer mensagem com um identificador específico ou apenas mensagens de um *host* e identificador específicos[2].

Todos os envios (*sends*) de mensagem do PVM são não-bloqueantes e as rotinas de recebimento (*receives*) podem ser bloqueantes ou não-bloqueantes. O envio possibilita que a aplicação continue executando tão logo o *buffer* de transmissão esteja disponível para ser utilizado novamente pela aplicação, não dependendo da execução de um *receive* para poder retornar. O *send* bloqueia o processo somente quando o tamanho da mensagem excede o tamanho do *buffer* de envio e precisar ser dividida. Nesse caso, é necessário que o *host* receptor execute um *receive* para liberar o *buffer*, permitindo assim a continuidade do envio da mensagem. O *receive* bloqueante retorna apenas quando existem dados no *buffer* de recepção. A versão não bloqueante dessa função permite apenas a verificação desse *buffer*, retornando um código que indica se existem ou não mensagens no *buffer*.

Além da comunicação ponto-a-ponto, o PVM disponibiliza para a aplicação, algumas rotinas de comunicação coletiva. As rotinas `pvm_bcast()`, `pvm_gather()`, `pvm_scatter()` e `pvm_reduce()` implementam as operações de *broadcast*, *gather*, *scatter* e *reduce*, respectivamente, exemplificadas na Figura 1.5.

1.5.2 MPI - *Message Passing Interface*

O PVM foi a biblioteca que alcançou maior aceitação. Porém, segundo McBryan[18], o MPI surge como uma tentativa de padronização, indepen-

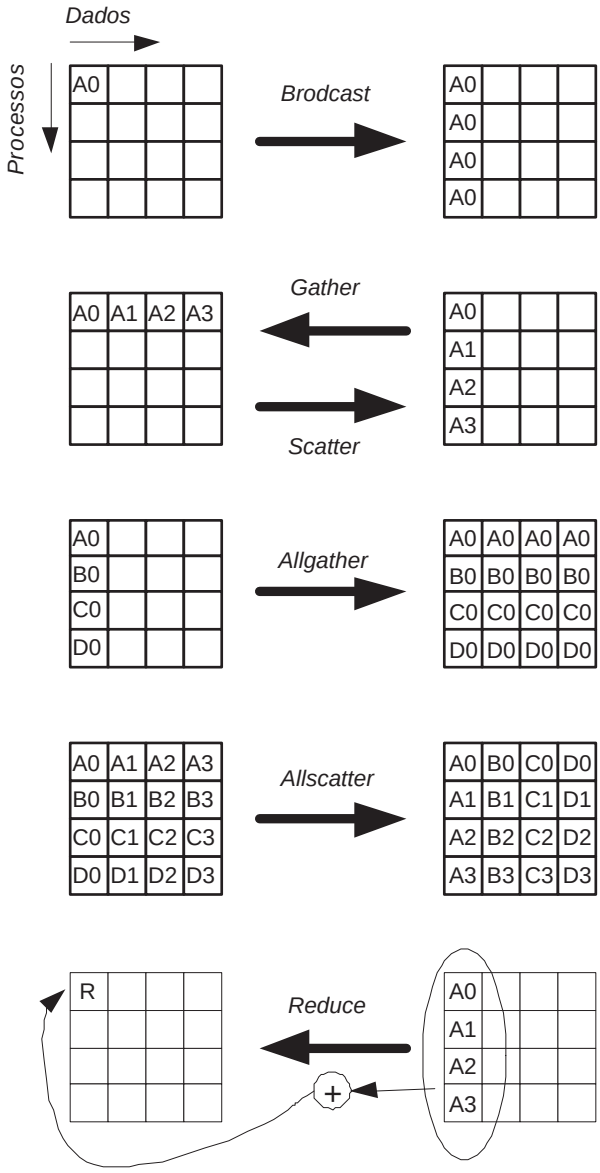


Figura 1.5: Representação das operações de comunicação coletiva.

dente do sistema paralelo, para ambientes de troca de mensagens. O processo de desenvolvimento do MPI iniciou-se em abril de 1992 com a agregação de vários representantes de várias organizações, principalmente européias e americanas, sendo que, em novembro do mesmo ano, uma primeira versão foi apresentada (MPI 1.0). Ele baseia-se nas melhores características de todas as bibliotecas de troca de mensagens, levando-se em consideração as características gerais dos sistemas paralelos, tentando explorar as vantagens de

cada um deles.

Os programas em MPI possuem o que é chamado de estilo SPMD (*Single Program, Multiple Data*), ou seja, cada processador executa uma cópia do mesmo programa. Cada instância do programa pode determinar a sua própria identidade e, dessa forma, executar operações distintas. As instâncias interagem através das funções da biblioteca MPI.

O MPI é apenas uma especificação sintática e semântica de rotinas constituintes da biblioteca de comunicação. Por se tratar de um software de domínio público, segundo o censo mantido pelo *Ohio SuperComputer Center*, atualmente existem pelo menos quinze implementações do MPI.

Assim como o PVM, o MPI define um conjunto de rotinas, que oferecem serviços de criação, execução, comunicação e sincronização de processos.

Criação e Execução de Processos

A forma de execução de um programa MPI depende da implementação que está sendo utilizada e é realizada através da linha de comando, através de um comando específico. Uma discussão sobre as implementações existentes para o MPI foge do escopo deste trabalho. No entanto, um conjunto de informações mais detalhadas podem ser encontradas na Internet. Aqui, vamos nos limitar a descrever sucintamente as principais funções de criação e execução de processos, sem nos preocuparmos com as suas formas de implementação.

A criação e execução de processos envolve basicamente quatro rotinas básicas:

- `MPI_Init()`: Inicializa a biblioteca MPI e captura os argumentos passados para o programa através da linha de comando. Cada instância do programa em execução obtém uma cópia da linha de comando. Esta função inicializa o **comunicador**[4] padrão, que é o `MPI_COMM_WORLD`, que define o escopo das operações de comunicação;
- `MPI_Comm_size()`: Determina o número de processos que foram iniciados (`size`);
- `MPI_Comm_rank()`: Determina a identidade do processo (`rank`). O *rank* varia de 0 até `size`-1;
- `MPI_Finalize()`: Finaliza o processo chamado;

[4]Outros comunicadores podem ser definidos, mas para programas simples, o comunicador padrão é suficiente.

Comunicação e Sincronização de Processos

O MPI inclui uma variedade de funções para comunicação e sincronização globais. Estas funções permitem a interação entre todos os membros de um grupo de processos iniciados. As funções de comunicação ponto a ponto mais comuns são:

- `MPI_Send()`: Envia uma mensagem para outro processo.
- `MPI_Receive()`: Recebe uma mensagem de um processo.

Ambas as rotinas acima são bloqueantes. No entanto, o MPI também suporta outros modos de envio e recebimento de mensagens, através de variantes das funções descritas acima.

Além das rotinas de comunicação ponto a ponto, o MPI também oferece rotinas para comunicação coletiva. As funções mais comumente usadas são exemplificadas na Figura 1.5 e descritas logo abaixo:

- `MPI_Bcast()`: Distribui uma cópia da mensagem para cada um dos membros do grupo de processos, incluindo o remetente;
- `MPI_Scatter()`: Particiona um vetor `a` de `size` elementos, enviando uma mensagem contendo `a[i]` para cada processo `i` pertencente ao grupo;
- `MPI_Gather()`: Concatena as mensagens recebidas de cada processo do grupo, armazenando-as em um vetor de tamanho `size`. A mensagem recebida do processo `i` é armazenada na posição `i` do vetor;
- `MPI_Reduce()`: Concatena os valores recebidos de cada processador do grupo, reduzindo-os a um único valor. A operação de redução envolve uma operação de soma, máximo ou qualquer outra operação binária associativa ou comutativa.

Além das rotinas acima, existem outras variações que podem ser utilizadas em programas MPI e que tornam transparentes para o programador as operações de comunicação global. Além disso, o MPI também implementa uma rotina de sincronização, chamada de `MPI_Barrier()`, que possibilita a sincronização de processos através da emissão de um sinal de controle que indica que todos os processos do grupo chamaram a função.

1.5.3 PVM vs. MPI

Uma discussão das diferenças entre as duas ferramentas é aceitável apenas ao nível de funcionalidade oferecido pelas respectivas bibliotecas de troca de mensagem, visto que o PVM é uma implementação completa, incluindo a especificação e implementação propriamente dita, e o MPI é apenas uma descrição sintática e semântica de uma biblioteca. Detalhes de implementação não são tratados a fundo por documentos do padrão MPI.

Fundamentalmente, o MPI e o PVM diferem na relação que apresentam entre complexidade e funcionalidade. O MPI possui uma especificação longa e relativamente complexa, para se tornar viável como padrão eficiente para arquiteturas computacionais tão diversas e para fornecer todas as características consideradas importantes para uma biblioteca de troca de mensagens. O PVM, por outro lado, considera como um objetivo de projeto ser simples e suficientemente completo, a fim de facilitar o trabalho do programador.

Enfim, a escolha de uma ferramenta deve ser feita levando em conta quais as necessidades da aplicação, sobre qual sistema paralelo será executada e a relação entre as características do ambiente e a implementação a ser utilizada, entre outros aspectos. Dessa maneira, devemos ser cauteloso para afirmar que uma plataforma de portabilidade seja, no geral, melhor que outra. Apesar disso, sabe-se que já se passaram alguns anos desde o lançamento da última versão do PVM, enquanto que o MPI continua avançando.

1.6 Conclusão

Neste capítulo, iniciamos com uma visão geral da área de computação paralela, do problema do Fecho Transitivo e seu uso na resolução de outros problemas clássicos em grafos. Em seguida, apresentamos os conceitos fundamentais, estruturas de representação e algoritmos de percurso de grafos que serão necessários para o entendimento das técnicas e estruturas de dados utilizadas pelos algoritmos na resolução dos problemas estudados.

Na Seção 1.3 foram apresentados o conceito de processamento paralelo, as métricas utilizadas para análise de algoritmos paralelos e os principais fatores que determinam o desempenho desses algoritmos sobre sistemas paralelos de computação.

Nas Seções 1.4 e 1.5 foram apresentados, respectivamente, uma descrição dos principais modelos de computação paralela, incluindo uma breve comparação entre eles e as principais características e funções de dois ambientes de troca de mensagens utilizados para a implementação de algoritmos paralelos.

Como já foi dito, nosso estudo concentra-se no algoritmo paralelo para o fecho transitivo, proposto por Cáceres *et al*[6], que utiliza o modelo BSP/CGM. Além disso, Leighton[17] apresenta alguns algoritmos paralelos para modelos de memória distribuída, que utilizam o fecho transitivo como subrotina para sua resolução. Entre eles incluem-se o problema dos componentes conexos, dos caminhos mais curtos, da busca em largura e da árvore geradora mínima. A partir destes algoritmos, vamos desenvolver os algoritmos BSP/CGM, demonstrando, através de resultados empíricos, o uso deste modelo na implementação de algoritmos paralelos para os problemas citados acima.

No Capítulo 2, apresentamos os principais algoritmos seqüenciais conhecidos e que resolvem os problemas citados acima. Além disso, é descrita uma solução seqüencial usando a mesma estrutura do algoritmo de Warshall para cada um dos problemas estudados. O objetivo é possibilitar a compreensão da idéia a ser aplicada, antes da paralelização dos algoritmos.

No Capítulo 3, apresentamos o algoritmo de Cáceres *et al*[6], que utiliza a estrutura do algoritmo de Warshall para implementar um algoritmo paralelo no modelo BSP/CGM para o fecho transitivo. Além disso, neste capítulo também descrevemos outros problemas em grafos relacionados que podem ser resolvidos usando a mesma estrutura do algoritmo de Warshall.

No Capítulo 4, descrevemos o ambiente de implementação, incluindo alguns detalhes da linguagem e do ambiente utilizados. Além disso, são apresentadas as tabelas e os gráficos de tempo obtidos da execução dos algoritmos implementados.

Finalmente, no Capítulo 5, apresentamos as conclusões, envolvendo comentários sobre os resultados obtidos, as dificuldades encontradas e algumas sugestões para trabalhos futuros.

Capítulo 2

Algoritmos Seqüenciais

Neste capítulo, apresentamos a descrição dos principais algoritmos seqüenciais para resolver alguns problemas clássicos em teoria dos grafos. Vamos começar com o algoritmo de Warshall [36] para determinar o fecho transitivo de um grafo dirigido. Em seguida, apresentaremos algoritmos para caminhos mais curtos, busca em largura e árvore geradora mínima.

Embora, aparentemente, os problemas sejam muito diferentes, procuraremos descrever todos os algoritmos usando a mesma estrutura fundamental, baseada no algoritmo seqüencial de Warshall para o fecho transitivo. Além disso, este capítulo apresenta os principais algoritmos que tratam cada um dos problemas acima de forma independente. O objetivo é compreender cada um dos problemas estudados e a idéia dos processos seqüenciais que serão utilizados pelos algoritmos paralelos descritos no próximo capítulo.

2.1 Fecho Transitivo

Segundo Nuutila [22], o **Fecho Transitivo** de um grafo dirigido é um importante subproblema de diversas aplicações computacionais em redes de computadores, sistemas paralelos e distribuídos, banco de dados e no projeto de compiladores.

Segundo Leighton [17], a necessidade de computar o fecho transitivo apresenta-se, normalmente, em situações nas quais um grafo é utilizado para representar relações ou precedência entre objetos. Por exemplo, uma aresta dirigida do vértice i ao vértice j pode representar a seguinte relação: i é mais ou tão importante quanto j, denotada por $i \geq j$. Portanto, calculando o fecho transitivo nós estamos computando todas as deduções da relação estabelecida entre os vértices do grafo, ou seja, se $i \geq j$ e $j \geq k$, então $i \geq k$. De outro modo, a aresta dirigida de i a j pode indicar a dependência que

a tarefa representada pelo vértice j possui em relação à tarefa representada pelo vértice i, de tal forma que o início de j está condicionada ao término de i. Neste caso, o fecho transitivo provê uma tabela de precedências completa do grafo. A Figura 2.1 mostra um grafo G, o seu fecho G^* e as respectivas matrizes de adjacência A e A^*.

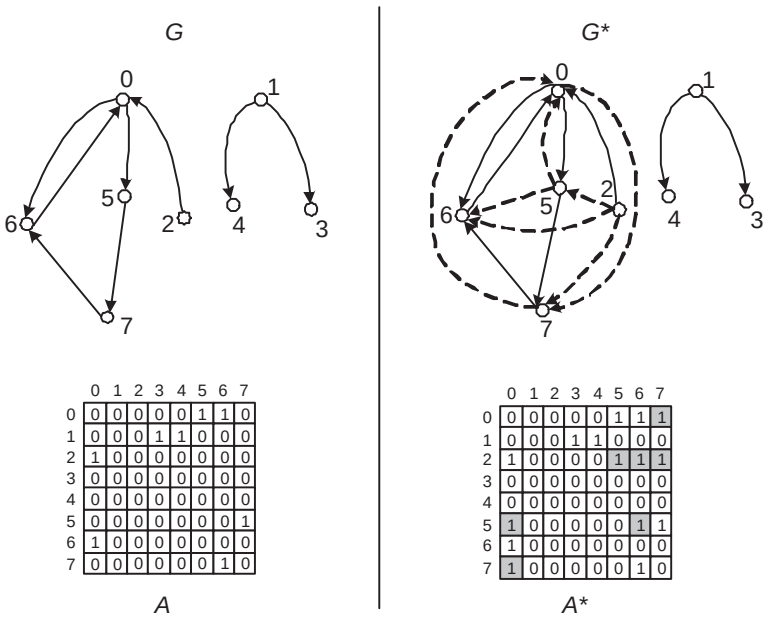


Figura 2.1: Um exemplo de um grafo G e o seu fecho transitivo G^* e suas respectivas matrizes de adjacências A e A^*. As posições sombreadas da matriz A^* correspondem às arestas pontilhadas em G^*.

Nos últimos anos, muitos algoritmos para computar o fecho transitivo têm sido propostos. Apesar disso e da constante melhoria no desempenho dos computadores, ainda existe a necessidade de algoritmos e representações mais eficientes para o fecho transitivo. Segundo Nuutila [22], uma das razões pela qual isso acontece é a quantidade de memória exigida pelos algoritmos conhecidos para o problema do fecho transitivo. Outro fator a ser observado é que o tamanho das entradas parece crescer na mesma proporção que o aumento da capacidade de memória dos computadores. Já que a velocidade da CPU também cresce na mesma proporção que a capacidade de memória, somente os algoritmos lineares conseguem manter os seus tempos de execução sobre entradas padrão e os algoritmos usuais para o fecho transitivo, tais como [11, 21, 27, 35, 36], não são lineares.

2.1.1 Algoritmo de Warshall

O Algoritmo de Warshall [36] é o mais conhecido algoritmo seqüencial para calcular o fecho transitivo e é utilizado por muitos autores que apresentam algoritmos para problemas em grafos, tais como [1] e [28].

A idéia do algoritmos de Warshall é a seguinte: se um grafo contém caminhos $v \xrightarrow{*} w$ e $w \xrightarrow{*} u$ de tal forma que os vértices internos ao caminho pertençam a um conjunto específico S, então o grafo também contém um caminho $v \xrightarrow{*} u$, com os vértices internos pertencendo ao conjunto $S \cup \{w\}$.

A entrada do algoritmo consiste de um grafo dirigido $G = (V, E)$. A **matriz de adjacências** é a estrutura escolhida para a representação dos grafos de entrada e saída do algoritmo. O algoritmo transforma uma matriz de adjacências que representa um grafo de entrada em uma matriz de adjacências que representa o fecho transitivo, obtida por três laços aninhados que fazem o percurso através dos potenciais caminhos do grafo de entrada. Este algoritmo é muito semelhante ao de multiplicação de matrizes. Portanto, é curto e fácil de implementar, mas a sua complexidade de tempo é $O(n^3)$. Segundo Nuutila [23], isso torna este algoritmo inferior a alguns outros algoritmos seqüenciais recentemente desenvolvidos para o problema.

Se interpretarmos uma relação binária A sobre um conjunto finito S como um grafo dirigido, então, encontrar os elementos de R, o fecho transitivo da relação, corresponde a inserir arestas no grafo dirigido. Em particular, para qualquer par de arestas $s_i s_k$ e $s_k s_j$ inseridos em R, nós acrescentamos a aresta $s_i s_j$ ao conjunto. Ou seja, nós podemos assumir que $s_i R s_j$ se nós já sabemos que, para algum k, $s_i R s_k$ e $s_k R s_j$.

Baase[1] discute uma forma pouco eficiente de computar o fecho transitivo, utilizando um laço externo **repita** que será executado até que todas as arestas do fecho sejam computadas e organizando a tripla de laços **para** de tal modo que o índice k controle o laço mais interno. Dessa forma, uma mesma tripla poderá ser processada pelo algoritmo mais de uma vez e o algoritmo terá complexidade de $O(n^4)$. O algoritmo 1 corresponde ao algoritmo de Warshall. Ele simplesmente processa as triplas de vértices que representam as arestas que estão sendo analisadas na ordem correta e possui complexidade de $O(n^3)$.

Existem algumas variações do algoritmo de Warshall que foram propostas posteriormente com o objetivo de melhorar o tempo de processamento. Uma delas foi apresentada por Warren[35]. No algoritmo de Warren, a matriz é percorrida linha por linha em dois passos. No primeiro passo são examinadas as posições abaixo da diagonal principal e no segundo passo, as posições acima da diagonal principal. Segundo Nuutila [22], ambos os algoritmos percorrem e marcam as mesmas posições da matriz, mas em ordens distintas, o que dá

ALGORITMO 1: Algoritmo de Warshall

Entrada: Matriz de adjacências $A_{n\times n}$ do grafo G
Saída: Fecho transitivo do grafo G

```
1: para k ← 1 até n faça
2:    para i ← 1 até n faça
3:       para j ← 1 até n faça
4:          A[i, j] ← A[i, j] ou (A[i, k] e A[k, j])
5:       fim para
6:    fim para
7: fim para
```

ao algoritmo de Warren uma certa vantagem sobre o de Warshall devido ao melhor aproveitamento dos dados que estão disponíveis na memória.

Como já foi dito, muitos algoritmos melhores que o de Warshall foram propostos para o problema do fecho transitivo. No entanto, estamos interessados na estrutura do algoritmo de Warshall, que poderá ser utilizada como base para o desenvolvimento de algoritmos que resolvem os problemas descritos nas seções que se seguem.

2.2 Caminhos Mais Curtos

Dado um grafo G orientado com pesos w_{ij} nas suas arestas, o problema de calcular os **caminhos mais curtos** para cada par de vértices i a j, consiste em encontrar um caminho cuja soma dos pesos das arestas no caminho entre i e j seja mínimo. Para garantir que um menor caminho entre cada par de vértices poderá ser encontrado, vamos assumir que cada uma das possíveis arestas de G, exceto os laços, estão presentes, possivelmente com peso infinito. Além disso, G não possui circuitos de peso negativo.

O algoritmo de Dijkstra encontra os caminhos mínimos a partir de um vértice sobre grafos com pesos não negativos e tem como entrada um grafo dirigido (ou não dirigido) G e dois vértices específicos v e w. O problema se restringe em encontrar o caminho mais curto entre v e w. Este algoritmo determina somente os caminhos mais curtos entre v e w, e todos os vértices examinados antes que w seja encontrado. Ou seja, se desejarmos que o algoritmo de Dijkstra determine os caminhos mais curtos de v a todos os vértices do grafo, e não apenas até o vértice w, devemos modificá-lo.

O algoritmo de Bellman-Ford também apresenta-se como uma opção para encontrar os caminhos mais curtos em um grafo com pesos nas arestas. A diferença é que este último calcula os caminhos mais curtos de um vértice inicial para todos os demais e admite grafos com pesos negativos nas arestas.

A única restrição é, como já foi dito no início desta seção, que o grafo não possua um circuito de peso negativo.

Por outro lado, o problema dos caminhos mais curtos entre todos os vértices de um grafo dirigido pode ser diretamente resolvido por um algoritmo que possui estrutura semelhante ao algoritmo de Warshall, apresentado na Seção 2.1.1.

Nas subseções seguintes vamos apresentar os algoritmos citados acima.

2.2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra é um algoritmo guloso para encontrar os caminhos mais curtos a partir de um grafo com pesos não negativos nas arestas. Este algoritmo faz uso de uma técnica denominada **técnica do relaxamento ou relaxação**, na qual as distâncias dos menores caminhos são atualizadas.

Este algoritmo mantém um conjunto S que contém os vértices com os caminhos mais curtos calculados até o momento. Para cada vértice $v \in V$, existe um valor $d[v]$ que funciona como um limitante superior para um caminho mais curto do vértice s até o vértice v. O valor de $d[v]$ significa que temos um caminho de s para v de peso $d[v]$. A cada iteração, o algoritmo seleciona um novo vértice v para ser incluído em S, tal que v é escolhido entre os vértices de $V - S$ com menor valor de $d[v]$. O vértice v é incluído em S e, em seguida, todas as arestas que saem de v são processadas pelo algoritmo. O peso da aresta $i \rightarrow j$ está armazenado na posição $w(i, j)$ de uma matriz de pesos. Caso esta aresta não exista, $w(i, j) = \infty$.

O algoritmo 2 descreve o algoritmo de Dijkstra. Ao final da execução, o algoritmo encontrará os caminhos mais curtos entre um vértice de origem e cada um dos vértices visitados até que o vértice de destino seja encontrado.

A complexidade de tempo do passo 1 do algoritmo de Dijkstra é $O(|V|)$. Assumindo que usamos um vetor linear para representar $d[]$, indexando pelos vértices e Q sendo implementada como uma lista de adjacências, podemos realizar o passo 8 complexidade de tempo total $O(|V|^2)$ e o passo 10 com complexidade $O(|E|)$. Assim, a complexidade final do algoritmo ficaria $O(|V|^3)$.

Na subseção seguinte, apresentamos o algoritmo de Bellman-Ford.

2.2.2 Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford faz uso da mesma técnica utilizada pelo algoritmo de Dijkstra. Este algoritmo computa os caminhos mais curtos de um vértice inicial de origem a todos os demais, inclusive em grafos com pesos negativos. A única restrição é que o grafo não possua nenhum circuito de peso negativo.

