Algoritmos Paralelos de Granularidade Grossa em Grafos Bipartidos Convexos

MARCO AURÉLIO STEFANES

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU
DE
DOUTOR EM CIÊNCIA DA COMPUTAÇÃO

Orientador: Prof. Dr. José Augusto Ramos Soares

Durante a elaboração deste trabalho, o autor recebeu apoio financeiro da FAPESP

São Paulo, abril de 2003

Algoritmos Paralelos de Granularidade Grossa em Grafos Bipartidos Convexos

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Marco Aurélio Stefanes
e aprovada pela comissão julgadora.

São Paulo, 12 de fevereiro de 2003.

Banca examinadora:

- Prof. Dr. José Augusto R. Soares (orientador) - IME-USP
- Prof. Dr. Siang Wun Song - IME-USP
- Prof. Dr. Jayme L. Szwarcfiter - UFRJ
- Prof. Dr. Valmir C. Barbosa - UFRJ
- Prof. Dr. Henrique Mongelli - UFMS

À minha família

Agradecimentos

Ao Prof. José Augusto, pela paciência e dedicação com que me orientou para que juntos produzíssemos este trabalho.

Ao prof. Marcos Gubitoso, pelas primeira dicas sobre MPI e tantas outras.

Ao Prof. Alfredo Goldman, pelas boas dicas e ajuda na implementação.

Ao Prof. Coelho, sempre disposto a ajudar.

À equipe do instituto francês ID-INRIA que possibilitaram a realização dos testes dos algoritmos no I-cluster.

À Eliany, pelo carinho ao longo deste trabalho.

Aos casais Charlie Brown e Débora, Fábio e Valguima, grandes amigos que estiveram sempre presentes e muito me ajudaram em diversos momentos difíceis.

Ao amigo do peito Claus, amigo de todas as horas.

Aos amigos cruspianos Adelmo, Andréa, Danielle, Ivan, Elanir, Marcelus, e Ravel e aos amigos imeanos Bárbara, Cecília, Elói, Emmanuel, Irene, José Antônio, Maité, Mário, Raul e Said pelo companherismo.

Aos amigos Jair, Karin, Marcelo e Fátima, às vezes tão longe, às vezes tão perto.

Ao Pinho e ao Feijão pelo apoio na CPG.

Aos funcionários do CRUSP, em especial, ao Robson e ao Hélio.

Resumo

Neste trabalho discutimos os principais modelos de computação paralela e apresentamos, como principal foco do trabalho, soluções para alguns problemas em classes especiais de grafos usando modelos de granularidade grossa que acreditamos sirvam de reflexão para a validação de tais modelos. Tratamos alguns problemas em grafos bipartidos convexos. Estes problemas são: encontrar um emparelhamento máximo, encontrar um conjunto independente máximo, determinar um circuito hamiltoniano e determinar os caminhos mínimos entre todos os pares de vértices em grafos bipartidos biconvexos. Relatamos os resultados experimentais da implementação de alguns dos algoritmos apresentados.

Como principais contribuições, descrevemos uma adaptação para o Modelo BSP/CGM de um algoritmo PRAM para encontrar uma coloração em grafos cujo grau máximo é limitado por uma constante; fazemos uma correção no algoritmo BSP/CGM de Bose *et al* [BCDL99] para encontrar um emparelhamento máximo em grafos bipartidos convexos; descrevemos um novo algoritmo seqüencial para encontrar um conjunto independente máximo nesta classe de grafo e estendemos a idéia deste algoritmo formulando um algoritmo para o modelo BSP/CGM; e desenvolvemos um algoritmo seqüencial linear para encontrar um circuito hamiltoniano de fácil paralelização nesta mesma classe.

Abstract

In this work we discuss the main models of parallel computing and we present, as central focus, algorithms for some problems in special classes of graphs using coarse grained parallel models. We believe that the algorithms presented help the reflection about such models. We address the following problems in convex bipartite graphs under coarse grained parallel models: finding a maximum matching, finding a maximum indenpendent set, solving the Hamiltonian circuit problem, and finding all-pairs shortest paths in doubly convex bipartite graphs. We report experimental implementation results for some of the algorithms.

As main contribuitions: *(i)* we describe a BSP/CGM algorithm for finding a coloring in constant degree graphs based on a PRAM algorithm, *(ii)* we correct the Bose *et al* [BCDL99]'s BSP/CGM algorithm for finding a maximum matching in convex bipartite graphs, *(iii)* we describe a new sequential algorithm for finding maximum independent set in convex bipartite graphs, *(iv)* we elaborate BSP/CGM algorithm for finding maximum independent set in the same class of graphs, and *(v)* we develop a sequential Hamiltonian circuit algorithm which is easily parallelizable.