ALGORITMO 2: Algoritmo de Dijkstra

Entrada: (1) Lista de arestas do grafo G; (2) Vértice s de origem e t de destino; (3) Matriz w de pesos das arestas;
Saída: Caminho mais curto de s até t

1: **para** cada vértice $v \in V$ **faça**
2: $d[v] \leftarrow \infty$
3: **fim para**
4: $d[s] \leftarrow 0$
5: $S \leftarrow \emptyset$
6: $Q \leftarrow V$
7: **enquanto** $Q \neq 0$ **faça**
8: $u \leftarrow$ Extrai_Min(Q) {Em relação a $d[]$}
9: $S \leftarrow S \cup \{u\}$
10: **para** cada vértice v adjacente a u **faça**
11: **se** $d[v] > d[u] + w(u, v)$ **então**
12: $d[v] \leftarrow d[u] + w(u, v)$
13: **fim se**
14: **fim para**
15: **fim enquanto**

Este algoritmo inicializa a distância do vértice de origem com o valor 0 e todos os demais vértices do grafo com ∞. Depois disso, o algoritmo executa $|V| - 1$ interações sobre todas as arestas do grafo e atualizando a distância até o destino de cada aresta. Por último, o algoritmo verifica cada uma das arestas novamente para detectar circuitos com peso negativo. Se algum circuito de peso negativo for encontrado, o algoritmo retorna FALSO, caso contrário, o algoritmo retorna VERDADEIRO, juntamente com os caminhos mais curtos encontrados a partir do vértice de origem.

O algoritmo 3 descreve o algoritmo de Bellman-Ford.

Como já foi dito, se existir um circuito negativo, não poderemos garantir que os caminhos encontrados nos grafos correspondem aos caminhos mais curtos. Ou seja, se existirem arestas (u, v) tais que $w(u, v) + d[u] < d[v]$, o algoritmo retorna FALSO. Esse teste é realizado pelo passo 12 do algoritmo.

A complexidade de tempo do algoritmo de Bellman-Ford é $O(|E||V|)$. Dessa forma, se você precisa resolver o problema dos caminhos mais curtos para um grafo com arestas com peso positivo, o algoritmo de Dijkstra nos dá uma solução mais eficiente. Se todas as arestas do grafo possuem peso igual a 1, um algoritmo de busca em largura, que será discutido mais adiante, é o mais indicado. Por fim, para encontrar os caminhos mais curtos entre todos os vértices de um grafo com pesos nas arestas, vamos apresentar na subseção seguinte o algoritmo de Floyd-Warshall.

ALGORITMO 3: Algoritmo de Bellman-Ford

Entrada: (1) Grafo G com pesos nas arestas; (2) Vértice s de origem; (3) Matriz w de pesos das arestas;
Saída: Caminho mais curto de s até todos os demais vértices de G

```
1: para cada vértice v ∈ V faça
2:    d[v] ← ∞
3: fim para
4: d[s] ← 0
5: para i ← 1 até |V| − 1 faça
6:    para cada aresta (u, v) ∈ E faça
7:       se d[v] > d[u] + w(u, v) então
8:          d[v] ← d[u] + w(u, v)
9:       fim se
10:   fim para
11: fim para
12: para cada aresta (u, v) ∈ E faça
13:   se d[v] > d[u] + w(u, v) então
14:      retorne FALSO
15:   fim se
16: fim para
17: retorne VERDADEIRO
```

2.2.3 Algoritmo Usando o Fecho Transitivo

Nesta subseção, estamos interessados em encontrar os caminhos mais curtos entre todos os pares de vértices no grafo. Na realidade, este problema pode ser resolvido executando os algoritmos para os caminhos mais curtos apresentados anteriormente, a partir de um vértice, para cada elemento de V. O algoritmo apresentado aqui também é conhecido como **Algoritmo de Floyd-Warshall**.

Se todas as arestas de G são não negativas, podemos usar o algoritmo de Dijkstra, o que nos dá um algoritmo de complexidade $O(|V|^3)$.

Se o algoritmo contém arestas de peso negativo, mas sem circuitos de peso negativo, podemos usar o algoritmo de Bellman-Ford, que nos dá um algoritmo com complexidade de tempo $O(|V|^2|E|)$ ou $O(|V|^4)$, no caso de grafos densos.

Para o algoritmo de Floyd-Warshall, vamos supor que temos uma matriz de pesos $w_{n\times n}$, tal que a posição $w(i, j)$ da matriz de adjacências armazena o peso da aresta $i \to j$. Caso esta aresta não exista, $w(i, j) = \infty$.

O algoritmo de Floyd-Warshall considera os vértices intermediários de um caminho mais curto P, ou seja, os vértices de P que não são os extremos e consiste de n iterações, onde n é o total de vértices do grafo. Na primeira iteração, trocamos a aresta $i \to j$, para $1 \leq i, j \leq n$, pelo caminho mais

curto de i a j, exceto i e j, que passe somente pelo vértice 1. Esta operação é executada pela comparação entre $w(i,1) + w(1,j)$ e $w(i,j)$ e selecionando o menor valor, onde $w(i,j) = w_0(i,j)$ corresponde ao peso da aresta $i \rightarrow j$. O resultado desta comparação é chamado $w_1(i,j)$. Na segunda iteração, trocamos o caminho de i a j calculado durante a primeira iteração pelo caminho de menor peso de i a j que, desta vez, pode passar pelos vértices 1 e 2. Este caminho é determinado pela comparação entre $w_1(i,2) + w_1(2,j)$ e $w_1(i,j)$. O menor entre esses dois valores será $w_2(i,j)$. Durante a k-ésima iteração, computamos

$$w_k(i,j) = \min(w_{k-1}(i,j), w_{k-1}(i,k) + w_{k-1}(k,j)) \tag{2.1}$$

para determinar o caminho mais curto entre i e j que passa somente pelos vértices $1, 2, \ldots, k$. O caminho mais curto entre cada par de vértices será encontrado após a n-ésima iteração. A figura 2.2 ilustra as iterações do algoritmo. As posições $w_{ij}^{(k)}$ em cada matriz indicam o peso de caminho mais curto de i a j que atravessa somente os vértices $\{1, 2, \ldots, k\}$. Para simplificar, o grafo da figura não inclui os caminhos calculados a cada estágio do algoritmos, mas estes não são difíceis de serem encontrados.

Algoritmo 4: Algoritmo de Floyd-Warshall

Entrada: Matriz de adjacências $A_{n \times n}$ do grafo G, contendo os pesos das arestas
Saída: Na matriz A, a distância entre todos os pares de vértices de G

```
1: para k ← 1 até n faça
2:    para i ← 1 até n faça
3:       para j ← 1 até n faça
4:          w(i,j) ← min(w(i,j), w(i,k) + w(k,j))
5:       fim para
6:    fim para
7: fim para
```

O algoritmo 4 descreve o algoritmo de Floyd-Warshall. Ao final da execução, o algoritmo encontrará os caminhos mais curtos entre todos os pares de vértices do grafo de entrada. Este algoritmo possui complexidade de $O(|V|^3)$. A Figura 2.2 ilustra a execução do algoritmo 4.

2.3 Busca em Largura

Dado um grafo G não dirigido, sem pesos nas arestas e um vértice de origem s, denominado raiz, uma **busca em largura** percorre de forma sistemática as arestas de G para encontrar todo vértice que pode ser alcançado a partir

da raiz. O nome busca em largura vem do fato que o algoritmo visita primeiramente todos os seus vizinhos antes de proceder a busca sobre os vizinhos de seus vizinhos. Uma forma de pensar na busca em largura é pensar no movimento de uma onda que se expande quando jogamos um pedra em uma bacia com água. Os vértices na mesma "onda"estão à mesma distância da raiz.

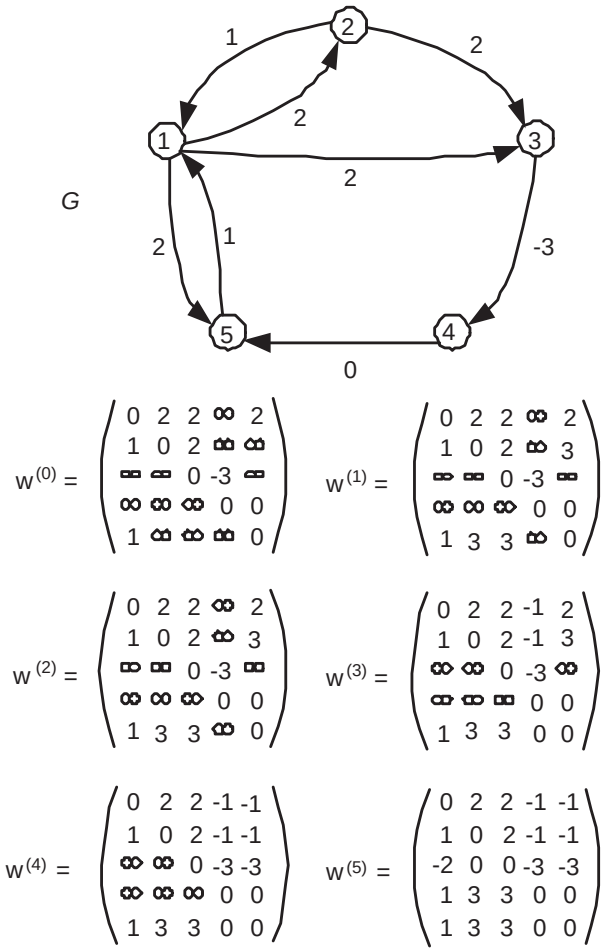


Figura 2.2: [17] Um exemplo mostrando os passos da execução do algoritmo 4 que computa os caminhos mais curtos em um grafo G.

2.3.1 Algoritmo de Busca em Largura

Este algoritmo constrói uma árvore T sobre G denominada **árvore geradora de busca em largura** ou simplesmente **árvore de busca em largura**, que inicialmente contém apenas o vértice raiz. Cada vez que um novo vértice v é encontrado pelo algoritmo através de uma aresta $e = (u, v)$, onde u já foi encontrado anteriormente, a aresta e é inserida na árvore. Após a execução do algoritmo, cada caminho de um vértice qualquer até a raiz é o caminho mais curto em G. Neste caso, estamos considerando que todas as arestas possuem peso igual a 1. A figura 2.3 mostra um grafo e uma de suas árvore de busca em largura. Observe que uma árvore de busca em largura não é necessariamente única, mesmo que a mesma raiz seja selecionada.

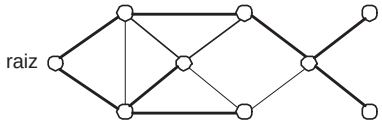


Figura 2.3: Um grafo e uma de suas árvores de busca em largura.

O algoritmo 5 descreve o algoritmo de busca em largura. A busca em largura é executada com o auxílio de uma fila Q, na qual são inseridos os vizinho de cada vértice visitado para posterior análise.

Cada vértice é colorido como *branco* somente uma vez, e é colocado na fila com a cor *cinza*. Como cada operação de inserir e remover da fila gasta tempo $O(1)$, o tempo total para as operações na fila é $O(|V|)$. Para cada vértice, o laço do passo 10 percorre todos os seus vizinhos. Dessa forma, cada aresta é atravessada duas vezes, uma para cada extremidade. Portanto, o tempo total do algoritmo é $O(|V| + |E|)$.

2.3.2 Algoritmo Usando Fecho Transitivo

Dado um grafo G conexo, não dirigido e sem pesos nas arestas, uma árvore geradora de busca em largura T de G pode ser facilmente encontrada usando a mesma idéia do algoritmo de Warshall para o fecho transitivo.

O algoritmo que resolve o problema de computar uma árvore de busca em largura de um grafo G, baseado no fecho transitivo, é muito semelhante ao algoritmo que resolve o problema de computar os caminhos mais curtos de G, apresentado na Seção 2.2.3. Neste caso particular, é atribuído peso 1 para cada aresta $e \in E$ e peso ∞ para cada aresta $a \notin E$. O peso do caminho

ALGORITMO 5: Algoritmo de Busca em Largura

Entrada: (1) Grafo G; (2) Vértice raiz s.
Saída: A árvore T de busca em largura de G

```
1: para cada vértice u ∈ V − s faça
2:     cor(u) ← branco
3:     d(u) ← ∞
4: fim para
5: cor(s) ← cinza
6: d(s) ← 0
7: Q ← {s}
8: enquanto Q ≠ 0 faça
9:     u ← inicio(Q)
10:    para cada vértice v adjacente a u faça
11:        se cor(v) = branco então
12:            cor(v) ← cinza
13:            d(v) ← d(u) + 1
14:            InsereFila(Q, v)
15:        fim se
16:        RemoveFila(Q)
17:        cor(u) ← preto
18:    fim para
19: fim enquanto
```

mais curto da raiz até um determinado vértice é simplesmente a distância do vértice até a raiz do grafo.

Dessa forma, após aplicar o algoritmo de Floyd-Warshall para encontrar os menores caminhos entre todos os pares de vértices de G, uma árvore de busca em largura pode ser encontrada através da seleção das arestas que ligam um vértice a uma distancia i da raiz a um outro vértice a uma distância $i-1$ para todo $i > 0$. Obviamente, por definição, cada um dos vértices que se encontra a uma distância i da raiz pode estar ligado a mais de um vértice a uma distância $i-1$. No entanto, não importa qual dos vértices será selecionado, visto que a árvore geradora não é única e o algoritmo se propõe a encontrar apenas uma.

Após a execução do Algoritmo 6 a matriz *BFSTree* armazena a árvore geradora encontrada. O vetor *dist* é um vetor auxiliar utilizado para armazenar as distâncias da raiz escolhida a todos os demais vértices do grafo G. Esta informação é importante para a identificação das arestas que irão compor a árvore geradora.

A complexidade do Algoritmo 6 é de $O(|V|^3)$ em virtude da execução do algoritmo de Floyd-Warshall como subrotina para encontrar todos os caminhos mais curtos em G. Observe que a complexidade não demonstra nenhum tipo de ganho que possa ser obtido. No entanto, neste trabalho, estamos in-

ALGORITMO 6: Busca em Largura

Entrada: (1) Matriz de adjacências $A_{n\times n}$ do grafo G; (2) Vértice raiz r.
Saída: Árvore geradora de busca em largura do grafo

```
1: Calcule os caminhos mais curtos entre os vértices de G (Floyd-Warshall)
2: para i ← 1 até n faça
3:    dist[i] ← A[i][r]
4:    para j ← 1 até n faça
5:       BFSTree[i][j] ← ∞
6:    fim para
7: fim para
8: para cada vértice v ∈ V e v ≠ r faça
9:    para cada vértice s ∈ V faça
10:      se dist[s] = dist[v] − 1 então
11:         BFSTree[v][s] ← 1
12:         BFSTree[s][v] ← 1
13:      fim se
14:   fim para
15: fim para
```

teressados somente em mostrar como a estrutura do algoritmo de Warshall pode ser utilizada para a implementação de um algoritmo para a busca em largura.

2.4 Árvore Geradora Mínima

Dado um grafo G não dirigido e com peso nas arestas, uma **árvore geradora de custo mínimo** ou simplesmente **árvore geradora mínima** consiste de uma árvore geradora com o menor valor possível para o somatório dos pesos de todas as arestas de G. A necessidade de se encontrar a árvore geradora mínima aparece em muitas aplicações, particularmente quando utilizamos grafos para modelar o projeto de redes de computadores, envolvendo retransmissão de mensagens entre estações (*broadcast*).

Para simplificar, vamos assumir que todas as arestas do grafo possuem pesos distintos. Vale lembrar que esta restrição não é significante, visto que duas arestas com o mesmo peso podem ser diferenciadas se assumirmos que as suas extremidades (i, j) podem funcionar como rótulos.

2.4.1 Algoritmo de Kruskal

O algoritmo de Kruskal é um algoritmo guloso que constrói um subgrafo T de G, onde T corresponde a árvore geradora mínima de G. Inicialmente, T contém somente os vértices de G. A cada iteração é acrescida uma aresta

ao subgrafo T. O algoritmo escolhe a próxima aresta e com menor peso, de forma que $T+e$ continue acíclico. O algoritmo procede desta maneira até que todas as arestas tenham sido examinadas. O grafo T resultante corresponde, então, a árvore geradora mínima de G.

ALGORITMO 7: Algoritmo de Kruskal

Entrada: (1) Grafo $G = (V, E)$; (2) Matriz de pesos w.
Saída: Árvore geradora mínima T.

```
1: T ← ∅ {conjunto de arestas da árvore geradora mínima}
2: C ← ∅ {conjunto de componentes}
3: para todo v ∈ V faça
4:    C ← C + {v}
5: fim para
6: para cada vértice {x, y} ∈ E, em ordem não decrescente faça
7:    Seja c_x e c_y os conjuntos de x e de y em C
8:    se c_x ≠ c_y então
9:       T ← T + {x, y}
10:      C ← C − c_x − c_y + {(c_x ∪ c_y)}
11:   fim se
12: fim para
```

A complexidade do algoritmo de Kruskal depende da implementação usada para manipular os conjuntos de componentes. Observe que somente a ordenação das arestas já gasta tempo $O(|E| \log |V|)$. A implementação mais simples para a manipulação dos conjuntos é utilizar um vetor de rótulos, indexado pelos vértices de G. Inicialmente, cada posição possui um rótulo distinto. Dessa forma, encontrar um rótulo de um conjunto gasta tempo $O(1)$. Para fazer a união entre dois conjuntos podemos trocar todos os rótulos de um conjunto pelo do outro, gastando tempo $O(|V|)$. Como são feitas $|V|-1$ uniões, tem-se uma implementação com complexidade de tempo $O(|E| \log |V| + |V^2|)$.

2.4.2 Algoritmo de Prim

O algoritmo de Prim consiste em iniciar a construção de uma árvore a partir de um vértice qualquer e ir incrementando esta árvore com arestas, sempre mantendo-a acíclica e conexa. A próxima aresta a ser incluída deve ser uma de menor peso, daí o porque deste algoritmo também ser considerado como um algoritmo guloso.

A chave para uma boa implementação deste algoritmo é fazer a busca da próxima aresta de forma eficiente. Neste caso, pode ser usada a estrutura de árvore AVL. Para esta estrutura, a construção da árvore pode ser feita em

ALGORITMO 8: Algoritmo de Prim

Entrada: (1) Grafo $G = (V, E)$; (2) Matriz de pesos w; (3) Vértice raiz r.
Saída: Árvore geradora mínima T.

```
1: Q ← V {O conjunto de vértices inseridos em T é V − Q. Inicialmente T é vazio.}
2: para todo v ∈ Q faça
3:    peso(v) ← ∞
4: fim para
5: peso(r) ← 0
6: predecessor(r) ← NULL
7: enquanto Q ≠ ∅ faça
8:    u ← Extrai_Min(Q) {será inserida a aresta (u, predecessor(u)) em T.}
9:    para cada vértice v adjacente a u faça
10:      se v ∈ Q e w(u, v) < peso(v) então
11:         predecessor(v) ← u
12:         peso(v) ← w(u, v)
13:      fim se
14:   fim para
15: fim enquanto
```

tempo $O(n \log n)$ e as operações de consulta, inserção e remoção em tempo $O(\log n)$. Portanto, a complexidade do algoritmo usando esta estrutura é de $O(|E| \log |V|)$.

2.4.3 Algoritmo Usando Fecho Transitivo

Para explicar o algoritmo que encontra a árvore geradora mínima usando a mesma estrutura do algoritmo de Floyd-Warshall, vamos fazer uso do seguinte fato.

Lema 1 *Se todas as arestas possuem pesos distintos, então a aresta (i, j) pertence a árvore geradora mínima de G se, e somente se, todo caminho de comprimento maior ou igual a dois entre i e j contém uma aresta com peso maior que $w_{i,j}$.*

Prova. Seja $w_{i,j}$ o peso da aresta (i, j) para $1 \leq i, j \leq N$, e seja T uma árvore geradora mínima de G. Primeiramente, vamos mostrar que se $(i, j) \in T$, então todo caminho de comprimento maior ou igual a dois de i a j em G contém uma aresta com peso maior que $w_{i,j}$. A prova é por contradição; isto é, assumimos que $(i, j) \in T$ e que existe um caminho $\mathcal{P}_{i,j}$ de i a j com todas as arestas com peso menor que $w_{i,j}$. Se removermos a aresta (i, j) de T, produziremos duas subárvores T_i e T_j que, juntas, alcançam todos os vértices de G. Entretanto, T_i contém o vértice i e T_j contém o vértice j. Visto que o caminho $\mathcal{P}_{i,j}$ liga os vértices i e j, ele deve conter alguma aresta (i', j') ligando

T_i a T_j. Seja $T' = T_i \cup T_j \cup (i', j')$. Observe que T' é uma árvore geradora, pois ela contém $n - 1$ arestas e não contém ciclos. Observe também que o peso de T' é menor do que o peso de T, pois por suposição, $w_{i',j'} < w_{i,j}$. Isso contradiz a hipótese de que T é uma árvore geradora mínima, o que conclui a primeira parte da prova.

Por outro lado, nós assumimos que todo caminho de comprimento maior ou igual a dois de i a j contém uma aresta de peso maior que $w_{i,j}$, mas que $(i', j') \notin T$. Seja (i', j') uma aresta de peso maior que $w_{i,j}$ no caminho que liga i a j em T, e sejam $T_{i'}$ e $T_{j'}$ as subárvores de T formadas pela remoção da aresta (i', j') de T. Pelo mesmo argumento anterior, podemos concluir que $T' = T_i \cup T_j \cup (i, j)$ é uma árvore geradora de G com peso menor do que T, o que é uma contradição. Dessa forma, o lema está demonstrado. □

A partir do Lema 1, o problema de descrever um algoritmo para encontrar a árvore geradora mínima fica mais simples. Na realidade, o algoritmo é idêntico ao algoritmo para encontrar os caminhos mais curtos, exceto pelo fato de que se define como rótulo para o caminho o valor do peso da aresta mais pesada ao invés da soma dos pesos das arestas. Portanto, para computar o caminho mínimo de i a j nesta nova visão, utiliza-se o algoritmo de Floyd-Warshall, descrito na Seção 2.2.3, apenas substituindo-se a equação 2.1 pela seguinte equação:

$$w_k(i, j) = \min(w_{k-1}(i, j), \max(w_{k-1}(i, k), w_{k-1}(k, j))) \tag{2.2}$$

ALGORITMO 9: Algoritmo de Floyd-Warshall

Entrada: Matriz de adjacências $A_{n \times n}$ do grafo G, contendo os pesos das arestas
Saída: Os pesos das arestas de maior peso que fazem parte do caminho de i a j em T

```
1: para k ← 1 até n faça
2:    para i ← 1 até n faça
3:       para j ← 1 até n faça
4:          w(i, j) ← min(w(i, j), max(w(i, k), w(k, j)))
5:       fim para
6:    fim para
7: fim para
```

Como resultado, obtém-se o peso $w_{i,j}^{(N)}$ da aresta de maior peso que faz parte do caminho que liga i a j. Em seguida, a escolha das arestas da árvore geradora mínima pode ser feita em um passo simples de acordo com o Lema 1, ou seja, uma aresta (i, j) pertence a árvore geradora mínima se, e somente se, $w_{i,j} = w_{i,j}^{(N)}$.

2.5 Conclusão

Apesar das diferenças estruturais existentes entre todos os problemas em grafos descritos neste capítulo, mostramos que o fecho transitivo pode ser utilizado na solução dos mesmos. A execução do algoritmo de Warshall para computar o fecho transitivo pode ser considerada como uma operação de pré-processamento que disponibiliza um conjunto de informações em uma matriz de adjacências para a computação da busca em largura a estrutura do algoritmo de Warshall pode ser utilizada para computar os caminhos mais curtos e a árvore geradora mínima. Dessa forma, embora aparentemente os problemas sejam muito diferentes, mostramos que é possível descrever todos os algoritmos citados usando a mesma estrutura fundamental, baseada no algoritmo seqüencial de Warshall para o fecho transitivo.

Vale ressaltar que os tempos obtidos por estas implementações seqüenciais alternativas podem não ser satisfatórios. No entanto, com a paralelização de alguns dos métodos descritos neste capítulo podemos obter melhores resultados e, de forma mais rápida e eficiente, disponibilizar as informações para a resolução dos demais problemas relacionados. Além disso, com alguns dos resultados obtidos com as implementações paralelas para os problemas estudados, usando a estrutura fundamental do algoritmo de Warshall, se mostram mais eficientes que algumas outras implementações paralelas para o mesmo problema, tais como a busca em largura[15].

No capítulo seguinte, vamos detalhar alguns algoritmos paralelos para o fecho transitivo e problemas relacionados, começando pelo algoritmo PRAM para multiplicação de matrizes, apresentado por JáJá[15] que, segundo o autor, é uma importante ferramenta para o desenvolvimento de um algoritmo paralelo para resolver o problema do fecho transitivo, visto que os grafos dirigidos podem ser representados por sua matrizes de adjacências. Em seguida, apresentamos os algoritmos realísticos baseados no algoritmo de Cáceres *et al* [6] para o fecho transitivo no modelo BSP/CGM.

Capítulo 3

Algoritmos Paralelos

Neste capítulo, apresentamos o algoritmo paralelo no modelo PRAM descrito por JáJá [15] para encontrar o fecho transitivo de um grafo dirigido. Este algoritmo é baseado no algoritmo de multiplicação de matrizes, apresentado pelo mesmo autor.

Em seguida, apresentamos o algoritmo de Cáceres *et al*[6] para computar o fecho transitivo e suas aplicações, relacionando-o com algoritmos paralelos para o cálculo dos caminhos mais curtos, busca em largura e árvore geradora mínima. Para cada um dos problemas citados acima, são descritos os algoritmos e as suas respectivas complexidades.

3.1 Algoritmos Paralelos para o Fecho Transitivo

3.1.1 No Modelo PRAM

Um algoritmo paralelo para computar o fecho transitivo de um grafo dirigido no modelo PRAM é apresentado em [15]. Considere o problema de calcular o fecho transitivo de um grafo $G = (V, E)$ com n vértices e m arestas. Segundo JáJá [15], já que os grafos dirigidos podem ser representados por sua matrizes de adjacências, a computação do produto de matrizes passa a ser uma importante ferramenta para o desenvolvimento de um algoritmo paralelo para resolver o problema do fecho transitivo. Dessa forma, com o objetivo de facilitar a descrição do algoritmo paralelo para resolver o problema do fecho transitivo, vamos antes falar brevemente sobre o algoritmo PRAM para a multiplicação de matrizes.

Multiplicação de Matrizes

Seja $A = (a_{ij})$ e $B = (b_{ij})$ duas matrizes $m \times n$ e $n \times p$, respectivamente. Seja $C = AB$ o produto da matriz A pela matriz B; isto é, a posição (i, j) da matriz C é definida por $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, onde $1 \leq i \leq m$ e $1 \leq j \leq p$. JáJá [15] descreve um algoritmo para calcular todos os produtos $a_{ik}b_{kj}$ para todos i, k, j e, usando uma árvore binária balanceada, calcular cada c_{ij}. O algoritmo descrito possui complexidade de tempo paralelo $O(\log n)$, usando um total de $O(mnp)$ processadores sobre o modelo CREW PRAM, ou tempo $O(1)$, usando $O(mnp)$ processadores no modelo CRCW PRAM.

Algoritmo 10: Multiplicação de Matrizes no Modelo CREW PRAM

Entrada: Duas matrizes A e B, $n \times n$ armazenadas em uma memória compartilhada, onde $n = 2^k$. As variáveis localmente inicializadas são n e a tripla de índices (i, j, k), identificando o processador.

Saída: O produto $C = AB$ armazenado na memória compartilhada.

1: $C'(i, j, k) \leftarrow A(i, k) \cdot B(k, j)$
2: **para** $h \leftarrow 1$ **até** $\log n$ **faça**
3: **se** $(k \leq \frac{n}{2^h})$ **então**
4: $C'(i, j, k) \leftarrow C'(i, j, 2k - 1) + C'(i, j, 2k)$
5: **fim se**
6: **fim para**
7: **se** $(k = 1)$ **então**
8: $C(i, j) \leftarrow C'(i, j, 1)$
9: **fim se**

Observe que o algoritmo acima requer leitura concorrente, já que diferentes processadores podem ter acesso ao mesmo dado durante a execução do passo 1. Por exemplo, os processadores $P_{i,1,k}, P_{i,2,k}, \ldots, P_{i,n,k}$ acessam a posição $A(i, k)$ da matriz durante o passo 1. Portanto, este algoritmo executa em tempo paralelo $O(\log n)$, utilizando $O(n^3)$ processadores no modelo PRAM CREW.

Se considerarmos que, para o problema do fecho transitivo, utilizamos a matriz de adjacências de um grafo dirigido qualquer, então estas matrizes podem ser vistas como matrizes *booleanas* ou binárias e, portanto, a multiplicação delas pode ser definida sobre as operações lógicas **OU** e **E**. Dessa forma, a posição (i, j) da matriz resultante do produto de A por B corresponde à operação **OU** de n termos resultantes da operação **E** entre $a_{ik}b_{kj}$, onde $1 \leq k \leq n$. Cada operação **OU** pode ser calculada em tempo $O(1)$, usando um total de $O(n)$ processadores no modelo PRAM CRCW Comum. Portanto, considerando que as matrizes de entrada possuem dimensões $n \times n$, a multiplicação de matrizes binárias pode ser feita em tempo $O(1)$, usando $O(n^3)$ processadores.

Outra operação importante relacionada com a multiplicação de matrizes e que também é utilizada por JáJá no desenvolvimento do algoritmo PRAM para a computação do fecho transitivo é o cálculo de A^{2^s}, onde A é uma matriz $n \times n$ e s é um inteiro positivo. O método de repetição sucessivas de quadrados consiste de s iterações do cálculo de $B = B^2$, com $B = A$ inicialmente. Portanto, pode-se obter sucessivamente $B = A, B = A^2, B = A^4, \ldots,$ e finalmente, $B = A^{2^s}$.

A complexidade final do método de repetição sucessivas de quadrados depende do algoritmo utilizado para calcular o quadrado das matrizes. Em particular, este método possui complexidade de tempo $O(s)$, usando $O(sn^3)$ processadores no modelo PRAM CRCW Comum ou $O(s \log n)$, usando $O(sM(n))$ processadores no modelo PRAM CREW, onde $M(n)$ corresponde à complexidade do melhor algoritmo seqüencial que resolve o problema.

Agora, seja $G = (V, E)$ uma grafo dirigido representado pela sua matriz de adjacências B. Seja B^* a matriz de adjacências do fecho transitivo de G. Como já foi dito, já que os valores armazenados nestas matrizes são 0 ou 1, elas podem ser vistas como matrizes binárias. O seguinte teorema reduz a computação de B^* ao cálculo de uma potência da matriz de adjacências.

Teorema 1 *Seja B uma matriz de adjacências $n \times n$ de um grafo dirigido $G = (V, E)$. Então, a matriz de adjacências B^* do fecho transitivo de G é dada por $B^* = (I + B)^{2^{\lceil \log n \rceil}}$, onde I é a matriz identidade $n \times n$.*

Prova: [15].

Corolário 1 *O fecho transitivo de um grafo dirigido com n vértices pode ser obtido em tempo $O(\log n)$, usando $O(n^3 \log n)$ processadores no modelo PRAM CRCW Comum, ou em tempo $O(\log^2 n)$, usando $O(M(n) \log n)$ processadores no modelo PRAM CREW, onde $M(n)$ é o melhor algoritmo seqüencial conhecido para multiplicar duas matrizes $n \times n$.*

3.1.2 No Modelo BSP/CGM

Seja $D = (V, E)$ um grafo dirigido com um $|E| = m$ arestas e $|V| = n$ vértices. Seja $S \subseteq V$. Denotamos por $D(S)$ o grafo dirigido formado exatamente pelas arestas de D que possuem pelo menos um de seus extremos em S. Se A é um caminho em D, denotamos seu comprimento por $|A|$. Uma **extensão linear** de D é uma seqüência $\{v_1, \ldots, v_n\}$ de seus vértices, tal que $(v_i, v_j) \in E \Rightarrow i < j$.

Os passos seguintes definem o algoritmo paralelo no modelo BSP/CGM, apresentado por Cáceres *et al* [6] para computar o fecho transitivo de D,

ALGORITMO 11: Algoritmo BSP/CGM para Computar o Fecho Transitivo

Entrada: (1) Um grafo dirigido $D = (V, E)$, com $|V| = n$ vértices e $|E| = m$ arestas; (2) p processadores.

Saída: D^t, o fecho transitivo de D

(1) Encontre uma extensão linear L de D;

(2) Seja $S_0, \ldots, S_{p-1}$ uma partição de $V(D)$, cujas partes tenham cardinalidades tão iguais quanto possível, e onde cada S_j seja formado por vértices consecutivos em L. Para $j = 0, \ldots, (p-1)$, atribua os vértices de S_j ao processador p_j;

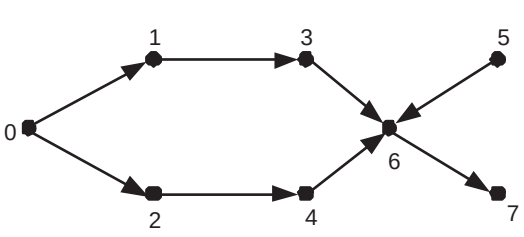
(3) Em paralelo, cada processador p_j seqüencialmente:

- (3.1) constrói o grafo dirigido $D(S_j)$ de D
- (3.2) computa o fecho transitivo $D^t(S_j)$ de $D(S_j)$
- (3.3) inclui em D as arestas $D^t(S_j) \backslash D(S_j)$

(4) Após todos os processadores terem completado o passo (3), verifique se $D^t(S_j) = D(S_j)$, para todos os processadores p_j. Se verdadeiro, o algoritmo é finalizado e D é o fecho transitivo do grafo dirigido de entrada. Caso contrário, vá para o passo (3).

usando p processadores, onde $1 \leq p \leq n$. Este algoritmo utiliza a mesma estrutura do algoritmo seqüencial de Warshall, apresentado na seção 2.1.1.

A entrada do Algoritmo 11 consiste de um arquivo contendo as informações necessárias que representem a matriz de adjacências de um grafo dirigido qualquer. Para facilitar a explicação de cada uma das etapas do algoritmo, assuma como exemplo que o desenho do grafo de entrada é mostrado na Figura 3.1(a). Para este grafo, o arquivo de entrada deve conter as informações contidas na Figura 3.1(b).



(a)

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

(b)

Figura 3.1: (a) Um exemplo de grafo dirigido de entrada e (b) sua matriz de adjacências.

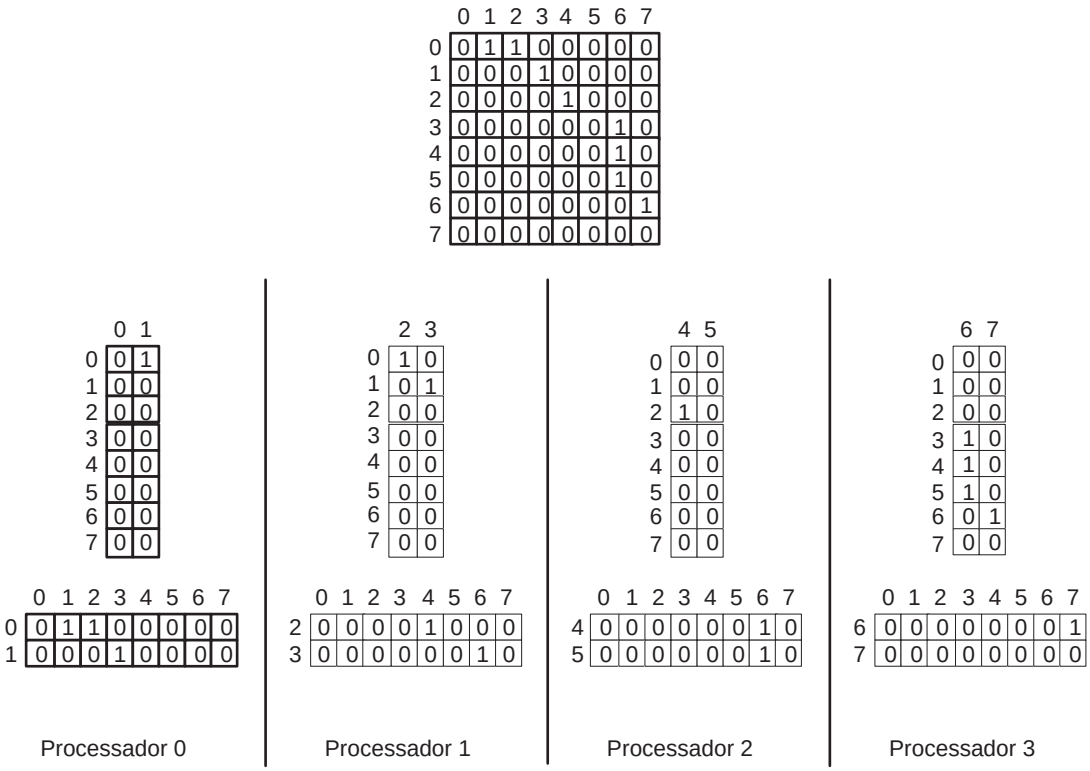


Figura 3.2: Um exemplo do particionamento e da distribuição das submatrizes entre 4 processadores.

O algoritmo proposto por Cáceres *et al* [6] usa um algoritmo seqüencial $O(mn)$. Vamos apresentar uma versão (mais orientada para implementação) que é baseada no algoritmo de Warshall. Na nossa proposta, assumimos que o grafo de entrada já é uma extensão linear de $D = (V, E)$.

No algoritmo 12, a entrada é uma matriz de adjacências A, particionada em p submatrizes de linhas e colunas que serão distribuídas entre os p processadores da seguinte forma: cada processador p_i recebe duas submatrizes, $A[\frac{in}{p}..\frac{(i+1)n}{p} - 1][0..n-1]$ e $A[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p} - 1]$. A Figura 3.2 é um exemplo do particionamento da matriz da Figura 3.1(b) entre $p = 4$ processadores. Na primeira rodada, cada processador p_j computa o fecho transitivo relativo às duas submatrizes, usando o algoritmo de Warshall.

Após a primeira rodada, cada processador verifica se alguma das arestas geradas necessita ser enviada a outros processadores. É, então, realizada uma rodada de comunicação. Como veremos, após $O(\log p)$ rodadas de computação/comunicação, o algoritmo 12 computa o fecho transitivo de $D = (V, E)$.

ALGORITMO 12: Algoritmo de Warshall Modificado

Entrada: (1) p processadores $p_0, p_1, \ldots, p_{p-1}$; (2) Uma matriz $n \times n$, A que representa a matriz de adjacências de um grafo dirigido, distribuída entre os p processadores da seguinte forma: cada processador p_i recebe duas submatrizes, $A[\frac{in}{p}..\frac{(i+1)n}{p}-1][0..n-1]$ e $A[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p}-1]$.

Saída: A matriz D que representa o fecho transitivo do grafo de entrada.

1: **repita**
2: **para** $k = \frac{p_i n}{p}$ **até** $\frac{(p_i+1)n}{p-1}$ **faça**
3: **para** $i = 0$ **até** $n-1$ **faça**
4: **para** $j = 0$ **até** $n-1$ **faça**
5: **se** *rodadas* > 1 **então**
6: *receba arestas*
7: **fim se**
8: **se** $A[i][k] = 1$ **e** $A[k][j] = 1$ **então**
9: $A[i][j] = 1$
10: **fim se**
11: **se** *rodadas* $< \log p$ **então**
12: *envie arestas*
13: **fim se**
14: **fim para**
15: **fim para**
16: **fim para**
17: **até** *rodadas* $= \log p$

A corretude do algoritmo é assegurada pelo seguinte teorema:

Teorema 2 *O algoritmo 11 computa corretamente o fecho transitivo de um grafo dirigido de entrada. Além disso, ele requer, no máximo,* $1 + \lceil \log p \rceil$ *iterações do Passo (3).*

Prova: [6] Seja D_i o grafo dirigido D obtido após a i-ésima iteração do Passo 3. Seja D_0 o grafo dirigido de entrada e D_i^t o fecho transitivo de D_i, $i = 0, 1, \ldots$. Visto que $D_i(S_j)$ é o subgrafo de D_i, segue que todas as arestas do fecho transitivo $D_i^t(S_j)$ de $D_i(S_j)$ também pertencem a D_i^t, e portanto a D_0^t. Conseqüentemente, alme de mostrar que o algoritmo computa corretamente o fecho transitivo D_0^t de D_0, é suficiente mostrar que toda aresta de D_0^t é também uma aresta de algum $D_i^t(S_j)$.

Dessa forma, seja $(v, w) \in E(D_0^t)$. Nós mostramos que $(v, w) \in E(D_i^t(S_j))$, para algum i, j. Como (v, w) é uma aresta de D_0^t, D_0 contém um caminho $z_1, \ldots, z_l$, de $v = z_1$ até $w = z_l$. Para cada k, $1 \leq k \leq l$, denote por $P(z_k)$ o processador para o qual z_k é enviado. Como o envio de vértices para os processadores obedece à ordem de uma extensão linear, segue que $P(z_1), \ldots, P(z_l)$ é uma seqüência não-decrescente. Conseqüentemente, após

completar a primeira iteração do Passo 3, nós sabemos que D_1 contém um caminho $v-w$, denotado por A_1, formado unicamente por (um subconjunto de) vértices de $z_1, \ldots, z_l$, enviados a processadores distintos. Portanto, $|A_1| \leq p$. Se $|A_1| = 1$, então $(v, w) \in E(D_1^t(S_j))$, implicando na corretude do algoritmo. Caso contrário, sejam z'_{k-1}, z'_k, z'_{k+1} três vértices consecutivos em A. Seja $P(z'_k) = j$. Conseqüentemente, $(z'_{k-1}, z'_k), (z'k, z'_{k+1}) \in E(D_1(S_j))$. Isto significa que ao final da segunda iteração do Passo 3, $(z'_{k-1}, z'_{k+1}) \in E(D_2)$. Conseqüentemente, D_2 contém um caminho $v - w$, denotado por A_2, formado por um subconjunto de vértices de A_1, satisfazendo $|A_2| = \lceil |A_1|/2 \rceil$. Por indução, segue que $|A_{\lceil \log |A_1| + 1 \rceil}| = 1$, isto é, $(v, w) \in E(D^t_{\lceil \log |A_1| + 1 \rceil}(S_j))$, como requerido. Além disso, não mais do que $1 + \lceil \log p \rceil$ iterações do Passo 3 são necessárias. □

Corolário 2 *O algoritmo 12 computa corretamente o fecho transitivo de um grafo dirigido de entrada. Além disso, ele requer, no máximo,* $1 + \lceil \log p \rceil$ *rodadas de computação/comunicação.*

Basicamente, o algoritmo 12 consiste de, no máximo, $1 + \lceil \log p \rceil$ computações paralelas de um algoritmo seqüencial de Warshall [36] para o fecho transitivo, cuja complexidade é $O(\frac{n^3}{p})$. Já que, no máximo, $1 + \lceil \log p \rceil$ iterações são executadas, a complexidade total é $O((\log p)(\frac{n^3}{p}))$.

O número de rodadas de comunicação do algoritmo de Cáceres *et al* [6] pode ser superior a $O(\log p)$ se o grafo dirigido de entrada não estiver rotulado de acordo com uma extensão linear[4]. Vale observar que, na implementação paralela usando a estrutura do algoritmo de Warshall, mesmo gerando grafos que não correspondam a uma extensão linear, o fecho dos mesmos pôde ser obtido em $O(\log p)$ rodadas. No caso de nosso exemplo, a extensão linear é o próprio grafo da Figura 3.1.

3.2 Aplicações do Fecho Transitivo

3.2.1 Caminhos Mais Curtos

Dado um grafo dirigido G com n vértices e com pesos w_{ij} nas arestas, considere o problema de computar o caminho de menor peso, ou simplesmente, o caminho mais curto, dirigido de i para j, para todo par de vértices i e j. Além de assegurar que o caminho mais curto existe entre cada par de vértices e que G não contém circuitos dirigidos com pesos negativos, o grafo dirigido com os pesos associados às arestas é representado por uma matriz de pesos w_{ij}, dada por:

$$w_{ij} = \begin{cases} w(v_i, v_j) & \text{se } v_iv_j \in E \\ \infty & \text{se } i \neq j \text{ e } v_iv_j \notin E \\ 0 & \text{se } i = j \end{cases}$$

Assim como no caso dos algoritmos seqüenciais, descritos no Capítulo 2, o problema dos caminhos mais curtos pode ser resolvido por um algoritmo paralelo cuja estrutura é muito semelhante ao algoritmo do fecho transitivo, descrito na Seção 3.1.2. A diferença básica entre estes algoritmos reside na diferença entre os valores dos conjuntos de dados de entrada e, conseqüentemente, entre as operações realizadas sobre estes valores. Enquanto, no caso do fecho transitivo, a entrada é composta pela matriz de adjacências, contendo somente valores binários sobre os quais são realizadas as operações lógicas de **e** e **ou**, na computação dos menores caminhos, temos como entrada a matriz de pesos, sobre as quais são realizadas as operações de **soma** e **mínimo**. O Algoritmo 13 corresponde ao algoritmo no modelo BSP/CGM para o cálculo dos menores caminhos.

ALGORITMO 13: Caminhos Mais Curtos

Entrada: (1) p processadores $p_0, p_1, \ldots, p_{p-1}$; (2) Uma matriz $n \times n$, W, cujo elemento w_{ij} representa o peso associados a aresta que liga o vértice i ao vértice j, distribuída entre os p processadores da seguinte forma: cada processador p_i recebe duas submatrizes, $W[\frac{in}{p}..\frac{(i+1)n}{p} - 1][0..n-1]$ e $W[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p} - 1]$.

Saída: A matriz D, onde d_{ij} representa o custo do caminho mais curto entre os vértices i e j.

```
1: repita
2:   para k = p_i n / p até ((p_i + 1)n / p) − 1 faça
3:     para i = 0 até n − 1 faça
4:       para j = 0 até n − 1 faça
5:         se rodadas > 1 então
6:           receba arestas
7:         fim se
8:         W[i][j] = min{W[i][j], (W[i][k] + W[k][j])}
9:         se rodadas < log p então
10:          envie arestas
11:        fim se
12:      fim para
13:    fim para
14:  fim para
15: até rodadas = log p
```

Observe que, após a execução do Algoritmo 13, o fecho transitivo D pode ser obtido diretamente da matriz W que armazena os caminhos mais curtos entre todos os pares de vértices do grafo. Isso pode ser feito simplesmente pela aplicação da seguinte regra:

$$D[i][j] = \begin{cases} 1 & \text{se } W[i][j] < \infty \\ 0 & \text{se } W[i][j] = \infty \end{cases}$$

Dessa forma, pode-se concluir que o problema de computar os caminhos mais curtos entre todos os pares de vértices i e j é mais geral que o problema de computar o fecho transitivo. Além disso, devido à semelhança entre as estruturas dos algoritmos, assim como para o fecho transitivo, a complexidade total do algoritmo para computar os caminhos mais curtos é igual $O((\log p)(\frac{n^3}{p}))$.

3.2.2 Busca em Largura

ALGORITMO 14: Busca em Largura

Entrada: (1) p processadores $p_0, p_1, \ldots, p_{p-1}$; (2) Uma matriz $n \times n$, A que representa a matriz de adjacências de um grafo dirigido, distribuída entre os p processadores da seguinte forma: cada processador p_i recebe duas submatrizes, $A[\frac{in}{p}..\frac{(i+1)n}{p}-1][0..n-1]$ e $A[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p}-1]$; (3) Um vértice raiz r.

Saída: Árvore geradora T de busca em largura do grafo.

```
1: Calcule CAMINHOS_MAIS_CURTOS(A)
2: se r está em p_i então
3:    broadcast as distâncias da raiz
4: fim se
5: para i = p_i n / p até (p_i+1)n / p−1 faça
6:    para j = 0 até n − 1 faça
7:       se dist[j] = dist[i] − 1 então
8:          T[i][j] ← 1
9:          T[j][i] ← 1
10:      fim se
11:   fim para
12: fim para
```

Dado um grafo G conexo, não dirigido e sem pesos nas arestas, uma árvore geradora de busca em largura T de G é uma árvore geradora para a qual cada caminho de um vértice qualquer até o vértice raiz de T é o caminho mais curto em G.

Deve-se observar que o problema de computar uma árvore geradora de busca em largura de um grafo G é muito semelhante ao problema de computar os caminhos mais curtos em G. Neste caso particular, é atribuído peso 1 para cada aresta $e \in E$ e peso ∞ para cada aresta $a \notin E$, onde E corresponde ao conjunto de arestas de G. O peso do caminho mais curto da raiz até um determinado vértice é simplesmente a distância do vértice até a raiz do

grafo. O algoritmo BSP/CGM para computar a árvore geradora de busca em largura corresponde a paralelização do algoritmo seqüencial descrito na Seção 2.3.2.

Após computar os caminhos mais curtos entre todos os pares de vértices de G, uma árvore de busca em largura pode ser encontrada através da seleção das arestas que ligam um vértice a uma distância i da raiz a um outro vértice a uma distância $i-1$ para todo $i > 0$. Como já foi dito, a complexidade do algoritmo para encontrar os caminhos mais curtos é igual $O((\log p)(\frac{n^3}{p}))$. A complexidade dos passos 5 a 12, que selecionam as arestas da árvore geradora, é igual a $O((\frac{n^2}{p}))$. A complexidade total do algoritmo é, portanto, igual a $O((\log p)(\frac{n^3}{p}) + (\frac{n^2}{p}))$ ou, simplesmente, $O((\log p)(\frac{n^3}{p}))$.

3.2.3 Árvore Geradora Mínima

ALGORITMO 15: Árvore Geradora Mínima

Entrada: (1) p processadores $p_0, p_1, \ldots, p_{p-1}$; (2) Uma matriz $n \times n$, W, cujo elemento w_{ij} representa o peso associados a aresta que liga o vértice i ao vértice j, distribuída entre os p processadores da seguinte forma: cada processador p_i recebe duas submatrizes, $W[\frac{in}{p}..\frac{(i+1)n}{p} - 1][0..n-1]$ e $W[0..n-1][\frac{in}{p}..\frac{(i+1)n}{p} - 1]$.

Saída: A matriz T, onde t_{ij} armazena o peso da aresta de maior custo no caminho que liga o vértice i ao vértice j.

1: **repita**
2: **para** $k = \frac{p_i n}{p}$ **até** $\frac{(p_i+1)n}{p-1}$ **faça**
3: **para** $i = 0$ **até** $n-1$ **faça**
4: **para** $j = 0$ **até** $n-1$ **faça**
5: **se** *rodadas* > 1 **então**
6: *receba mensagens*
7: **fim se**
8: $W[i][j] = \min\{W[i][j], \max(W[i][k], W[k][j])\}$
9: **se** *rodadas* $< \log p$ **então**
10: *envie mensagens*
11: **fim se**
12: **fim para**
13: **fim para**
14: **fim para**
15: **até** *rodadas* $= \log p$
16: Selecione as arestas de T, a partir de W.

Dado o Lema 1, demonstrado na Seção 2.4.3, desenvolver um algoritmo paralelo no modelo BSP/CGM para computar a árvore geradora mínima de um grafo dirigido com peso nas arestas se torna um problema mais simples. Na verdade, este algoritmo pode possuir a mesma estrutura do algoritmo

apresentado na Seção 3.2.1, utilizado para computar os caminhos mais curtos de um grafo.

Para computar a árvore geradora mínima, leva-se em consideração o peso da aresta mais pesada ao invés da soma dos pesos das arestas do caminho. Neste caso, assumindo a existência da matriz de pesos W, também utilizada na computação dos caminhos mais curtos, deve-se computar cada w_{ij} da seguinte forma:

$$w_k(i,j) = \min(w_{k-1}(i,j), \max(w_{k-1}(i,k) + w_{k-1}(k,j))) \tag{3.1}$$

onde k representa o número de interações realizadas.

O Passo 16 do Algoritmo 15 pode ser feita em um passo simples através da seleção de uma aresta $e = (i,j)$ para pertencer a árvore geradora mínima se, e somente se, $W_0[i][j] = W_n[i][k]$, onde $W_0[i][j]$ corresponde aos pesos de cada uma das arestas do grafo de entrada, individualmente, e $W_n[i][k]$ corresponde aos valores armazenados após a execução do algoritmo.

3.3 Conclusão

Como já foi dito, apesar de se tratarem de problemas distintos, o fecho transitivo, os caminhos mais curtos, a busca em largura e a árvore geradora mínima podem ser resolvidos usando a mesma estrutura fundamental do algoritmo de Cáceres *et al*[6]. Além disso, esta idéia possibilita a implementação de algoritmos que podem produzir resultados melhores que outros algoritmos paralelos, em outros modelos de computação paralela, anteriormente propostos para os mesmos problemas. Este é o caso da busca em largura.

No entanto, vale lembrar que este trabalho possui um foco experimental e não estamos interessados em descrever algoritmos melhores, mas sim comprovar a utilidade do modelo BSP/CGM para a implementação do algoritmo de Cáceres *et al*[6]. No próximo capítulo, descreveremos os resultados obtidos das implementações.

Capítulo 4

Implementações

Neste capítulo, apresentamos os resultados experimentais obtidos da implementação do algoritmo BSP/CGM para o fecho transitivo, descrito em [6], e das implementações paralelas para cada um dos algoritmos desenvolvidos para problemas relacionados em grafos, usando a mesma estrutura fundamental. Todas as implementações descritas aqui são baseadas no algoritmo de Cáceres *et al* [6] e nas idéias descritas no Capítulo 3, para a implementação destes algoritmos usando o modelo BSP/CGM de computação paralela. Os resultados experimentais apresentados demonstram a utilidade do uso de um modelo de granularidade grossa na resolução de problemas que manipulam um grande volume de dados.

4.1 Ambiente de Implementação

Os algoritmos descritos foram implementados na linguagem C e a troca de mensagens foi realizada através de funções da biblioteca MPI. O MPI foi o escolhido por ser uma das bibliotecas que possuem interface com a linguagem, dando um suporte mais adequado para a implementação de algoritmos paralelos no modelo BSP/CGM. Suas funções oferecem facilidades de comunicação e sincronização, assim como rotinas básicas para *broadcast*, comunicação coletiva e computação de vetores.

As implementações foram executadas no ***Beowulf*** do Instituto de Computação da UNICAMP, que consiste de um *cluster* de PCs com 66 processadores Pentium 450MHz, com 256MB de memória, interligados por uma rede de interconexão de 100Mbits.

O primeiro *Beowulf* foi desenvolvido em 1994 pelo CESDIS (*Center for Excellence in Space Data and Information Science*) que é uma divisão da USRA (*University Space Research Association*), localizada em Greenbelt

Maryland. Os computadores que compunham inicialmente a máquina *Beowulf* consistiam de 16 processadores DX4 ligados a uma rede Ethernet com canais dedicados. Alguns algoritmos de processamento paralelo foram implementados com sucesso nesta estrutura.

As circunstâncias históricas que levaram ao sucesso dessa forma de computação são várias. Entre elas, podemos citar a popularização dos PCs, a redução dos custos de componentes de hardware, como memórias e processadores; o desenvolvimento das redes de comunicação digital e a disponibilidade de redes de alta velocidade; o surgimento dos sistemas operacionais e programas gratuitos de livre acesso como o GNU-Linux e, finalmente, o desenvolvimento de bibliotecas de processamento paralelo como PVM e MPI que podem ser executadas em diversos ambientes.

4.2 Descrição das Implementações

Os resultados mostrados nesta seção medem os tempos de execução das implementações paralelas para os problemas do fecho transitivo, dos caminhos mais curtos, da busca em largura e da árvore geradora mínima, usando a mesma estrutura proposta por Cáceres *et al*[6]. Na implementação, foi utilizada uma versão paralela do algoritmo de fecho transitivo de Warshall. Os tempos obtidos não envolvem os gastos com a leitura e a distribuição inicial dos dados e o envio dos resultados finais para o processo pai. Todos os códigos fontes encontram-se organizados no Apêndice A.

A principal dificuldade foi encontrar uma forma de evitar o envio de arestas duplicadas. A solução encontrada foi o uso de uma matriz auxiliar que marca as arestas enviadas, evitando que elas sejam enviadas novamente, exceto quando for realmente necessário. Outra questão diz respeito ao uso das rotinas de comunicação coletiva do MPI que oferecem maior comodidade para o programador, além de contribuírem para a produção de um código mais legível. No entanto, é importante observar que a implementação de cada uma destas rotinas é transparente para o programador e, algumas vezes, resultados melhores podem ser obtidos através de implementações explícitas das rotinas de comunicação. Por último, o uso da função `realloc` para o redimensionamento do *buffer* de envio dinâmico também justifica-se pela comodidade de seu uso. Apesar disso, vale lembrar que a reorganização dos dados na memória pode aumentar muito o tempo de execução do algoritmo quando executado sobre grafos densos. Mesmo com todas estas questões importante que envolvem os detalhes de implementação, os resultados obtidos pela implementação do algoritmo de Cáceres *et al*[6] são melhores que os descritos em [24, 25].

4.2.1 Grafos de Entrada

Os grafos usados em nossos testes foram gerados de forma aleatória por um gerador de grafos, cujo código se encontra no Apêndice A. Os grafos gerados podem ser esparsos ou densos, dirigidos ou não e com ou sem pesos nas arestas. Em todos os teste realizados, utilizamos grafos densos, já que um pequeno número de arestas poderia ocasionar uma grande redução dos tempos de comunicação, prejudicando a obtenção dos tempos totais a partir do modelo utilizado.

4.3 Resultados dos Testes Realizados

Os resultados mostrados nesta seção consideram o tempo gasto com computação local e comunicação, sem considerar o pré-processamento, envolvendo a leitura, a preparação e a distribuição das informações para os processadores.

Além disso, neste trabalho, o tempo de execução seqüencial (T_s) não corresponde ao tempo da melhor implementação seqüencial, mas sim ao tempo de execução da implementação paralela, executada sobre um único processador.

4.3.1 Fecho Transitivo

Para o fecho transitivo foram considerados grafos dirigidos quaisquer, representados por suas matrizes de adjacências. Foram gerados grafos de tamanhos distintos, contendo 240, 256, 480, 512, 960 e 1920 vértices e uma quantidade de arestas aleatória. Os resultados são mostrados na Tabela 4.1.

	Tamanho da Matriz (quantidade de vértices)					
P	240×240	256×256	480×480	512×512	960×960	1920×1920
1	4.421866	5.395929	35.900598	46.592329	299.549277	2332.344211
2	2.927146	3.559008	23.179427	29.330198	191.742761	1475.260064
4	1.832646	2.179686	13.868741	17.624690	110.707855	889.141083
8	1.181770	1.928428	8.662276	10.628476	64.969344	516.801934
16	1.058808	1.049598	5.928634	8.027607	40.798095	323.153385
30	1.271521	—	—	—	—	—
32	—	2.005186	4.822010	5.593857	31.761969	288.520773
40	—	—	6.502613	—	—	—
60	1.603905	—	9.533927	—	—	—
64	—	2.520606	—	9.947398	29.599836	186.137676

Tabela 4.1: Tempos obtidos pelo fecho transitivo.

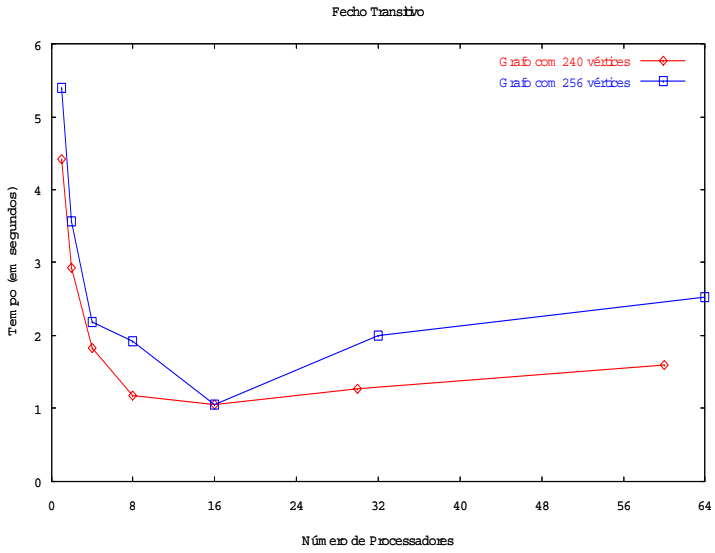


Figura 4.1: Tempos obtidos para a execução do fecho transitivo sobre grafos com 240 e 256 vértices.

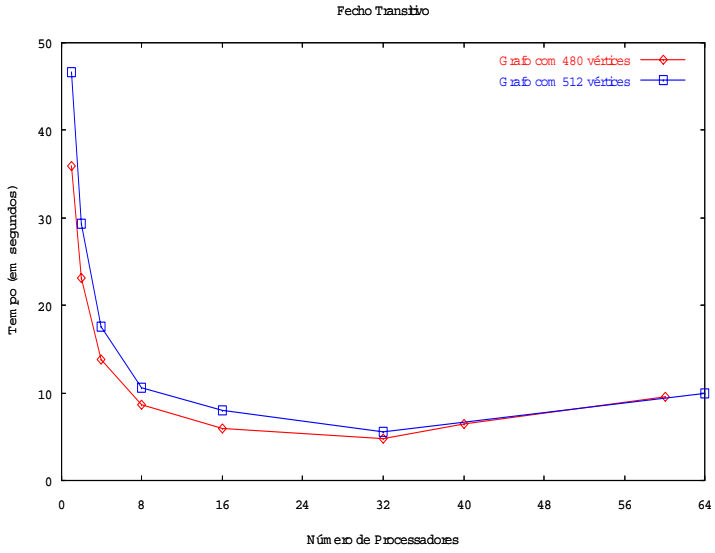


Figura 4.2: Tempos obtidos para a execução do fecho transitivo sobre grafos com 480 e 512 vértices.

Na Figura 4.1, podemos observar o comportamento do tempo de execução do algoritmo sobre grafos menores quando aumentamos o número de processadores. Observe que o tempo diminui até um ponto mínimo e, em seguida, volta a aumentar. A razão que justifica o formato da curva é que, quando aumentamos o número de processadores envolvidos, o tempo de com-

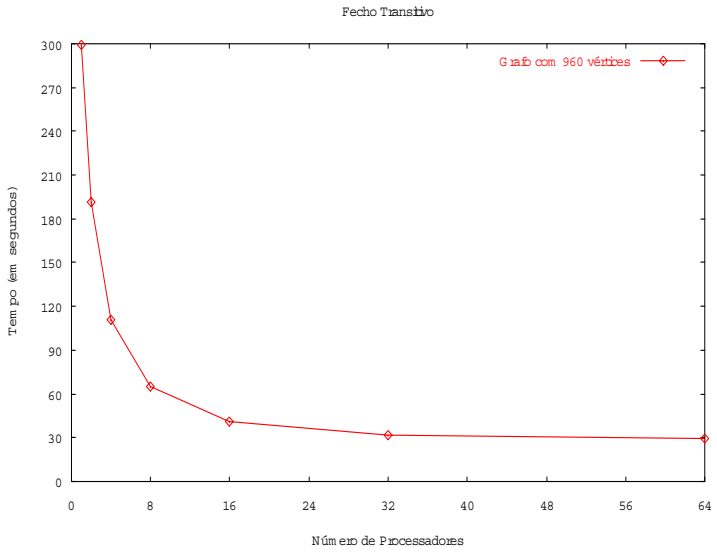


Figura 4.3: Tempos obtidos para a execução do fecho transitivo sobre um grafo com 960 vértices.

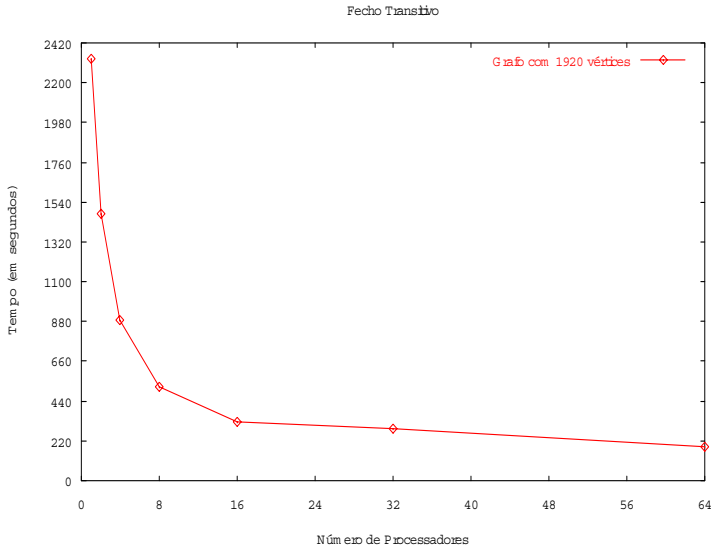


Figura 4.4: Tempos obtidos para a execução do fecho transitivo sobre um grafo com 1920 vértices.

putação diminui enquanto o tempo de comunicação aumenta, pois à medida que aumentamos o número de processadores, aumentamos também o número de rodadas de comunicação necessárias. Como estamos lidando com grafos pequenos, as curvas podem, algumas vezes, apresentar um comportamento

estranho. Apesar disso, é fácil observar que existe um número de processadores "ideal", para o qual é obtido o tempo mínimo (o ponto em que $p = 16$). A partir daí, a comunicação sobrepõe o tempo de computação.

Na Figura 4.2 já estamos lidando com grafos maiores. Observe que o ponto considerado "ideal"está se afastando do eixo y. Isto significa que, à medida que aumentamos o tamanho dos grafos, a comunicação demora mais para sobrepor os tempos de computação.

Nas Figuras 4.4 e 4.3 conseguimos observar uma queda brusca nos tempos de execução e, em seguida, a estabilização da curva, representando o momento em que computação e comunicação começam a se equilibrar.

4.3.2 Caminhos Mais Curtos

Para o caminhos mais curtos foram considerados grafos dirigidos com pesos nas arestas, representados por suas matrizes de adjacências. Foram gerados grafos de tamanhos distintos, contendo 256, 512, 768, 1024, 1536 e 1920 vértices e uma quantidade de arestas aleatória. Os resultados são mostrados na Tabela 4.2.

	Tamanho da Matriz (quantidade de vértices)					
P	256×256	512×512	768×768	1024×1024	1536×1536	1920×1920
1	7.845035	62.531597	188.849117	513.884804	1702.284128	3352.862994
2	8.642363	62.036187	180.841975	499.797966	—	—
4	7.852381	50.212808	147.146103	440.317591	—	—
8	7.491179	44.583123	136.261825	439.093990	—	—
16	6.819917	40.941487	120.477691	423.972134	—	—
32	8.539449	40.198423	115.776391	393.431537	—	—
64	14.252890	40.784190	101.619750	280.893185	—	—

Tabela 4.2: Tempos obtidos pelo caminhos mais curtos.

O gráfico da Figura 4.5 mostra o comportamento do tempo em grafos com um número menor de vértices. Observe que em grafos com 256 vértices, fica difícil analisarmos o comportamento deste algoritmo, já que o tempo gasto com a comunicação acaba, na maior parte do tempo, sendo maior que a computação local, aumentando os tempos de execução. Já com uma instância maior (512 vértices), a análise fica bem mais fácil, demonstrando o ganho de tempo obtido com o aumento do número de processadores. Novamente, vale observar que, a partir de um certo número de processadores, a comunicação

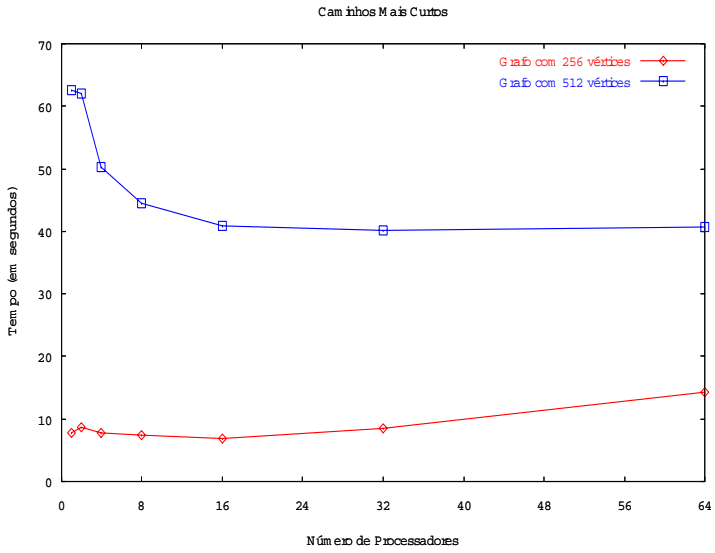


Figura 4.5: Tempos obtidos para a execução do caminhos mais curtos sobre grafos com 256 e 512 vértices.

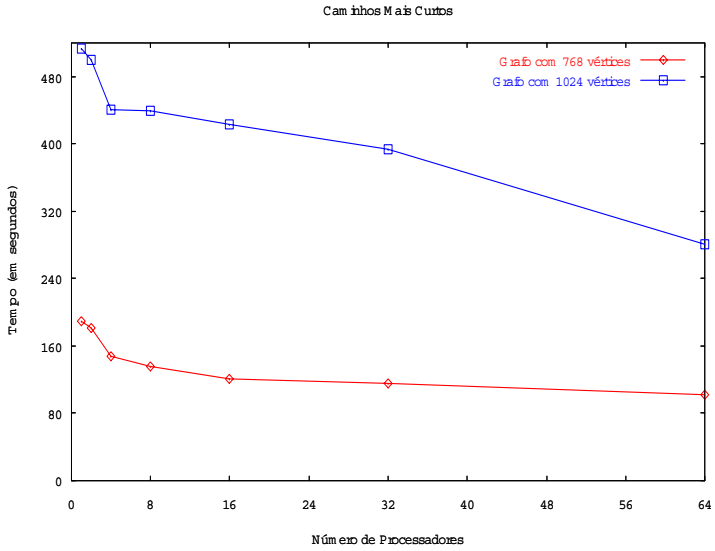


Figura 4.6: Tempos obtidos para a execução do caminhos mais curtos sobre grafos com 768 e 1024 vértices.

se sobrepõe à computação, determinando o tempo de execução total do algoritmo. Na Figura 4.6, os ganhos ficam mais evidentes, pois os tamanhos das instâncias são maiores.

Vale observar que para grafos com 1536 e 1920 vértices não foi possível obter o tempo usando 2 ou mais processadores devido às limitações de memória das máquinas utilizadas.

4.3.3 Busca em Largura

Para a busca em largura foram considerados grafos não dirigidos sem pesos nas arestas, representados por suas matrizes de adjacências. Foram gerados grafos de tamanhos distintos, contendo 256, 512, 768, 1024, 1280 e 1536 vértices e uma quantidade de arestas aleatória. Os resultados são mostrados na Tabela 4.3.

	Tamanho da Matriz (quantidade de vértices)					
P	256×256	512×512	768×768	1024×1024	1280×1280	1536×1536
1	7.757786	50.277371	207.011170	470.778300	967.118080	1652.871317
2	7.609814	48.931065	185.123847	422.170187	843.740241	—
4	6.657822	38.563992	137.554280	357.302964	687.311701	—
8	6.044066	34.776200	101.714795	307.019435	659.361218	—
16	7.070756	30.665694	83.075097	306.377876	638.869091	—
32	8.815327	34.465701	85.333302	338.654184	803.632251	—
64	12.058557	35.554484	76.069719	233.045812	710.245165	—

Tabela 4.3: Tempos obtidos pela busca em largura.

As Figuras 4.7, 4.8 e 4.9 mostram os tempos obtidos para grafos pequenos (256 vértices). Nesta instância, o aumento no número de processadores não melhora o desempenho do algoritmo. No entanto, para grafos maiores, a partir de 512 vértices, observamos que o desempenho do algoritmo passa a ser melhor com o aumento no número de processadores. Nestas instâncias, como temos grafos maiores, o tempo gastos com a computação local sobrepõe a comunicação entre os processadores.

A exemplo dos caminhos mais curtos, para grafos com 1536 vértices não foi possível obter o tempo usando 2 ou mais processadores devido às limitações de memória das máquinas utilizadas.

4.3.4 Árvore Geradora Mínima

Para a árvore geradora mínima foram considerados grafos não dirigidos com pesos nas arestas, representados por suas matrizes de adjacências. Foram

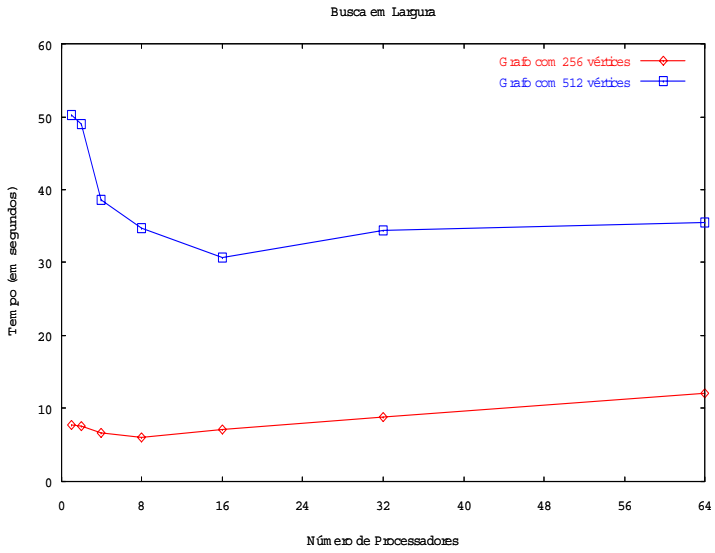


Figura 4.7: Tempos obtidos para a execução da busca em largura sobre grafos com 256 e 512 vértices.

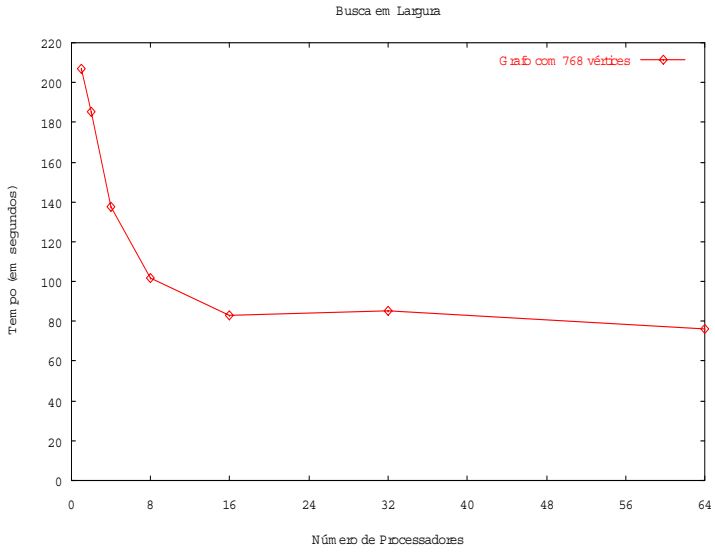


Figura 4.8: Tempos obtidos para a execução da busca em largura sobre um grafo com 768 vértices.

gerados grafos de tamanhos distintos, contendo 256, 512, 768 e 1024 vértices e uma quantidade de arestas aleatória. Os resultados são mostrados na Tabela 4.4.

Na Figura 4.10, são mostrados os tempos de execução para grafos me-

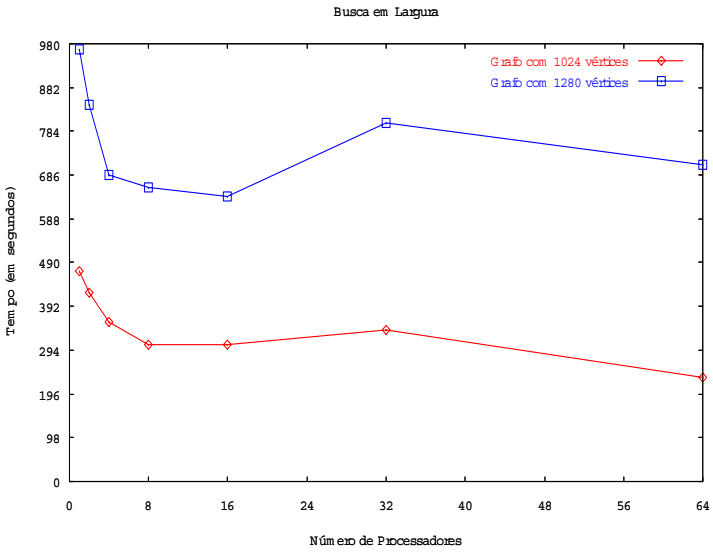


Figura 4.9: Tempos obtidos para a execução da busca em largura sobre grafos com 1024 e 1280 vértices.

nores. Considerando instâncias de 256 vértices, o aumento no número de processadores não melhora o desempenho do algoritmo. Para grafos maiores, a partir de 512 vértices, até certo ponto, a computação local sobrepõe a comunicação, determinando o tempo de execução total do algoritmo. No entanto, o ganho de desempenho é melhor observado na Figura 4.11, que mostra os resultados obtidos sobre grafos com 768 e 1024 vértices.

	Tamanho da Matriz (quantidade de vértices)			
P	256×256	512×512	768×768	1024×1024
1	6.174234	49.292222	161.642229	390.017599
2	6.113953	43.237032	139.769591	345.623904
4	5.480300	34.665256	105.388723	274.263012
8	4.731710	27.560711	75.044037	234.201602
16	4.271173	22.975276	62.178564	210.376178
32	5.805975	26.178214	60.030282	177.475328
64	14.477365	28.382323	62.492045	144.856295

Tabela 4.4: Tempos obtidos pela árvore geradora mínima.

A exemplo dos dois casos anteriores, para grafos maiores não foi possível obter o tempo usando 2 ou mais processadores devido às limitações de memória das máquinas utilizadas.

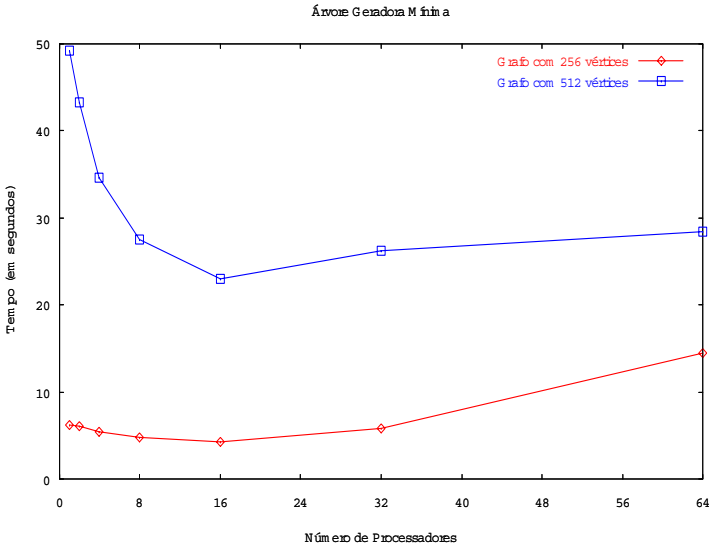


Figura 4.10: Tempos obtidos para a execução da árvore geradora mínima sobre grafos com 256 e 512 vértices.

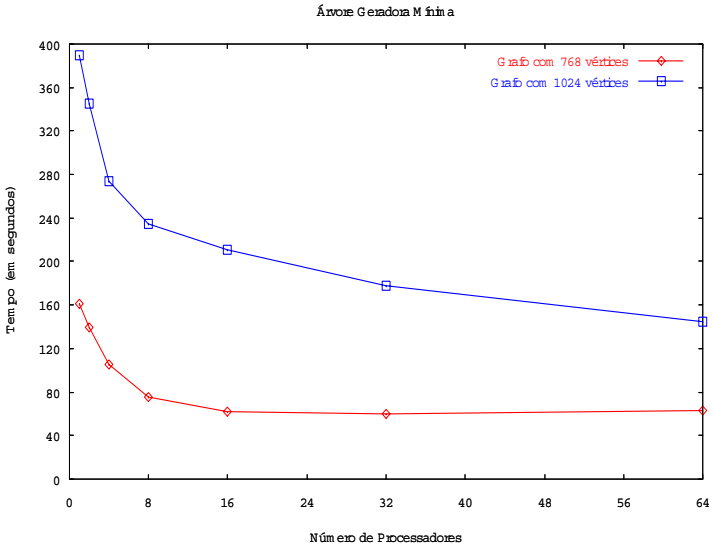


Figura 4.11: Tempos obtidos para a execução da árvore geradora mínima sobre grafos com 768 e 1024 vértices.

4.4 Conclusão

Pelos resultados obtidos sobre os teste realizados, podemos observar que, de fato, a comunicação afeta bastante o desempenho do algoritmo, de tal forma que, em alguns casos, o aumento no número de processadores acaba

aumentando o tempo total de execução.

A diferença entre os tempos de implementações distintas para grafos de mesmo tamanho, deve-se, principalmente, ao fato do processamento realizado após a computação do fecho transitivo e ao tamanho das mensagens. Além disso, a quantidade de arestas na matriz de adjacências também podem determinar o desempenho das implementações. Todos os testes acima, foram realizados sobre grafos densos gerados aleatóriamente.

Capítulo 5

Conclusão

No final dos anos 80, o desenvolvimento de algoritmos paralelos para o modelo PRAM foi bastante grande. Infelizmente, os resultados teóricos obtidos não foram observados nas implementações nas máquinas existentes. Nos anos 90, surgem os modelos realísticos BSP, LogP e CGM. Os algoritmos desenvolvidos nesses modelos, quando implementados em máquinas reais, apresentam resultados significativos ao utilizar-se mais de um processador.

Se comparadas a alguns resultados previamente obtidos, como os apresentados por Pagourtzis *et al* em [24, 25], a implementação do algoritmo do fecho transitivo, proposto por Cáceres *et al* [6], mostra resultados melhores, comprovando a utilidade do modelo BSP/CGM. O mesmo acontece com as implementações dos algoritmos para os problemas relacionados, caminhos mais curtos, busca em profundidade e árvore geradora mínima, quando comparadas a algumas de suas versões paralelas em outro modelos de computação paralela [5, 15]. As figuras 5.1, 5.2 e 5.3 mostram os *speedups* obtidos para o fecho transitivo. Os valores foram obtidos considerando-se o tempo seqüencial como sendo o tempo do algoritmo paralelo executado sobre 1 processador.

Os modelos BSP, LogP e CGM trouxeram um progresso considerável à área de algoritmos paralelos, mas claramente o estado da arte necessita de futuras pesquisas. Esses modelos são um bom campo para que estudantes de pós-graduação possam desenvolver suas pesquisas.

Enfim, entre os modelos realísticos descritos neste trabalho, o modelo BSP/CGM foi o escolhido para o projeto, a análise e a implementação dos algoritmos estudados neste trabalho de pesquisa por destacar-se como o mais simples e poderoso, principalmente por possuir fases bem definidas de computação e comunicação.

As principais contribuições de nosso trabalho estão relacionadas aos resultados experimentais obtidos, que comprovam a utilidade do modelo BSP/CGM para a implementação de algoritmos paralelos. Além disso, neste

documento estão agrupadas diversas referências e material relacionados aos conceitos envolvidos e aos algoritmos mais conhecidos para resolver problemas clássicos em teoria dos grafos.

Apesar dos resultados obtidos serem satisfatórios, vale ressaltar que as limitações de memória impuseram algumas dificuldades para a realização de teste com entradas de tamanho maior. Dessa forma, pode-se sugerir como trabalhos futuros a melhoria das implementações, através da utilização de estruturas de dados que aproveitem melhor o espaço de memória disponível ou até mesmo a comparação com outros tipos de implementação, explorando outras abordagens para o problema de computar o fecho transitivo e, possivelmente, também podem ser aplicadas aos problemas relacionados descritos aqui.

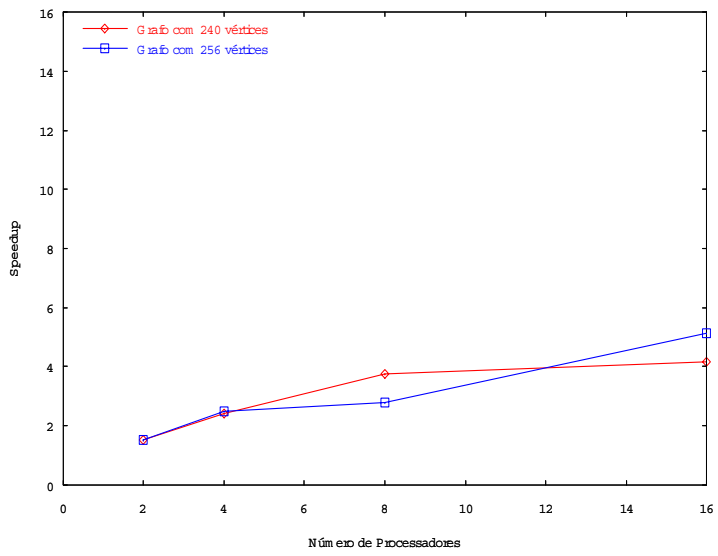


Figura 5.1: *Speedups* para 2, 4, 8 e 16 processadores obtidos para a execução do fecho transitivo sobre grafos com 240 e 256 vértices.

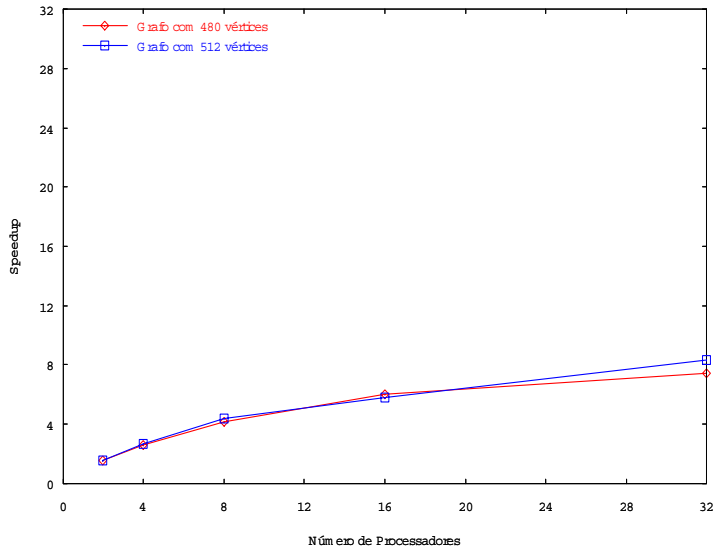


Figura 5.2: *Speedups* para 2, 4, 8, 16 e 32 processadores obtidos para a execução do fecho transitivo sobre grafos com 480 e 512 vértices.

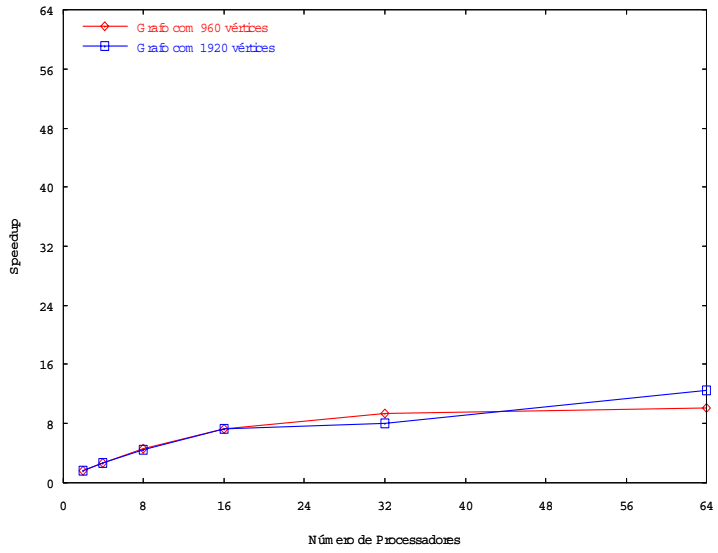


Figura 5.3: *Speedups* para 2, 4, 8, 16, 32 e 64 processadores obtidos para a execução do fecho transitivo sobre grafos com 960 e 1920 vértices.

Apêndice A

Códigos Fontes

Neste apêndice, apresentamos os códigos fontes das implementações do fecho transitivo, dos componentes conexos, dos caminhos mais curtos, da busca em largura e da árvore geradora mínima. Todas as implementações foram desenvolvidas para o modelo BSP/CGM e seus algoritmos, descritos no Capítulo 3.

Além das implementações citadas acima, é apresentado um gerador de grafos que foi utilizado para criar os arquivos de entrada, contendo as matrizes de adjacências dos grafos gerados.

A.1 Fecho Transitivo

```
// Programa: Fecho.c
// Programador: Amaury A. de Castro Jr./Edson Norberto Caceres
// Data: 15/01/2003
// O Dialogo: Este programa recebe um grafo representado atraves de sua matriz de
// adjacencias (n X n) e envia duas submatrizes (n/p X n e n X n/p) para as tarefas.
// Cada tarefa computa o fecho transitivo do subgrafo e envia uma MSG as demais
// tarefas com o fecho pertencente as demais tarefas. Apos O(log p) rodadas, o
// programa obtem o fecho transitivo do grafo.

// Bibliotecas
#include<mpi.h>
#include<stdio.h>
#include<math.h>
#include<string.h>

// Declaracao das constantes globais
#define TAMMAX 2048
#define TAMANHO 2048

// Definicao de tipos
typedef enum{false, true} boolean;
```

```
struct {
   double tempo;
   int idproc;
} msg_env, msg_rec;

// Declaracao das variaveis locais
   int MatrizDados[TAMMAX][TAMMAX];     // Matriz Dados lidos do arquivos de entrada
   int MatrizDadosT[TAMMAX][TAMMAX];    // Matriz dados transposta
   int MatrizLinhas[TAMANHO][TAMMAX];   // Matriz linhas
   int FechoLinhas[TAMANHO][TAMMAX];    // Fecho por linhas
   int MatrizColunas[TAMMAX][TAMANHO];  // Matriz colunas
   int MatrizColunasT[TAMANHO][TAMMAX]; // Matriz colunas transposta
   int FechoColunas[TAMMAX][TAMANHO];   // Fecho por Colunas

   boolean Ja_Foi[TAMMAX][TAMMAX]; //Matriz que indica as arestas ja enviadas

   // Vetores auxiliares para envio das matrizes correspondetes para os processos criados
   int *VetDados, *VetDadosT, *VetLinhas, *VetColunasT;

   int *sbuf;       // Qtd de arestas enviadas para cada processo criado
   int *sbuf_din;   // Arestas a serem enviadas pelo processo corrente
   int *send_displ; // Posicoes iniciais de cada conjunto de arestas enviada
   int *rbuf;       // Qtd de arestas recebidas de cada processo criado
   int *rec_displ;  // Posicoes iniciais de cada conjunto de arestas recebida
   int *buf_send;   // Buffer de envio
   int *buf_rec;    // Buffer de recebimento
   int *pos;        // Ponteiros para cada bloco de arestas a ser enviado

   // Variaveis auxiliares identificacao de processos, tamanhos e contagem
   int rank, size, tam, numelem, tamS, tamR, tam_din, tam_max;
   int i, j, k, l, I, J;

   int root = 0;         // Variavel para identificacao do processo pai
   double rounds;        // Variavel para contagem do numero de rodadas
   double start, finish; // Variaveis auxiliares para a tomada de tempo

   FILE *ArqM, *ArqS;    // Variaveis para fluxo de entrada e saida de dados (arquivos)
   boolean compl;        // Identificacao de informacoes a serem enviadas
   char file_name[30];   // Armazena o nome do arquivo de saida

// Inicio da funcao principal
int main(int argc, char *argv[])
{

// Passo 1. Inicilizacao do ambiente MPI
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size); // numero de tarefas
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // identificacao da tarefa

// Passo 2. Leitura dos dados e alocacao dos vetores dinamicos
// Passo 2.1. Inicializacao do nome do arquivo de saida
   strcpy(file_name, "FechoPar");
// Passo 2.2. Leia os dados (somente o processo pai)
   if (rank == root)
   {
      if( argc < 2 )
      {
         printf( "\n Parametros incorretos.\n Uso: fecho <arq>, onde:\n" );
         printf( "\t <arq> - nome do arquivo de entrada.\n\n" );
         MPI_Finalize();
         return 0;
      }
```

```
// Passo 2.3. Abre o arquivo para leitura dos dados de entrada
      ArqM = fopen(argv[1], "r");
      if (!ArqM)
         printf("ERRO NA ABERTURA DO ARQUIVO DE ENTADA!");

// Passo 2.4. Leitura do numero de vertices do grafo (dimensao da matriz)
      fscanf(ArqM, "%d", &tam_max);

// Passo 2.5. Alocacao do vetores
      VetDados = (int *)malloc((tam_max*tam_max)*sizeof(int));
      VetDadosT = (int *)malloc((tam_max*tam_max)*sizeof(int));
   } // fim if

// Passo 2.6. Broadcast do tamanho do grafo para todas os processos
   MPI_Bcast( &tam_max, 1, MPI_INT, root, MPI_COMM_WORLD);

// Passo 2.7. Calcula as dimensoes das matrizes locais a cada processo
   tam = tam_max / size; // numero de linhas da matriz
   numelem = tam * tam_max; // tamanho da submatriz

// Passo 2.8. Alocacao das matrizes locais a cada processo
   VetLinhas = (int *)malloc((numelem)*sizeof(int));
   VetColunasT = (int *)malloc((numelem)*sizeof(int));


// Passo 2.9. Leitura dos Dados (somente processo pai)
   if (rank == root)
   {
      printf("Numero de vertice = %d.\n", tam_max);

//Passo 2.9.1. Le os dados do arquivo e inicializa VetDados para envio
      k = 0;
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            fscanf(ArqM, "%d", &MatrizDados[i][j]);
            VetDados[k] = MatrizDados[i][j];
            k++;
         }
//Passo 2.9.2. Calcula a matriz transposta
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            MatrizDadosT[j][i] = MatrizDados[i][j];
         }

//Passo 2.9.2. Inicializa VetDadosT (matriz transposta) para envio
      k = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            VetDadosT[k] = MatrizDadosT[i][j];
            k++;
         }
      } // fim for

      fclose(ArqM);
   } // fim da preparacao dos dados (processador 0)
```

```
// Passo 2.10. Mensagem que informa o numero de vertices e as dimensoes das matrizes
// locais em cada processo
   printf("\nrank = %d, TAMMAX: %d, TAMANHO: %d, numelem: %d\n\n", rank, tam_max,
           tam, numelem);

// Passo 3. Envie os dados as tarefas filhos
// Passo 3.1. Distribua os blocos de numelem linhas
   MPI_Scatter(VetDados, numelem, MPI_INT, VetLinhas, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.2. Distribua os blocos de numelem colunas
   MPI_Scatter(VetDadosT, numelem, MPI_INT, VetColunasT, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.3. Inicilize a MatrizLinhas e a matriz Ja_Foi
   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         MatrizLinhas[i][j] = VetLinhas[k];
         Ja_Foi[i][j] = false;
         k++;
      }
   }

// Passo 3.4. Inicilize a MatrizColunasT
   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         MatrizColunasT[i][j] = VetColunasT[k];
         k++;
      }
   }

// Passo 3.5. Inicilize a MatrizColunas
   k = 0;
   for (i = 0; i < tam_max; i++)
   {
      for (j = 0; j < tam; j++)
      {
        MatrizColunas[i][j] = MatrizColunasT[j][i];
        k++;
      }
   }

// Passo 4. Faca log p vezes
   rounds =0.0;

// Passo 4.1. Tomada do tempo inicial
   MPI_Barrier(MPI_COMM_WORLD);
   start = MPI_Wtime();

// Passo 4.2. Calcular o fecho
   do {
     sbuf_din = (int *)malloc(sizeof(int));

// Passo 4.2.1. Inicializa o contador de arestas a serem enviadas pelo processo corrente
      l = 0;
```

```
// Passo 4.2.2. Inicialize as matrizes FechoLinhas e FechoColunas locais
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            FechoLinhas[i][j] = MatrizLinhas[i][j];
            FechoColunas[j][i] = MatrizColunas[j][i];
         }

// Passo 4.2.3.Calcule o Fecho da submatriz armazenada no processo corrente
      for (k = 0; k < tam; k++)
      {
         for (i = 0; i < tam_max; i++)
         {
// Passo 4.2.4. Verifique se existem as aresta i->k e k->j
            for (j = 0; j < tam_max; j++)
            {
               if (MatrizColunas[i][k] && MatrizLinhas[k][j] && i != j)
               {
// Passo 4.2.4.1. Atualize o fecho da submatriz armazenada no processador
                  compl = false;
                  if ((i >= tam*rank) && (i < tam*(rank+1)))
                  {
                     MatrizLinhas[i-rank*tam][j] = 1;
                     if (FechoLinhas[i-rank*tam][j] != 1)
                        FechoLinhas[i-rank*tam][j] = 2;
                     compl = true;
                  } // fim if
                  if ((j >= tam*rank) && (j < tam*(rank+1)))
                  {
                     MatrizColunas[i][j-rank*tam] = 1;
                     if (FechoColunas[i][j-rank*tam] != 1)
                        FechoColunas[i][j-rank*tam] = 2;
                     compl = true;
                  } // fim if
                  if (!(compl) && !(Ja_Foi[i][j]))
                  {
                     sbuf_din = (int *)realloc(sbuf_din, (2+l)*sizeof(int));
                     sbuf_din[l] = i;
                     sbuf_din[l+1] = j;
                     Ja_Foi[i][j] = true;
                     l+=2;
                  } // fim if
               } // fim if
            } // fim for j
         } // fim for i
      } // fim for k

      tam_din = l; //armazena o tamanho do buffer dinamico

// Passo 5. Determine o numero de arestas transitivas geradas a serem enviadas
// Passo 5.1. Dimensione o vetor do numero de arestas a serem envidas
      sbuf = (int *)malloc(size*sizeof(int));
// Passo 5.2. Inicialize o vetor sbuf
      for (k = 0; k < size; k++)
         sbuf[k] = 0;

//Passo 5.3. Faz a contagem das arestas inseridas no vetor dinamico a serem enviadas
// para cada processo
      for (i = 0; i < tam_din; i+=2)
      {
```

```
         if (sbuf_din[i]/tam != rank)
            sbuf[sbuf_din[i]/tam] += 2;

         if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
            sbuf[sbuf_din[i+1]/tam] += 2;
      }

// Passo 5.4. Determine o numero de arestas geradas a serem enviadas no bloco de linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               sbuf[j/tam]+=2;
            } // fim if
         } // fim for j

// Passo 5.5. Determine o numero de arestas geradas a serem enviadas no bloco de colunas
      for (k = 0; k < tam_max; k++)
         for (l = 0; l < tam; l++)
         {
            if ((FechoColunas[k][l] == 2) && (k/tam != rank))
            {
               sbuf[k/tam]+=2;
            } // fim if
         } // fim for l

// Passo 6. Aloca o vetor de ponteiros para as posicoes de envio
      pos = (int *)malloc(size*sizeof(int));

// Passo 6.1. Dimensione o vetor das posicoes das arestas
      tamS = 0;
      for (i = 0; i < size; i++)
      {
         pos[i] = tamS;
         tamS = tamS + sbuf[i];
      }

// Passo 6.2. Aloca o vetor de posicoes de envio
      send_displ =  (int *)malloc(size*sizeof(int));

// Passo 6.3. Compute o vetor das posicoes de envio send_displ
      send_displ[0] = 0;
      for (i = 1; i < size; i++)
         send_displ[i] = send_displ[i-1] + sbuf[i-1];

// Passo 6.4. Dimensione o vetor de recebimento
      buf_send =  (int *)malloc(tamS*sizeof(int));

// Passo 7. Inicializacao do vetor send_buf
      for (i = 0; i < tamS; i++)
      {
         buf_send[i] = 0;
      }

// Passo 8. Insere no buffer de envio as arestas do buffer dinamico a serem enviadas
// para cada processo criado
      for (i = 0; i < tam_din; i+=2)
      {
         if (sbuf_din[i]/tam != rank)
         {
```

```
            buf_send[pos[sbuf_din[i]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i]/tam]+1] = sbuf_din[i+1];
            pos[sbuf_din[i]/tam] += 2;
         }

         if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
         {
            buf_send[pos[sbuf_din[i+1]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i+1]/tam]+1] = sbuf_din[i+1];
            pos[sbuf_din[i+1]/tam] += 2;
         }
      } // fim for

// Passo 8.1. Compute as arestas a serem enviadas do bloco das linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               I = i + rank*tam;
               buf_send[pos[j/tam]] = I;
               buf_send[pos[j/tam]+1] = j;
               pos[j/tam]+=2;
            } // fim if
         } // fim for j

// Passo 8.2. Compute as arestas a serem enviadas no bloco de colunas
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam; j++)
         {
            if ((FechoColunas[i][j] == 2) && (i/tam != rank))
            {
               J = j + rank*tam;
               buf_send[pos[i/tam]] = i;
               buf_send[pos[i/tam]+1] = J;
               pos[i/tam]+=2;
            } // fim if
         } // fim for l

// Passo 9. Dimensione o tamanho do buffer de recebimento
      rbuf =  (int *)malloc(size*sizeof(int));

// Passo 10. Envie/receba o numero de arestas para/de as tarefas
      MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

// Passo 11. Dimensione o vetor de armazenamento rec_displ
      tamR = 0;
      for (i = 0; i < size; i++)
         tamR = tamR + rbuf[i];
      rec_displ =  (int *)malloc(size*sizeof(int));

// Passo 11.2. Compute o vetor das posicoes de recebimento
      rec_displ[0] = 0;
      for (i = 1; i < size; i++)
         rec_displ[i] = rec_displ[i-1] + rbuf[i-1];

// Passo 11.3. Dimensione o vetor de recebimento
      buf_rec =  (int *)malloc(tamR*sizeof(int));
```

```
// Passo 12. Envie/receba as arestas para/de as tarefas
      MPI_Alltoallv(buf_send, sbuf, send_displ, MPI_INT, buf_rec, rbuf, rec_displ,
                    MPI_INT, MPI_COMM_WORLD);

// Passo 12.1. Adicione as arestas recebidas em MatrizLinhas e MatrizColunas
      for (j = 0; j < tamR; j+=2)
      {
// Passo 12.1.1. Atualize o fecho da submatriz armazenada no processador
         if ((buf_rec[j] >= tam*rank) && (buf_rec[j] < tam*(rank+1)))
            MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] = 1;
         if ((buf_rec[j+1] >= tam*rank) && (buf_rec[j+1] < tam*(rank+1)))
            MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank] = 1;
      }

// Passo 13. Libere o espaco atribuido aos vetores alocados dinamicamente
      free(sbuf);
      free(sbuf_din);
      free(send_displ);
      free(buf_send);
      free(rec_displ);
      free(rbuf);
      free(buf_rec);
      free(pos);

      rounds = rounds + 1.0;
   } while (rounds < log10(size)/log10(2) + 1);

// Passo 14. Tomada de tempo final
   MPI_Barrier(MPI_COMM_WORLD);
   finish = MPI_Wtime();

// Passo 15. Armazene o resultado no arquivo de saida
// Passo 15.1. Abra o arquivo de saida
   strcat(file_name, ".txt");
   ArqS = fopen(file_name, "w");

// Passo 15.2. Escreva nos arquivos
   printf("tempo processador %d: %lf\n", rank, (finish-start));
   msg_env.tempo = (finish-start);
   msg_env.idproc = rank;

// Passo 15.3. Retorne o maior tempo obtido
   MPI_Reduce(&msg_env, &msg_rec, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);

   if (rank == root)
      printf("\n\n*** NUMERO DE PROCESSOS = %d - PROCESSO MAIS LONGO (%d) = %lf ***\n\n",
              size, msg_rec.idproc, msg_rec.tempo);

// Passo 16. Prepare os dados resultantes para serem enviados ao processo mestre
   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         VetLinhas[k] = MatrizLinhas[i][j];
         k++;
      }
   } // fim for

   k = 0;
```

```
   for (i = 0; i < tam_max; i++)
   {
      for (j = 0; j < tam; j++)
      {
         VetColunasT[k] = MatrizColunas[i][j];
         k++;
      }
   } // fim for


// Passo 17. Envie os resultados para ao processo mestre
// Passo 17.1. Receba os blocos de numelem linhas
   MPI_Gather(VetLinhas, numelem, MPI_INT, VetDados, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 17.2. Distribua os blocos de numelem colunas
   MPI_Gather(VetColunasT, numelem, MPI_INT, VetDadosT, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 17.3. Imprime o resultado no arquivo de saida
   if (rank == root)
   {
      k = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
            fprintf(ArqS, "%3d ", VetDados[k++]);
         fprintf(ArqS, "\n");
      } // fim i
   }

// Passo 18. Feche os arquivos
   fclose(ArqS);

// Passo 19. Libere a memoria alocada
   if (rank == root)
   {
      free(VetDados);
      free(VetDadosT);
   }
   free(VetLinhas);
   free(VetColunasT);

// Passo 20. Finalize o MPI
   MPI_Finalize();
   return 0;
} // fim funcao main
```

A.2 Caminhos Mais Curtos

```
// Programa: caminhos.c
// Programador: Amaury A. de Castro Jr./Edson Norberto Caceres
// Data: 15/01/2003
// O Dialogo: Este programa recebe um grafo representado atraves de sua matriz de
// adjacencias (n X n) e envia duas submatrizes (n/p X n e n X n/p) para as tarefas.
// Cada tarefa computa todos os caminhos mais curtos do subgrafo, usando a mesma
// estrutura do algoritmo de Caceres et al e envia uma MSG as demais tarefas com o
// fecho pertencente as demais tarefas. Apos O(log p) rodadas, o programa obtem todos
// os caminhos mais curtos do grafo.

// Bibliotecas
#include<mpi.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>

// Declaracao das constantes globais
#define TAMMAX 2048
#define TAMANHO 2048
#define INFINITO 9999

// Definicao de tipos
typedef enum{false, true} boolean;

struct {
   double tempo;
   int idproc;
} msg_env, msg_rec;

// Declaracao das variaveis locais
   int MatrizDados[TAMMAX][TAMMAX];     // Matriz Dados lidos do arquivos de entrada
   int MatrizDadosT[TAMMAX][TAMMAX];    // Matriz dados transposta
   int MatrizLinhas[TAMANHO][TAMMAX];   // Matriz linhas
   int FechoLinhas[TAMANHO][TAMMAX];    // Fecho por linhas
   int MatrizColunas[TAMMAX][TAMANHO];  // Matriz colunas
   int MatrizColunasT[TAMANHO][TAMMAX]; // Matriz colunas transposta
   int FechoColunas[TAMMAX][TAMANHO];   // Fecho por Colunas

   boolean Ja_Foi[TAMMAX][TAMMAX]; //Matriz que indica as arestas ja enviadas

   // Vetores auxiliares para envio das matrizes correspondetes para os processos criados
   int *VetDados, *VetDadosT, *VetLinhas, *VetColunasT;

   int *sbuf;       // Qtd de arestas enviadas para cada processo criado
   int *sbuf_din;   // Arestas a serem enviadas pelo processo corrente
```

```
   int *send_displ; // Posicoes iniciais de cada conjunto de arestas enviada
   int *rbuf;       // Qtd de arestas recebidas de cada processo criado
   int *rec_displ;  // Posicoes iniciais de cada conjunto de arestas recebida
   int *buf_send;   // Buffer de envio
   int *buf_rec;    // Buffer de recebimento
   int *pos;        // Ponteiros para cada bloco de arestas a ser enviado

   // Variaveis auxiliares identificacao de processos, tamanhos e contagem
   int rank, size, tam, numelem, tamS, tamR, tam_din, tam_max, soma;
   int i, j, k, l, I, J;

   int root = 0;         // Variavel para identificacao do processo pai
   double rounds;        // Variavel para contagem do numero de rodadas
   double start, finish; // Variaveis auxiliares para a tomada de tempo

   FILE *ArqM, *ArqS;    // Variaveis para fluxo de entrada e saida de dados (arquivos)
   boolean compl;        // Identificacao de informacoes a serem enviadas
   char file_name[30];   // Armazena o nome do arquivo de saida

// inicio da funcao principal
int main(int argc, char *argv[])
{

// Passo 1. Inicilizacao
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size); // numero de tarefas
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // identificacao da tarefa

// Passo 2. Leitura dos dados e alocacao dos vetores dinamicos
// Passo 2.1. Inicializacao do nome do arquivo de saida
   strcpy(file_name, "CaminPar");

// Passo 2.2. Leia os dados
   if (rank == root) {

      if (argc < 2) {
         printf("\n Parametros incorretos.\n Uso: caminhos <arq>, onde:\n" );
         printf( "\t <arq> - nome do arquivo de entrada.\n\n" );
         MPI_Finalize();
         return 0;
      }

// Passo 2.3. Abre o arquivo para leitura dos dados de entrada
      ArqM = fopen(argv[1], "r");

      if (!ArqM)
     printf("ERRO NA ABERTURA DO ARQUIVO DE ENTADA!");

// Passo 2.4. Leitura do numero de vertices do grafo (dimensao da matriz)
      fscanf(ArqM, "%d", &tam_max);

// Passo 2.5. Alocacao do vetores
      VetDados = (int *)malloc((tam_max*tam_max)*sizeof(int));
      VetDadosT = (int *)malloc((tam_max*tam_max)*sizeof(int));
   } // fim if

// Passo 2.6. Broadcast do tamanho do grafo para todas os processos
   MPI_Bcast( &tam_max, 1, MPI_INT, root, MPI_COMM_WORLD);
```

```
// Passo 2.7. Calcula as dimensoes das matrizes locais a cada processo
   tam = tam_max / size; // numero de linhas da matriz
   numelem = tam * tam_max; // tamanho da submatriz

// Passo 2.8. Alocacao das matrizes locais a cada processo
   VetLinhas = (int *)malloc((numelem)*sizeof(int));
   VetColunasT = (int *)malloc((numelem)*sizeof(int));

// Passo 2.9. Leitura dos Dados (somente processo pai)
   if (rank == root)
   {
      k = 0;
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            fscanf(ArqM, "%d", &MatrizDados[i][j]);
            if (i == j)
               MatrizDados[i][j] = 0;
            else if (MatrizDados[i][j] == 0)
               MatrizDados[i][j] = INFINITO;

            VetDados[k] = MatrizDados[i][j];
            k++;
         }

// Passo 2.9.1. Compute a Matriz transposta da matriz de entrada
         for (i = 0; i < tam_max; i++)
            for (j = 0; j < tam_max; j++)
            {
               MatrizDadosT[j][i] = MatrizDados[i][j];
            }

// Passo 2.9.2. Armazene a matriz transposta no vetor VetDadosT
         k = 0;
         for (i = 0; i < tam_max; i++)
         {
            for (j = 0; j < tam_max; j++)
            {
               VetDadosT[k] = MatrizDadosT[i][j];
               k++;
            }
         } // fim for

// Passo 2.9.3. Feche o arquivo de entrada
         fclose(ArqM);
   } // fim (if) da leitura dos dados (processador 0)

// Passo 3. Envie os dados as tarefas filhos
// Passo 3.1. Distribua os blocos de numelem linhas
   MPI_Scatter(VetDados, numelem, MPI_INT, VetLinhas, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.2. Distribua os blocos de numelem colunas
   MPI_Scatter(VetDadosT, numelem, MPI_INT, VetColunasT, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.3. Armazene VetLinhas em MatrizLinhas
   k = 0;
   for (i = 0; i < tam; i++)
   {
```

```
      for (j = 0; j < tam_max; j++)
      {
         MatrizLinhas[i][j] = VetLinhas[k];
         k++;
      } // fim for (j)
   } // fim for (i)

// Passo 3.4. Armazene VetColunasT em MatrizColunasT
   k = 0;
   for (i = 0; i < tam; i++) {
      for (j = 0; j < tam_max; j++) {
         MatrizColunasT[i][j] = VetColunasT[k];
         k++;
      }
   }

// Passo 3.5. Compute MatrizColunas
   k = 0;
   for (i = 0; i < tam_max; i++) {
      for (j = 0; j < tam; j++) {
         MatrizColunas[i][j] = MatrizColunasT[j][i];
         k++;
      }
   }

// Passo 3.6. Inicialize as matrizes Fecho
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++) {
            FechoLinhas[i][j] = MatrizLinhas[i][j];
            FechoColunas[j][i] = MatrizColunas[j][i];
         }

// Passo 3.7. Inicialize o numero de rodadas
   rounds =0.0;

// Passo 4. Inicie a Tomada de Tempo
   MPI_Barrier(MPI_COMM_WORLD);
   start = MPI_Wtime();

// Passo 5. Faca log p vezes
   do {

// Passo 5.1. Dimensione o buffer de armazenamento
      sbuf_din = (int *)malloc(sizeof(int));

// Passo 5.2. Inicialize a matriz jah foi com peso INFINITO
      l = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            Ja_Foi[i][j] = INFINITO;
         }
      } // fim for

// Passo 5.3. Calcule os caminhos mais curtos da submatriz armazenada na tarefa
      for (k = 0; k < tam; k++)
      {
         for (i = 0; i < tam_max; i++)
         {
```

```
// Passo 5.3.1. Verifique se existem as aresta i->k e k->j
           for (j = 0; j < tam_max; j++)
           {
              compl1 = true;
              compl2 = true;
// Passo 5.3.2. Compute e atualize o caminho minimo da aresta (i,j) se as
//              arestas (i,k) e (k,j) tem peso finito e maior que zero e nao
//              provoque laco
              if ((MatrizColunas[i][k] < INFINITO) && (MatrizLinhas[k][j] < INFINITO) &&
                  (MatrizColunas[i][k] != 0) && (MatrizLinhas[k][j] != 0) && (i != j))
              {
// Passo 5.3.3.  Atualize o caminho minimo
               soma = MatrizColunas[i][k] + MatrizLinhas[k][j];
// Passo 5.3.4. Verifique em que processador esta o resultado (linha)
             if ((i >= tam*rank) && (i < tam*(rank+1)))
             {
                if (MatrizLinhas[i-rank*tam][j] > soma )
                   MatrizLinhas[i-rank*tam][j] = soma;
                if (FechoLinhas[i-rank*tam][j] != 1)
                   FechoLinhas[i-rank*tam][j] = 2;
              }
             else
             {
                compl1 = false;
             } // fim if/else
// Passo 5.3.5. Verifique em que processador esta o resultado (coluna)
             if ((j >= tam*rank) && (j < tam*(rank+1)))
             {
                if (MatrizColunas[i][j-rank*tam] > soma)
                   MatrizColunas[i][j-rank*tam] = soma;
                if (FechoColunas[i][j-rank*tam] != 1)
                   FechoColunas[i][j-rank*tam] = 2;
             }
             else
             {
                compl2 = false;
             } // fim if/else
// Passo 5.3.6. Se o novo elemento esta em uma linha ou colua diferente
             if (!(compl1 && compl2) && (soma < Ja_Foi[i][j]))
             {
                sbuf_din = (int *)realloc(sbuf_din, (3+l)*sizeof(int));
                sbuf_din[l] = i;
                sbuf_din[l+1] = j;
                sbuf_din[l+2] = soma;
                Ja_Foi[i][j] = soma;
                l+=3;
             } // fim if (compl1)
          } // fim if (exite um novo elemento)
       } // fim for j
    } // fim for i
  } // fim for k

// Passo 6. Armazene o tamanho do buffer dinamico
  tam_din = l; //armazena o tamanho do buffer dinamico

// Passo 7. Determine o numero de arestas transitivas geradas a serem enviadas
// Passo 7.1. Dimensione o vetor do numero de arestas a serem envidas
  sbuf = (int *)malloc(size*sizeof(int));
// Passo 7.2. Inicialize o vetor sbuf
  for (k = 0; k < size; k++)
     sbuf[k] = 0;
```

```
// Passo 7.3. Compute o espaco necessario para ser enviado para cada tarefa
   for (i = 0; i < tam_din; i+=3)
   {
// Passo 7.3.1. verifique em que tarefa esta a linha
      if (sbuf_din[i]/tam != rank)
         sbuf[sbuf_din[i]/tam] += 3;
// Passo 7.3.2. verifique me que tarefa esta a coluna
      if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
         sbuf[sbuf_din[i+1]/tam] += 3;
   } // fim for

// Passo 8. Determine o numero de arestas geradas a serem enviadas no bloco de linhas
   for (i = 0; i < tam; i++)
      for (j = 0; j < tam_max; j++)
      {
         if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
         {
            sbuf[j/tam]+=3;
         } // fim if
      } // fim for j

// Passo 9. Determine o numero de arestas geradas a serem enviadas no bloco de colunas
   for (k = 0; k < tam_max; k++)
      for (l = 0; l < tam; l++)
      {
         if ((FechoColunas[k][l] == 2) && (k/tam != rank))
         {
            sbuf[k/tam]+=3;
         } // fim if
      } // fim for l

//Passo 10. Aloca o vetor de ponteiros
   pos = (int *)malloc(size*sizeof(int));

// Passo 10.1. Dimensione o vetor das posicoes das arestas
   tamS = 0;
   for (i = 0; i < size; i++)
   {
      pos[i] = tamS;
      tamS = tamS + sbuf[i];
   }
   send_displ =  (int *)malloc(size*sizeof(int));

// Passo 10.2. Compute o vetor das posicoes de envio send_displ
   send_displ[0] = 0;
   for (i = 1; i < size; i++)
      send_displ[i] = send_displ[i-1] + sbuf[i-1];

// Passo 10.2. Dimensione o vetor de recebimento
   buf_send =  (int *)malloc(tamS*sizeof(int));

// Passo 11. Inicializacao do buf_send
   for (i = 0; i < tamS; i++)
   {
      buf_send[i] = INFINITO;
   }

// Passo 12. Insere no buffer de envio as arestas do buffer dinamico
   for (i = 0; i < tam_din; i+=3)
   {
```

```
      if (sbuf_din[i]/tam != rank)
      {
         buf_send[pos[sbuf_din[i]/tam]] = sbuf_din[i];
         buf_send[pos[sbuf_din[i]/tam]+1] = sbuf_din[i+1];
         buf_send[pos[sbuf_din[i]/tam]+2] = sbuf_din[i+2];
         pos[sbuf_din[i]/tam] += 3;
      }

      if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
      {
         buf_send[pos[sbuf_din[i+1]/tam]] = sbuf_din[i];
         buf_send[pos[sbuf_din[i+1]/tam]+1] = sbuf_din[i+1];
         buf_send[pos[sbuf_din[i+1]/tam]+2] = sbuf_din[i+2];
         pos[sbuf_din[i+1]/tam] += 3;
      }
   } // fim for

// Passo 12.1. Compute as arestas a serem enviadas do bloco das linhas
   for (i = 0; i < tam; i++)
      for (j = 0; j < tam_max; j++)
      {
         if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
         {
            I = i + rank*tam;
            buf_send[pos[j/tam]] = I;
            buf_send[pos[j/tam]+1] = j;
            pos[j/tam]+=3;
         } // fim if
      } // fim for j
// Passo 12.2. Compute as arestas a serem enviadas no bloco de colunas
   for (i = 0; i < tam_max; i++)
      for (j = 0; j < tam; j++)
      {
         if ((FechoColunas[i][j] == 2) && (i/tam != rank))
         {
            J = j + rank*tam;
            buf_send[pos[i/tam]] = i;
            buf_send[pos[i/tam]+1] = J;
            pos[i/tam]+=3;
         } // fim if
      } // fim for l

// Passo 12.3. Dimensione o tamanho do buffer de recebimento
   rbuf =  (int *)malloc(size*sizeof(int));
// Passo 13. Envie/receba o numero de arestas para/de as tarefas
   MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

// Passo 13.1. Dimensione o vetor de armazenamento rec_displ
   tamR = 0;
   for (i = 0; i < size; i++)
      tamR = tamR + rbuf[i];
   rec_displ =  (int *)malloc(size*sizeof(int));
// Passo 13.2. Compute o vetor das posicoes de recebimento
   rec_displ[0] = 0;
   for (i = 1; i < size; i++)
      rec_displ[i] = rec_displ[i-1] + rbuf[i-1];

// Passo 13.3. Dimensione o vetor de recebimento
   buf_rec =  (int *)malloc(tamR*sizeof(int));
// Passo 13.4. Envie/receba as arestas para/de as tarefas
   MPI_Alltoallv(buf_send, sbuf, send_displ, MPI_INT, buf_rec, rbuf, rec_displ,
                 MPI_INT, MPI_COMM_WORLD);
```

```
// Passo 14. Adicione as arestas recebidas em MatrizLinhas e MatrizColunas
   for (j = 0; j < tamR; j+=3)
   {
// Passo 14.1. Atualize o fecho da submatriz armazenada no processador
      if ((buf_rec[j] >= tam*rank) && (buf_rec[j] < tam*(rank+1)))
         if (MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] > buf_rec[j+2])
            MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] = buf_rec[j+2];
      if ((buf_rec[j+1] >= tam*rank) && (buf_rec[j+1] < tam*(rank+1)))
         if (MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank]> buf_rec[j+2])
            MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank] = buf_rec[j+2];
   }

// Passo 14.2. Libere o espaco atribuido a sbuf, send_displ, rbuf, rec_displ
   free(sbuf);
   free(sbuf_din);
   free(send_displ);
   free(buf_send);
   free(rec_displ);
   free(rbuf);
   free(buf_rec);
   free(pos);

   rounds = rounds + 1.0;
   } while (rounds <  log10(size)/log10(2)+1);

// Passo 15. Tomadad de tempo inicial
   MPI_Barrier(MPI_COMM_WORLD);
   finish = MPI_Wtime();

// Passo 16. Abra os arquivos
   strcat(file_name, ".txt");
   ArqS = fopen(file_name, "w");

// Passo 16.1. Escreva nos arquivos
   printf("tempo processador %d: %lf\n", rank, (finish-start));
   msg_env.tempo = (finish-start);
   msg_env.idproc = rank;

   MPI_Reduce(&msg_env, &msg_rec, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root,
               MPI_COMM_WORLD);

   if (rank == root)
      printf("\n\n *** NUMERO DE PROCESSOS = %d - PROCESSO MAIS LONGO (%d) =
              %lf *** \n\n", size, msg_rec.idproc, msg_rec.tempo);

   k = 0;
   for (i = 0; i < tam; i++) {
      for (j = 0; j < tam_max; j++)
      {
         VetLinhas[k] = MatrizLinhas[i][j];
         k++;
      }
   } // fim for

   k = 0;
   for (i = 0; i < tam_max; i++) {
      for (j = 0; j < tam; j++)
      {
         VetColunasT[k] = MatrizColunas[i][j];
         k++;
      }
   } // fim for
```

```
// Passo 17. Receba os blocos de numelem linhas
   MPI_Gather(VetLinhas, numelem, MPI_INT, VetDados, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 17.1. Distribua os blocos de numelem colunas
   MPI_Gather(VetColunasT, numelem, MPI_INT, VetDadosT, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

   if (rank == root)
   {
      k = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
            fprintf(ArqS, "%5d ", VetDados[k++]);
         fprintf(ArqS, "\n");
      } // fim i
   }

   if (rank == root)
   {
      free(VetDados);
      free(VetDadosT);
   }
   free(VetLinhas);
   free(VetColunasT);

// Passo 18. Feche os arquivos
   fclose(ArqS);

// Passo 18.1. Finalize o MPI
   MPI_Finalize();
   return 0;
} // fim funcao main
```

A.3 Busca em Largura

```
// Programa: bfstree.c
// Programador: Edson/Amaury
// Programador: Amaury A. de Castro Jr./Edson Norberto Caceres
// Data: 15/01/2003
// O Dialogo: Este programa recebe um vertice raiz e grafo representado atraves de
// sua matriz de adjacencias (n X n) e envia duas submatrizes (n/p X n e n X n/p)
// para as tarefas. Cada tarefa computa o comprimento do caminho da raiz ate os
// vertices peryencentes ao subgrafo e envia uma MSG as demais tarefas com o valor
// calculado. Apos O(log p) rodadas, o programa obtem uma matriz de disatancias que
// auxilia na escolhas das arestas pertencentes a arvore gerada pela busca em largura.

// Bibliotecas
#include<mpi.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>

// Declaracao das constantes globais
#define TAMMAX 2048
#define TAMANHO 2048
#define INFINITO 9999

// Definicao de tipos
typedef enum{false, true} boolean;

struct {
   double tempo;
   int idproc;
} msg_env, msg_rec;

// Declaracao das variaveis locais
   int MatrizDados[TAMMAX][TAMMAX];     // Matriz Dados lidos do arquivos de entrada
   int MatrizDadosT[TAMMAX][TAMMAX];    // Matriz dados transposta
   int MatrizLinhas[TAMANHO][TAMMAX];   // Matriz linhas
   int FechoLinhas[TAMANHO][TAMMAX];    // Fecho por linhas
   int MatrizColunas[TAMMAX][TAMANHO];  // Matriz colunas
   int MatrizColunasT[TAMANHO][TAMMAX]; // Matriz colunas transposta
   int FechoColunas[TAMMAX][TAMANHO];   // Fecho por Colunas

   int BFSTreeLinhas [TAMANHO][TAMMAX];
   int BFSTreeColunas [TAMMAX][TAMANHO];// Matrizes auxiliares
```

```
   int Ja_Foi[TAMANHO][TAMMAX]; //Matriz que indica as arestas ja enviadas

   // Vetores auxiliares para envio das matrizes correspondetes para os processos criados
   int *VetDados, *VetDadosT, *VetLinhas, *VetColunasT;

   int *sbuf;       // Qtd de arestas enviadas para cada processo criado
   int *sbuf_din;   // Arestas a serem enviadas pelo processo corrente
   int *send_displ; // Posicoes iniciais de cada conjunto de arestas enviada
   int *rbuf;       // Qtd de arestas recebidas de cada processo criado
   int *rec_displ;  // Posicoes iniciais de cada conjunto de arestas recebida
   int *buf_send;   // Buffer de envio
   int *buf_rec;    // Buffer de recebimento
   int *pos;        // Ponteiros para cada bloco de arestas a ser enviado
   int *distancias; // Armazena as distancias calculadas da raiz ate o vertice

   // Variaveis auxiliares identificacao de processos, tamanhos e contagem
   int rank, size, tam, numelem, tamS, tamR, tam_din, tam_max, soma, raiz_bfs;
   int Adj;
   int i, j, k, l, I, J;
   int root=0;
   double rounds;
   double start, finish;

   FILE *ArqM, *ArqS;
   boolean compl1, compl2, encontrou;
   char file_name[30];

// inicio da funcao principal
int main(int argc, char *argv[])
{

// Passo 1. Inicilizacao
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size); // numero de tarefas
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // identificacao da tarefa

// Passo 1.1. Inicializacao do nome do arquivo de saida
   strcpy(file_name, "BfsTreePar");

// Passo 2. Leia os dados
// Passo 2.1. Abra o arquivo de entrada e dimensione as matrizes
   if (rank == root)
   {
      if (argc < 2)
      {
         printf("\n Parametros incorretos.\n Uso: transitive <arq>, onde:\n" );
         printf( "\t <arq> - nome do arquivo de entrada.\n\n" );
         MPI_Finalize();
         return 0;
      }
      ArqM = fopen(argv[1], "r");
      if (!ArqM)
         printf("ERRO NA ABERTURA DO ARQUIVO DE ENTADA!");
      fscanf(ArqM, "%d", &tam_max);

      printf("\n\n Raiz do grafo (valor entre 0 e %d): ", tam_max-1);
      scanf ("%d", &raiz_bfs);

      VetDados = (int *)malloc((tam_max*tam_max)*sizeof(int));
      VetDadosT = (int *)malloc((tam_max*tam_max)*sizeof(int));
   } // fim if
```

```
// Passo 2.2. Envie o numero de linhas da submatriz e raiz a todos os processadores
   MPI_Bcast( &tam_max, 1, MPI_INT, root, MPI_COMM_WORLD);
   MPI_Bcast( &raiz_bfs, 1, MPI_INT, root, MPI_COMM_WORLD);

// Passo 2.3. Calcule o n\'{u}mero de linhas e o tamanho da submatriz
   tam = tam_max/size;
   numelem = tam*tam_max;
   printf("\nInicio rank: %d, TAMMAX: %d, TAMANHO: %d, numelem: %d\n\n", rank,
             tam_max, tam, numelem);

// Passo 2.4. Dimensione as matrizes que vao armazenar os dados
   VetLinhas = (int *)malloc((numelem)*sizeof(int));
   VetColunasT = (int *)malloc((numelem)*sizeof(int));
// Passo 2.5. Se tarefa == root, leia os dados
   if (rank == root)
   {
      k = 0;
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            fscanf(ArqM, "%d", &MatrizDados[i][j]);
            if (i == j)
               MatrizDados[i][j] = 0; //coloca valor 0 na diagonal principal
            else if (MatrizDados[i][j] == 0)
               MatrizDados[i][j] = INFINITO;
            VetDados[k] = MatrizDados[i][j];
            k++;
         }

// Passo 2.6. Compute a Matriz transposta da matriz de entrada
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            MatrizDadosT[j][i] = MatrizDados[i][j];
         }

// Passo 2.7. Armazene a matriz transposta no vetor VetDadosT
      k = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            VetDadosT[k] = MatrizDadosT[i][j];
            k++;
         }
      } // fim for

// Passo 2.8. Feche o arquivo de entrada
      fclose(ArqM);
   } // fim (if) da leitura dos dados (processador 0)

// Passo 3. Envie os dados as demais tarefas
// Passo 3.1. Distribua os blocos de numelem linhas
   MPI_Scatter(VetDados, numelem, MPI_INT, VetLinhas, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.2. Distribua os blocos de numelem colunas
   MPI_Scatter(VetDadosT, numelem, MPI_INT, VetColunasT, numelem, MPI_INT, root,
               MPI_COMM_WORLD);
```

```
// Passo 3.3. Armazene VetLinhas em MatrizLinhas
   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         MatrizLinhas[i][j] = VetLinhas[k];
         k++;
      } // fim for (j)
   } // fim for (i)

// Passo 3.4. Armazene VetColunasT em MatrizColunasT
   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         MatrizColunasT[i][j] = VetColunasT[k];
         k++;
      }
   }

// Passo 3.5. Compute MatrizColunas
   k = 0;
   for (i = 0; i < tam_max; i++)
   {
      for (j = 0; j < tam; j++)
      {
         MatrizColunas[i][j] = MatrizColunasT[j][i];
         k++;
      }
   }

// Passo 4. Inicialize o numero de rodadas
   rounds =0.0;

// Passo 4.1. Inicie a Tomada de Tempo
   MPI_Barrier(MPI_COMM_WORLD);
   start = MPI_Wtime();

// Passo 5. Faca log p vezes
   do {

// Passo 5.1. Dimensione o buffer de armazenamento
      sbuf_din = (int *)malloc(sizeof(int));
// Passo 5.2. Inicialize a primeira posicao do buffer de armazenamento
      l = 0;

      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            Ja_Foi[i][j] = INFINITO;
         }
      } // fim for

// Passo 5.3. Calcule o Fecho da submatriz armazenada na tarefa
      for (k = 0; k < tam; k++)
      {
         for (i = 0; i < tam_max; i++)
         {
```

```
// Passo 5.4. Verifique se existem as aresta i->k e k->j
           for (j = 0; j < tam_max; j++)
           {
              compl1 = true;
              compl2 = true;
// Passo 5.5. Compute e atualize o caminho minimo da aresta (i,j) se as
//            arestas (i,k) e (k,j) tem peso finito e maior que zero e nao
//            provoque laco
              if ((MatrizColunas[i][k] < INFINITO) && (MatrizLinhas[k][j] < INFINITO) &&
                  (MatrizColunas[i][k] != 0) && (MatrizLinhas[k][j] != 0) && (i != j)) {
// Passo 5.6.  Atualize o caminho minimo
                 soma = MatrizColunas[i][k] + MatrizLinhas[k][j];
// Passo 5.7. Verifique em que processador esta o resultado (linha)
              if ((i >= tam*rank) && (i < tam*(rank+1)))
              {
                 if (MatrizLinhas[i-rank*tam][j] > soma )
                    MatrizLinhas[i-rank*tam][j] = soma;
                 if (FechoLinhas[i-rank*tam][j] != 1)
                    FechoLinhas[i-rank*tam][j] = 2;
              }
              else
              {
                 compl1 = false;
              } // fim if/else
// Passo 5.8. Verifique em que processador esta o resultado (coluna)
              if ((j >= tam*rank) && (j < tam*(rank+1)))
              {
                 if (MatrizColunas[i][j-rank*tam] > soma)
                    MatrizColunas[i][j-rank*tam] = soma;
                 if (FechoColunas[i][j-rank*tam] != 1)
                    FechoColunas[i][j-rank*tam] = 2;
              }
              else
              {
                 compl2 = false;
              } // fim if/else
// Passo 5.9. Se o novo elemento esta em uma linha ou colua diferente
              if (!(compl1 && compl2) && (soma < Ja_Foi[i][j]))
              {
                 sbuf_din = (int *)realloc(sbuf_din, (3+l)*sizeof(int));
                 sbuf_din[l] = i;
                 sbuf_din[l+1] = j;
                 sbuf_din[l+2] = soma;
                 Ja_Foi[i][j] = soma;
                 l+=3;
              } // fim if (compl1)
           } // fim if (exite um novo elemento)
        } // fim for j
     } // fim for i
  } // fim for k
// Passo 5.10. Armazene o tamanho do buffer dinamico
  tam_din = l; //armazena o tamanho do buffer dinamico

// Passo 5.11. Determine o numero de arestas transitivas geradas a serem enviadas
// Passo 5.12. Dimensione o vetor do numero de arestas a serem envidas
     sbuf = (int *)malloc(size*sizeof(int));
// Passo 5.13. Inicialize o vetor sbuf
     for (k = 0; k < size; k++)
        sbuf[k] = 0;
// Passo 5.14. Compute o espaco necessario para ser enviado para cada tarefa
     for (i = 0; i < tam_din; i+=3)
     {
```

```
// Passo 5.15. verifique em que tarefa esta a linha
         if (sbuf_din[i]/tam != rank)
            sbuf[sbuf_din[i]/tam] += 3;
// Passo 5.16. verifique me que tarefa esta a coluna
         if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
            sbuf[sbuf_din[i+1]/tam] += 3;
      } // fim for


// Passo 5.17. Determine o numero de arestas geradas a serem enviadas no bloco de linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               sbuf[j/tam]+=3;
            } // fim if
         } // fim for j
// Passo 5.18. Determine o numero de arestas geradas a serem enviadas no bloco de colunas
      for (k = 0; k < tam_max; k++)
         for (l = 0; l < tam; l++)
         {
            if ((FechoColunas[k][l] == 2) && (k/tam != rank))
            {
               sbuf[k/tam]+=3;
            } // fim if
         } // fim for l

      pos = (int *)malloc(size*sizeof(int));

// Passo 5.19. Dimensione o vetor das posicoes das arestas
      tamS = 0;
      for (i = 0; i < size; i++)
      {
         pos[i] = tamS;
         tamS = tamS + sbuf[i];
      }
      send_displ =  (int *)malloc(size*sizeof(int));
// Passo 5.20. Compute o vetor das posicoes de envio send_displ
      send_displ[0] = 0;
      for (i = 1; i < size; i++)
         send_displ[i] = send_displ[i-1] + sbuf[i-1];

// Passo 5.21. Dimensione o vetor de recebimento
      buf_send =  (int *)malloc(tamS*sizeof(int));

      for (i = 0; i < tamS; i++)
      {
         buf_send[i] = INFINITO;
      }

      for (i = 0; i < tam_din; i+=3)
      {
         if (sbuf_din[i]/tam != rank)
         {
            buf_send[pos[sbuf_din[i]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i]/tam]+1] = sbuf_din[i+1];
            buf_send[pos[sbuf_din[i]/tam]+2] = sbuf_din[i+2];
            pos[sbuf_din[i]/tam] += 3;
         }
```

```
         if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
         {
            buf_send[pos[sbuf_din[i+1]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i+1]/tam]+1] = sbuf_din[i+1];
            buf_send[pos[sbuf_din[i+1]/tam]+2] = sbuf_din[i+2];
            pos[sbuf_din[i+1]/tam] += 3;
         }
      } // fim for

// Passo 5.22. Compute as arestas a serem enviadas do bloco das linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++) {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               I = i + rank*tam;
               buf_send[pos[j/tam]] = I;
               buf_send[pos[j/tam]+1] = j;
               pos[j/tam]+=3;
            } // fim if
         } // fim for j
// Passo 5.23. Compute as arestas a serem enviadas no bloco de colunas
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam; j++)
         {
            if ((FechoColunas[i][j] == 2) && (i/tam != rank))
            {
               J = j + rank*tam;
               buf_send[pos[i/tam]] = i;
               buf_send[pos[i/tam]+1] = J;
               pos[i/tam]+=3;
            } // fim if
         } // fim for l

// Passo 5.24. Dimensione o tamanho do buffer de recebimento
      rbuf =  (int *)malloc(size*sizeof(int));
// Passo 5.25. Envie/receba o numero de arestas para/de as tarefas
      MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

// Passo 5.26. Dimensione o vetor de armazenamento rec_displ
      tamR = 0;
      for (i = 0; i < size; i++)
         tamR = tamR + rbuf[i];
      rec_displ =  (int *)malloc(size*sizeof(int));
// Passo 5.27. Compute o vetor das posicoes de recebimento
      rec_displ[0] = 0;
      for (i = 1; i < size; i++)
         rec_displ[i] = rec_displ[i-1] + rbuf[i-1];

// Passo 5.28. Dimensione o vetor de recebimento
      buf_rec =  (int *)malloc(tamR*sizeof(int));
// Passo 5.29. Envie/receba as arestas para/de as tarefas
      MPI_Alltoallv(buf_send, sbuf, send_displ, MPI_INT, buf_rec, rbuf, rec_displ,
                    MPI_INT, MPI_COMM_WORLD);

// Passo 5.30. Adicione as arestas recebidas em MatrizLinhas e MatrizColunas
      for (j = 0; j < tamR; j+=3)
      {
// Passo 5.31. Atualize o fecho da submatriz armazenada no processador
         if ((buf_rec[j] >= tam*rank) && (buf_rec[j] < tam*(rank+1)))
            if (MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] > buf_rec[j+2])
               MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] = buf_rec[j+2];
```

```
         if ((buf_rec[j+1] >= tam*rank) && (buf_rec[j+1] < tam*(rank+1)))
            if (MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank]> buf_rec[j+2])
               MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank] = buf_rec[j+2];
      }
// Passo 5.32. Libere o espaco atribuido a sbuf, send_displ, rbuf, rec_displ
      free(sbuf);
      free(sbuf_din);
      free(send_displ);
      free(buf_send);
      free(rec_displ);
      free(rbuf);
      free(buf_rec);
      free(pos);

      rounds = rounds + 1.0;
   } while (rounds <  log10(size)/log10(2)+1);

// Passo 6. Construa o vetor de distancias a ser enviado pela raiz
   distancias = (int *)malloc((tam_max)*sizeof(int));

// Passo 7. Inicializa a matriz de distancias e marca os vertices ligados diretamente a raiz
   if (rank == (raiz_bfs/tam))
   {
      for (j = 0; j < tam_max; j++)
      {
         distancias[j] = MatrizColunas[j][(raiz_bfs-(tam*rank))];

// Passo 7.1. Marca os vertices que estao diretamente ligados aa raiz
         if ( distancias[j] == 1 )
         {
            BFSTreeLinhas[raiz_bfs-(tam*rank)][j] = 1;
            BFSTreeColunas[j][raiz_bfs-(tam*rank)] = 1;

            if ((j >= tam*rank) && (j < tam*(rank+1)))
            {
               BFSTreeLinhas[j][raiz_bfs-(tam*rank)] = 1;
               BFSTreeColunas[raiz_bfs-(tam*rank)][j] = 1;
            } // fim if
         } // fim if
      } // fim for j
   } // fim if

// Passo 8. Envie as distancias do no raiz a todos os processadores
   MPI_Bcast( distancias, tam_max, MPI_INT, (raiz_bfs/tam), MPI_COMM_WORLD);

// Passo 9. Calcula as demais arestas da Arvore BFS
   for ( i = (rank*tam); i < (rank+1)*tam; i++ )
   {
      j = 0;
      if ( distancias[i] > 1 )
      {
         encontrou = false;
         while ( (!encontrou) && (j < tam_max) )
         {
            if ((MatrizLinhas[i-(rank*tam)][j] == 1) && (distancias[j] == (distancias[i] - 1)))
            {
               BFSTreeLinhas[i-(rank*tam)][j] = 1;
               BFSTreeColunas[j][i-(rank*tam)] = 1;
               if ((j >= tam*rank) && (j < tam*(rank+1)))
               {
                  BFSTreeLinhas[j][i-(rank*tam)] = 1;
                  BFSTreeColunas[i-(rank*tam)][j] = 1;
               }
```

```
               encontrou = true;
            }// fim if
            j++;
         } // fim while
      }
   }// fim for i

   MPI_Barrier(MPI_COMM_WORLD);
   finish = MPI_Wtime();

   k = 0;
   for (i = 0; i < tam; i++)
   {
      for (j = 0; j < tam_max; j++)
      {
         VetLinhas[k] = BFSTreeLinhas[i][j];
         VetColunasT[k] = BFSTreeColunas[j][i];
         k++;
      }
   } // fim for

// Passo 10. Envie os resultados para a tarefa pai
// Passo 10.1. Receba os blocos de numelem linhas
   MPI_Gather(VetLinhas, numelem, MPI_INT, VetDados, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 10.2. Distribua os blocos de numelem colunas
   MPI_Gather(VetColunasT, numelem, MPI_INT, VetDadosT, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 11. Abra os arquivos
   strcat(file_name, ".txt");
   ArqS = fopen(file_name, "w");

   if (rank == root)
   {
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            fprintf(ArqS, "%5d ", (VetDados[i*tam_max+j] || VetDadosT[j*tam_max+i]));
         }
         fprintf(ArqS, "\n");
      } // fim i
   }

// Passo 12. Escreva nos arquivos
   printf("tempo processador %d: %lf\n", rank, (finish-start));

   msg_env.tempo = (finish-start);
   msg_env.idproc = rank;

   MPI_Reduce(&msg_env, &msg_rec, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);

   if (rank == root)
      printf("\n\n *** NUMERO DE PROCESSOS = %d - PROCESSO MAIS LONGO (%d) = %lf *** \n\n",
                  size, msg_rec.idproc, msg_rec.tempo);

// Passo 13. Feche os arquivos
   fclose(ArqS);
```

```
   if (rank == root)
   {
      free(VetDados);
      free(VetDadosT);
   }
   free(VetLinhas);
   free(VetColunasT);
   free(distancias);

// Passo 13. Finalize o MPI
   MPI_Finalize();
   return 0;
} // fim funcao main
```

A.4 Árvore Geradora Mínima

```
// Programa: mintree.c
// Programador: Amaury A. de Castro Jr./Edson Norberto Caceres
// Data: 15/01/2003
// O Dialogo: Este programa recebe um grafo representado atraves de sua matriz
// de adjacencias (n X n) e envia duas submatrizes (n/p X n e n X n/p) para as
// tarefas. Cada tarefa computa as arestas da arvore geradora minima do subgrafo
// e envia uma MSG as demais tarefas com os pesos das arestas. Apos O(log p)
// rodadas, o programa obtem a arvore geradora minima.

// Bibliotecas
#include<mpi.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>

// Declaracao das constantes globais
#define TAMMAX 2048
#define TAMANHO 2048
#define INFINITO 9999

// Definicao de tipos
struct {
   double tempo;
   int idproc;
} msg_env, msg_rec;

typedef enum{false, true} boolean;
```

```
// Declaracao das variaveis locais
   int MatrizDados[TAMMAX][TAMMAX];     // Matriz Dados lidos do arquivos de entrada
   int MatrizDadosT[TAMMAX][TAMMAX];    // Matriz dados transposta
   int MatrizLinhas[TAMANHO][TAMMAX];   // Matriz linhas
   int FechoLinhas[TAMANHO][TAMMAX];    // Fecho por linhas
   int MatrizColunas[TAMMAX][TAMANHO];  // Matriz colunas
   int MatrizColunasT[TAMANHO][TAMMAX]; // Matriz colunas transposta
   int FechoColunas[TAMMAX][TAMANHO];   // Fecho por Colunas

   boolean Ja_Foi[TAMMAX][TAMMAX]; //Matriz que indica as arestas ja enviadas

   // Vetores auxiliares para envio das matrizes correspondetes para os processos criados
   int *VetDados, *VetDadosT, *VetLinhas, *VetColunasT;

   int *sbuf;       // Qtd de arestas enviadas para cada processo criado
   int *sbuf_din;   // Arestas a serem enviadas pelo processo corrente
   int *send_displ; // Posicoes iniciais de cada conjunto de arestas enviada
   int *rbuf;       // Qtd de arestas recebidas de cada processo criado
   int *rec_displ;  // Posicoes iniciais de cada conjunto de arestas recebida
   int *buf_send;   // Buffer de envio
   int *buf_rec;    // Buffer de recebimento
   int *pos;        // Ponteiros para cada bloco de arestas a ser enviado

   // Variaveis auxiliares identificacao de processos, tamanhos e contagem
   int rank, size, tam, numelem, tamS, tamR, tam_din, tam_max, maior, raiz;
   int i, j, k, l, I, J;
   int root=0;
   double rounds;
   double start, finish;

   FILE *ArqM, *ArqS;
   boolean compl1, compl2, encontrou;
   char file_name[30];

// inicio da funcao principal
int main(int argc, char *argv[])
{

// Passo 1. Inicilizacao
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size); // numero de tarefas
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // identificacao da tarefa

   strcpy(file_name, "MinTree");

// Passo 2. Leia os dados
// Passo 2.1. Abra o arquivo de entrada e dimensione as matrizes
   if (rank == root)
   {
      if (argc < 2)
      {
         printf("\n Parametros incorretos.\n Uso: transitive <arq>, onde:\n" );
         printf( "\t <arq> - nome do arquivo de entrada.\n\n" );
         MPI_Finalize();
         return 0;
      }
      ArqM = fopen(argv[1], "r");
      if (!ArqM)
         printf("ERRO NA ABERTURA DO ARQUIVO DE ENTADA!");
```

```
      fscanf(ArqM, "%d", &tam_max);
      printf("\n\n Raiz do grafo (valor entre 0 e %d): ", tam_max-1);
      scanf ("%d", &raiz);

      VetDados = (int *)malloc((tam_max*tam_max)*sizeof(int));
      VetDadosT = (int *)malloc((tam_max*tam_max)*sizeof(int));
   } // fim if
// Passo 2.2. Envie o numero de linhas da submatriz a todos os processadores
   MPI_Bcast( &tam_max, 1, MPI_INT, root, MPI_COMM_WORLD);
   MPI_Bcast( &raiz, 1, MPI_INT, root, MPI_COMM_WORLD);
// Passo 2.3. Calcule o n\'{u}mero de linhas e o tamanho da submatriz
   tam = tam_max/size; // numero de linhas da submatriz
   numelem = tam*tam_max; // tamanho da submatriz
   printf("\nInicio rank: %d, TAMMAX: %d, TAMANHO: %d, numelem: %d\n\n",
            rank, tam_max, tam, numelem);
// Passo 2.4. Dimensione as matrizes que vao armazenar os dados
   VetLinhas = (int *)malloc((numelem)*sizeof(int));
   VetColunasT = (int *)malloc((numelem)*sizeof(int));
// Passo 2.5. Se tarefa == root, leia os dados
   if (rank == root)
   {
      k = 0;
      for (i = 0; i < tam_max; i++)
      for (j = 0; j < tam_max; j++)
      {
         fscanf(ArqM, "%d", &MatrizDados[i][j]);
         if (i == j)
            MatrizDados[i][j] = 0; //coloca valor 0 na diagonal principal
         else if (MatrizDados[i][j] == 0)
            MatrizDados[i][j] = INFINITO;
         VetDados[k] = MatrizDados[i][j];
         k++;
      }

// Passo 2.6. Compute a Matriz transposta da matriz de entrada
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam_max; j++)
         {
            MatrizDadosT[j][i] = MatrizDados[i][j];
         }

// Passo 2.7. Armazene a matriz transposta no vetor VetDadosT
      k = 0;
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            VetDadosT[k] = MatrizDadosT[i][j];
            k++;
         }
      } // fim for

// Passo 2.8. Feche o arquivo de entrada
      fclose(ArqM);
   } // fim (if) da leitura dos dados (processador 0)
// Passo 3. Envie os dados as demais tarefas
// Passo 3.1. Distribua os blocos de numelem linhas
   MPI_Scatter(VetDados, numelem, MPI_INT, VetLinhas, numelem, MPI_INT, root,
               MPI_COMM_WORLD);

// Passo 3.2. Distribua os blocos de numelem colunas
   MPI_Scatter(VetDadosT, numelem, MPI_INT, VetColunasT, numelem, MPI_INT, root,
               MPI_COMM_WORLD);
```

```
// Passo 3.3. Armazene VetLinhas em MatrizLinhas
   k = 0;
   for (i = 0; i < tam; i++) {
      for (j = 0; j < tam_max; j++) {
         MatrizLinhas[i][j] = VetLinhas[k];
         k++;
      } // fim for (j)
   } // fim for (i)

// Passo 3.4. Armazene VetColunasT em MatrizColunasT
   k = 0;
   for (i = 0; i < tam; i++) {
      for (j = 0; j < tam_max; j++) {
         MatrizColunasT[i][j] = VetColunasT[k];
         k++;
      }
   }

// Passo 3.5. Compute MatrizColunas
   k = 0;
   for (i = 0; i < tam_max; i++) {
      for (j = 0; j < tam; j++) {
         MatrizColunas[i][j] = MatrizColunasT[j][i];
         k++;
      }
   }

// Passo 3.6. Inicialize as matrizes Fecho
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++) {
            FechoLinhas[i][j] = MatrizLinhas[i][j];
            FechoColunas[j][i] = MatrizColunas[j][i];
         }

// Passo 4. Inicie a Tomada de Tempo
   MPI_Barrier(MPI_COMM_WORLD);
   start = MPI_Wtime();
   rounds =0.0;

// Passo 5. Faca log p vezes
   do {
// Passo A.3. Dimensione o buffer de armazenamento
      sbuf_din = (int *)malloc(sizeof(int));

// Passo 5.1. Inicialize a primeira posicao do buffer de armazenamento
      l = 0;

      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            Ja_Foi[i][j] = INFINITO;
         }
      } // fim for

// Passo 5.2. Calcule o Fecho da submatriz armazenada na tarefa
      for (k = 0; k < tam; k++)
      {
         for (i = 0; i < tam_max; i++)
         {
```

```
// Passo 5.3. Verifique se existem as aresta i->k e k->j
           for (j = 0; j < tam_max; j++)
           {
              compl1 = true;
              compl2 = true;
// Passo 5.4. Compute e atualize o caminho minimo da aresta (i,j) se as
//            arestas (i,k) e (k,j) tem peso finito e maior que zero e nao
//            provoque laco
              if ((MatrizColunas[i][k] < INFINITO) && (MatrizLinhas[k][j] < INFINITO) &&
                  (MatrizColunas[i][k] != 0) && (MatrizLinhas[k][j] != 0) && (i != j))
              {
// Passo 5.5.  Atualize o valor da arestas escolhida para a arvore geradora minima
                 maior = MatrizColunas[i][k];
                 if (maior < MatrizLinhas[k][j])
                    maior = MatrizLinhas[k][j];
// Passo 5.6. Verifique em que processador esta o resultado (linha)
                 if ((i >= tam*rank) && (i < tam*(rank+1)))
                 {
                    if (MatrizLinhas[i-rank*tam][j] > maior )
                       MatrizLinhas[i-rank*tam][j] = maior;
                    if (FechoLinhas[i-rank*tam][j] != 1)
                       FechoLinhas[i-rank*tam][j] = 2;
                 }
                 else
                 {
                    compl1 = false;
                 } // fim if/else
// Passo 5.7. Verifique em que processador esta o resultado (coluna)
                 if ((j >= tam*rank) && (j < tam*(rank+1)))
                 {
                    if (MatrizColunas[i][j-rank*tam] > maior)
                       MatrizColunas[i][j-rank*tam] = maior;
                    if (FechoColunas[i][j-rank*tam] != 1)
                       FechoColunas[i][j-rank*tam] = 2;
                 }
                 else
                 {
                    compl2 = false;
                 } // fim if/else
// Passo 5.8. Se o novo elemento esta em uma linha ou colua diferente
                 if (!(compl1 && compl2) && (maior < Ja_Foi[i][j]))
                 {
                    sbuf_din = (int *)realloc(sbuf_din, (3+l)*sizeof(int));
                    sbuf_din[l] = i;
                    sbuf_din[l+1] = j;
                    sbuf_din[l+2] = maior;
                    Ja_Foi[i][j] = maior;
                    l+=3;
                 } // fim if (compl1)
              } // fim if (exite um novo elemento)
           } // fim for j
        } // fim for i
     } // fim for k
// Passo 5.9. Armazene o tamanho do buffer dinamico
     tam_din = l; //armazena o tamanho do buffer dinamico
// Passo 5.10. Determine o numero de arestas transitivas geradas a serem enviadas
// Passo 5.11. Dimensione o vetor do numero de arestas a serem envidas
     sbuf = (int *)malloc(size*sizeof(int));
```

```
      for (k = 0; k < size; k++)
         sbuf[k] = 0;
         for (i = 0; i < tam_din; i+=3)
         {
            if (sbuf_din[i]/tam != rank)
               sbuf[sbuf_din[i]/tam] += 3;
            if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
               sbuf[sbuf_din[i+1]/tam] += 3;
      } // fim for

// Passo 5.12. Determine o numero de arestas geradas a serem enviadas no bloco de linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               sbuf[j/tam]+=3;
            } // fim if
         } // fim for j
// Passo 5.13. Determine o numero de arestas geradas a serem enviadas no bloco de colunas
      for (k = 0; k < tam_max; k++)
         for (l = 0; l < tam; l++)
         {
            if ((FechoColunas[k][l] == 2) && (k/tam != rank))
            {
               sbuf[k/tam]+=3;
         } // fim if
      } // fim for l

      pos = (int *)malloc(size*sizeof(int));

// Passo 5.14. Dimensione o vetor das posicoes das arestas
      tamS = 0;
      for (i = 0; i < size; i++)
      {
         pos[i] = tamS;
         tamS = tamS + sbuf[i];
      }
      send_displ =  (int *)malloc(size*sizeof(int));
// Passo 5.15. Compute o vetor das posicoes de envio send_displ
      send_displ[0] = 0;
      for (i = 1; i < size; i++)
         send_displ[i] = send_displ[i-1] + sbuf[i-1];
// Passo 5.16. Dimensione o vetor de recebimento
      buf_send =  (int *)malloc(tamS*sizeof(int));

      for (i = 0; i < tamS; i++)
      {
         buf_send[i] = INFINITO;
      }

      for (i = 0; i < tam_din; i+=3)
      {
         if (sbuf_din[i]/tam != rank)
         {
            buf_send[pos[sbuf_din[i]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i]/tam]+1] = sbuf_din[i+1];
            buf_send[pos[sbuf_din[i]/tam]+2] = sbuf_din[i+2];
            pos[sbuf_din[i]/tam] += 3;
         }
```

```
         if ((sbuf_din[i+1]/tam != rank) && (sbuf_din[i]/tam != sbuf_din[i+1]/tam))
         {
            buf_send[pos[sbuf_din[i+1]/tam]] = sbuf_din[i];
            buf_send[pos[sbuf_din[i+1]/tam]+1] = sbuf_din[i+1];
            buf_send[pos[sbuf_din[i+1]/tam]+2] = sbuf_din[i+2];
            pos[sbuf_din[i+1]/tam] += 3;
         }
      } // fim for

// Passo 5.17. Compute as arestas a serem enviadas do bloco das linhas
      for (i = 0; i < tam; i++)
         for (j = 0; j < tam_max; j++)
         {
            if ((FechoLinhas[i][j] == 2) && (j/tam != rank))
            {
               I = i + rank*tam;
               buf_send[pos[j/tam]] = I;
               buf_send[pos[j/tam]+1] = j;
               pos[j/tam]+=3;
            } // fim if
         } // fim for j
// Passo 5.18. Compute as arestas a serem enviadas no bloco de colunas
      for (i = 0; i < tam_max; i++)
         for (j = 0; j < tam; j++)
         {
            if ((FechoColunas[i][j] == 2) && (i/tam != rank))
            {
               J = j + rank*tam;
               buf_send[pos[i/tam]] = i;
               buf_send[pos[i/tam]+1] = J;
               pos[i/tam]+=3;
            } // fim if
         } // fim for l

// Passo 5.19. Dimensione o tamanho do buffer de recebimento
      rbuf =  (int *)malloc(size*sizeof(int));
// Passo 5.20. Envie/receba o numero de arestas para/de as tarefas
      MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

// Passo 5.21. Dimensione o vetor de armazenamento rec_displ
      tamR = 0;
      for (i = 0; i < size; i++)
         tamR = tamR + rbuf[i];
      rec_displ =  (int *)malloc(size*sizeof(int));
// Passo 5.22. Compute o vetor das posicoes de recebimento
      rec_displ[0] = 0;
      for (i = 1; i < size; i++)
         rec_displ[i] = rec_displ[i-1] + rbuf[i-1];
// Passo 5.24. Dimensione o vetor de recebimento
      buf_rec =  (int *)malloc(tamR*sizeof(int));
// Passo 5.25. Envie/receba as arestas para/de as tarefas
      MPI_Alltoallv(buf_send, sbuf, send_displ, MPI_INT, buf_rec, rbuf, rec_displ,
                    MPI_INT, MPI_COMM_WORLD);

// Passo 5.26. Adicione as arestas recebidas em MatrizLinhas e MatrizColunas
      for (j = 0; j < tamR; j+=3)
      {
// Passo 5.27. Atualize o fecho da submatriz armazenada no processador
         if ((buf_rec[j] >= tam*rank) && (buf_rec[j] < tam*(rank+1)))
            if (MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] > buf_rec[j+2])
               MatrizLinhas[buf_rec[j]-tam*rank][buf_rec[j+1]] = buf_rec[j+2];
```

```
         if ((buf_rec[j+1] >= tam*rank) && (buf_rec[j+1] < tam*(rank+1)))
            if (MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank]> buf_rec[j+2])
               MatrizColunas[buf_rec[j]][buf_rec[j+1]-tam*rank] = buf_rec[j+2];
      }

// Passo 5.28. Libere o espaco atribuido a sbuf, send_displ, rbuf, rec_displ
      free(sbuf);
      free(sbuf_din);
      free(send_displ);
      free(buf_send);
      free(rec_displ);
      free(rbuf);
      free(buf_rec);
      free(pos);

   rounds = rounds + 1.0;
   } while (rounds <  log10(size)/log10(2)+1);

   MPI_Barrier(MPI_COMM_WORLD);
   finish = MPI_Wtime();

// Passo 6. Escreva nos arquivos
   k = 0;
   for (i = 0; i < tam; i++)
      for (j = 0; j < tam_max; j++)
      {
         VetLinhas[k] = MatrizLinhas[i][j];
         VetColunasT[k] = MatrizColunas[j][i];
         k++;
      }

// Passo 7. Envie os resultados para a tarefa pai
// Passo 7.1. Receba os blocos de numelem linhas
   MPI_Gather(VetLinhas, numelem, MPI_INT, VetDados, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 7.2. Distribua os blocos de numelem colunas
   MPI_Gather(VetColunasT, numelem, MPI_INT, VetDadosT, numelem, MPI_INT, root,
              MPI_COMM_WORLD);

// Passo 8. Abra os arquivos
   strcat(file_name, ".txt");
   ArqS = fopen(file_name, "w");

// Passo 8.1. Escreva nos arquivos
   if (rank == root)
   {
      for (i = 0; i < tam_max; i++)
      {
         for (j = 0; j < tam_max; j++)
         {
            if (VetDados[i*tam_max+j] == MatrizDados[i][j])
               fprintf(ArqS, "%5d ", VetDados[i*tam_max+j]);
            else
               fprintf(ArqS, "%5d ", 0);
         }
         fprintf(ArqS, "\n");
      }
   }
```

```
// Passo 9. Escreva os tempos obtidos
   printf("tempo processador %d: %lf\n", rank, (finish-start));

   msg_env.tempo = (finish-start);
   msg_env.idproc = rank;

   MPI_Reduce(&msg_env, &msg_rec, 1, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);

   if (rank == root)
      printf("\n\n *** NUMERO DE PROCESSOS = %d - PROCESSO MAIS LONGO (%d) = %lf *** \n\n",
               size, msg_rec.idproc, msg_rec.tempo);

// Passo 10 Feche os arquivos
   fclose(ArqS);

// Passo 11. Finalize o MPI
   MPI_Finalize();
   return 0;
} // fim funcao main
```

A.5 Gerador de Grafos

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

//---------------------------------------------------------------------
// Definicao de variaveis
//---------------------------------------------------------------------
#define Max 2048     // numero maximo de elementos na lista
#define FALSE 0

//-------------------------------------------------------------
// Declaracao de funcoes utitlizadas
//-------------------------------------------------------------
void fout( int, FILE *, int );
void gera_grafo(int, char *, char *, char *);

//-------------------------------------------------------------
// Declaracao de variaveis
//-------------------------------------------------------------

FILE *fp;
int Grafo[Max][Max]; // vetor que armazena o proximo de cada elemento
```

```
//-------------------------------------------------------------
// Programa Principal
//-------------------------------------------------------------
main( int argc, char *argv[] )
{
   int nVertices;

        // verificando se os parametros de entrada estao corretos
        if( argc < 6 )
        {
                printf( "\n Parametros incorretos.\n Uso: gera <n> <tipo1>
                            <tipo2> <tipo3> <arqs>, onde:\n" );
                printf( "\t <n> - numero de vertices do grafo.\n" );
                printf( "\t <tipo1> - Tipo de grafo (quantidade de arestas):\n" );
                printf( "\t\t e - gera grafos esparsos;\n" );
                printf( "\t\t d - gera grafos densos;\n" );
                printf( "\t\t p - gera uma lista.\n" );
                printf( "\t <tipo2> - Tipo de arestas do grafo:\n" );
                printf( "\t\t o - gera grafos orientados;\n" );
                printf( "\t\t n - gera grafos nao orientados;\n" );
                printf( "\t <tipo3> - Peso nas arestas do grafo:\n" );
                printf( "\t\t c - gera grafo com peso nas arestas;\n" );
                printf( "\t\t s - gera grafo sem peso nas arestas;\n" );
                printf( "\t <arqs> - nome do arquivo de saida.\n\n" );
                return FALSE;
        }

        nVertices = atoi( argv[1] );

        // se numero de elementos maior que 200000 ou menor que 1
        if( nVertices < 1 || nVertices > Max )
        {
                printf( "\n%s: total elementos incorreto.(Maximo %ld)\n\n", argv[0], Max );
                return FALSE;
        }

        gera_grafo( nVertices, argv[2], argv[3], argv[4] );

        // envia resultado para o arquivo
        fp = fopen( argv[5], "wr" );
        fout( nVertices, fp, nVertices );
        fclose( fp );
}

//-----------------------------------------------------
// Funcao fout - direciona resultado para um arquivo
//-----------------------------------------------------
void fout( int f, FILE *fl, int num )
{
        int i, j;

        fprintf( fl, "%d\n", num );

        for( i=0; i<f; i++ )
        {
           for( j=0; j<f; j++ )
              fprintf( fl, "%d ", Grafo[i][j] );

           fprintf( fl, "\n");
        }
}
```

```
//-----------------------------------------------------------
// Funcao InitStruct - inicializa as estruturas utilizadas
//-----------------------------------------------------------
void gera_grafo (int qtd, char *tipo1, char *tipo2, char *tipo3 )
{
        int i, j,           // contadores
        aux,
        Maximo,  //numero maximo de arestas
            nArestas,      // contador do numero de arestas
            linha,      // indice da linha
            coluna;     //indice da coluna

        // inicializa a matriz
        for (i=0; i<qtd; i++ )
           for (j=0; j<qtd; j++ )
              Grafo[i][j] = 0;

        aux = qtd/2;
        Maximo = 2 * aux * ( aux - 1 ); //calcula o numero maximo de arestas

        // gera numeros aleatorios para a funcao rand()
        srand ((unsigned)time(0));
        nArestas = (int)(rand()%Maximo);

        //printf("Numero de arestas: %d \n", nArestas);

        if ( (*tipo1 == 'e') && ( nArestas >= (Maximo/2) ) )
        {
           nArestas = nArestas - (Maximo/2);
        }
        else if ( (*tipo1 == 'd' ) && ( nArestas <= (Maximo*Maximo/2) ) )
        {
           nArestas = nArestas + (Maximo/2);
        }
        else if ( *tipo1 == 'p' )
        {
           for (i=0; i<(qtd-1); i++ )
              Grafo[i][i+1] = 1;

           if ( *tipo2 == 'n' )
              for (i=0; i<(qtd-1); i++ )
                 Grafo[i+1][i] = 1;

           return;
        }

        //printf("Numero de arestas: %d \n", nArestas);

        // inicializa 'i'
        i = 0;

        // obtem proximos de cada elemento
        while (i < nArestas)
        {
           linha = (int) (rand()%qtd);
           coluna = (int) (rand()%qtd);

           if ( ( *tipo3 == 'c' ) && ( !Grafo[linha][coluna] ) )
           {
              aux = (int) (rand()%10);
              Grafo[linha][coluna] = aux;
```

```
            if ( *tipo2 == 'n' )
               Grafo[coluna][linha] = aux;
         }
         else if ( ( *tipo3 == 's' ) && ( !Grafo[linha][coluna] ) )
         {
            Grafo[linha][coluna] = 1;

            if ( *tipo2 == 'n' )
               Grafo[coluna][linha] = 1;
         }

         i++;
      }
}
```

Referências Bibliográficas

[1] S. Baase. *Computer Algorithms - Introduction to Design and Analysis*. Addison-Wesley, 1993.

[2] A. Beguelin, A. Geist, J. Dongarra, W. Jiang, R. Manchek, e V. Sunderam. *PVM - Parallel Virtual Machine: A Users Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[3] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, e P. Spirakis. Bsp vs logp. *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, páginas 25–32, Junho 1996.

[4] E. Cáceres. Comunicação pessoal (orientação), Março 2003.

[5] E. 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. *Lecture Notes in Computer Science*, 1256:390–400, 1997.

[6] E. Cáceres, S. W. Song, e J. L. Szwarcfiter. A parallel algorithm for transitive closure. Relatório Técnico RT-MAC-2002-04, DCC-IME-USP, 2002. 5 pg.

[7] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 2 edição, 1990.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, e T. von Eicken. Logp: Towards a realistic model of parallel computation. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 4:1–12, 1993.

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

[10] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *ACM Conference on Computational Geometry*, 1993.

[11] J. Ebert. A sensitive transitive closure algorithm. *Information Processing Letters*, 12:255–258, 1981.

[12] M. Goodrich. Parallel algorithms column 1: Models of computation. *SIGACT News*, (24):16–21, 1993.

[13] S. M. Götz. *Communication-Efficient Parallel Algorithms for Minimum Spanning Tree Computations*. Tese de Doutoramento, Department of Mathematics and Computer Science - University of Paderborn - Alemanha, Maio 1998.

[14] J. Hopcroft e R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[15] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[16] 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, 1999.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[18] O. A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[19] H. Mongelli. *Algoritmos Paralelos para Solução de Sistemas Lineares*. Tese de Mestrado, Instituto de Matemática e Estatística - USP - São Paulo/SP - Brasil, Junho 1995.

[20] H. Mongelli. *Algoritmos CGM para Busca Uni e Bidimensional de Padrões com e sem Escala*. Tese de Doutoramento, Instituto de Matemática e Estatística - USP - São Paulo/SP - Brasil, Abril 2000.

[21] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[22] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. Tese de Doutoramento, Helsinki University of Technology - Espoo, Finlândia, Junho 1995.

[23] E. Nuutila. An experimental study on transitive closure representations. Relatório técnico, Helsinki University of Technology - Espoo, Finlândia, 1996.

[24] A. Pagourtzis, I. Potapov, e W. Rytter. Pvm computation of the transitive closure: The dependency graph aproach. *Euro PVM/MPI 2001*, páginas 249–256, 2001.

[25] A. Pagourtzis, I. Potapov, e W. Rytter. Observations on parallel computation of transitive and max-closure problems. *Euro PVM/MPI 2002*, páginas 217–225, 2002.

[26] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[27] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[28] R. Sedgewick. *Algorithms*. Addison-Wesley, 2 edição, 1988.

[29] D. B. Skillicorn, J. M. D. Hill, e W. F. McColl. Questions and answers about bsp. Relatório Técnico PRG-TR-15-96, Oxford Univesity Computing Laboratory, 1996.

[30] M. A. Stefanes. *Algoritmos e Implementações Paralelas para Florestas Geradoras Mínimas*. Tese de Mestrado, Instituto de Matemática e Estatística - USP - São Paulo/SP - Brasil, Dezembro 1997.

[31] M. A. Stefanes. Algoritmos paralelos para modelo realísticos, Março 2000. Qualificação de Doutorado - Instituto de Matemática e Estatística - USP - São Paulo/SP - Brasil.

[32] J. L. Szwarcfiter. *Grafos e Algoritmos Computacionais*. Editora Campus, 1986.

[33] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

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

[35] H. S. Warren. A modification of Warshall algorithm for the transitive closure of binary relations. *Communications of the ACM*, 18(4):218–220, 1975.

[36] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.