Sumário

Capítulo 1

Introdução

A computação paralela, desde de seu surgimento há cerca de 20 anos, tem permitido que problemas complexos e aplicações de alto desempenho sejam tratados nas mais diversas áreas de computação científica e engenharia. Mais recentemente áreas como inteligência artificial e biologia computacional têm feito uso intenso de paralelismo. Apesar do início promissor, a avaliação a respeito da computação paralela ainda se divide em defensores entusiásticos e críticos extremos.

Na computação seqüencial, o modelo RAM se mostra uma referência para o desenvolvimento de algoritmos e suas respectivas implementações. Na computação paralela, entretanto, a relação entre o desempenho assintótico e teórico dos algoritmos com as respectivas implementações não encontrou ainda um modelo satisfatório.

O modelo PRAM [Jáj92, Cap. 1], apesar de sua importância conceitual e teórica, não consegue capturar com exatidão a noção de paralelismo. As características não incorporadas ao modelo, tais como custo adicional para referência à memória não-local e latência, têm um grande impacto no desempenho das implementações dos algoritmos. Os modelos de redes [Jáj92, Cap. 1], por sua vez, captam bem a inviabilidade em tempos equivalentes de acesso a uma memória global comparado ao acesso a uma memória local. Mas os algoritmos, por um lado, tendem a ser específicos para uma determinada topologia, perdendo assim generalidade. Por outro lado, a comunicação em tempo constante entre vizinhos não reflete a realidade dos computadores paralelos atuais.

A ampliação do uso da computação paralela está relacionada com a necessidade da compreensão e da incorporação em um modelo paralelo de características intrínsecas à computação paralela, bem como de ignorar aquelas características secundárias superáveis através da tecnologia. Um modelo sólido e uniforme, embora difícil de ser atingido em um ambiente de aceleradas inovações tecnológicas, poderia ser alcançado através de um poderoso modelo de máquina abstrato. Este modelo precisaria, de certa forma, balancear simplicidade com precisão e abstração com praticidade.

Neste trabalho começamos por discutir os principais problemas em relação à computação paralela. Um de nossos principais objetivos é fornecer uma visão geral dos modelos de computação paralela, focalizando principalmente os modelos realísticos. Além disso, apresentamos também alguns resultados que obtivemos nestes modelos para alguns problemas em classes especiais de grafos que acreditamos sirvam de reflexão para a validação de tais modelos.

1.1 Organização e Contribuições

Nesta seção descrevemos como este trabalho está estruturado e apresentamos resumidamente as principais contribuições de nossa tese.

O Capítulo 1 contém esta introdução. No Capítulo 2 descrevemos os modelos realísticos e alguns algoritmos básicos de comunicação que têm a finalidade de ilustrar o funcionamento dos modelos. No Capítulo 3 apresentamos algoritmos para ordenação usando os modelos realísticos e que serão usados no restante do trabalho. No Capítulo 4 relatamos alguns algoritmos nos modelos realísticos para dois problemas bem conhecidos em grafos: encontrar uma coloração em um grafo com grau máximo limitado por uma constante e encontrar a partir desta coloração um conjunto independente maximal nesta classe de grafo. Esse capítulo ilustra como alguns problemas têm seus algoritmos facilmente traduzidos para os modelos realísticos a partir de algoritmos dados no modelo PRAM. No Capítulo 5 tratamos alguns problemas em grafos bipartidos convexos. Este problemas são: encontrar um emparelhamento máximo, encontrar um conjunto independente máximo, determinar um circuito hamiltoniano e, além disso, determinar os caminhos mínimos entre todos os pares de vértices em grafos bipartidos biconvexos. No Capítulo 6 descrevemos os resultados experimentais da implementação do algoritmo de ordenação da Seção 3.2 e dos algoritmos para grafos bipartidos convexos do Capítulo 5. Por fim, no Capítulo 7 tecemos as considerações finais de nossa tese.

O Problema da Ordenação, discutido no Capítulo 3, é uma subrotina dos algoritmos para grafos bipartidos convexos do Capítulo 5. O Algoritmo 3-Coloração de Pseudo-Florestas, Seção 4.1.2, é utilizado como subrotina do Algoritmo $(\Delta + 1)$-Coloração da Seção 4.1.3 que, por sua vez, é uma subrotina para o algoritmo para determinar um conjunto independente maximal, Seção 4.2. O Problema do Emparelhamento Máximo, Seção 5.3, é utilizado como subrotina nos algoritmos para encontrar o conjunto independente máximo, 5.4, e para encontrar um circuito hamiltoniano, Seção 5.5. A Figura 1.1 mostra a relação entre os principais algoritmos deste trabalho.

No Capítulo 4 descrevemos uma adaptação para o Modelo BSP/CGM de um algoritmo PRAM para encontrar uma coloração em grafos cujo grau máximo é limitado por uma constante. O Capítulo 5, onde tratamos de grafos bipartidos convexos, contém os prin-

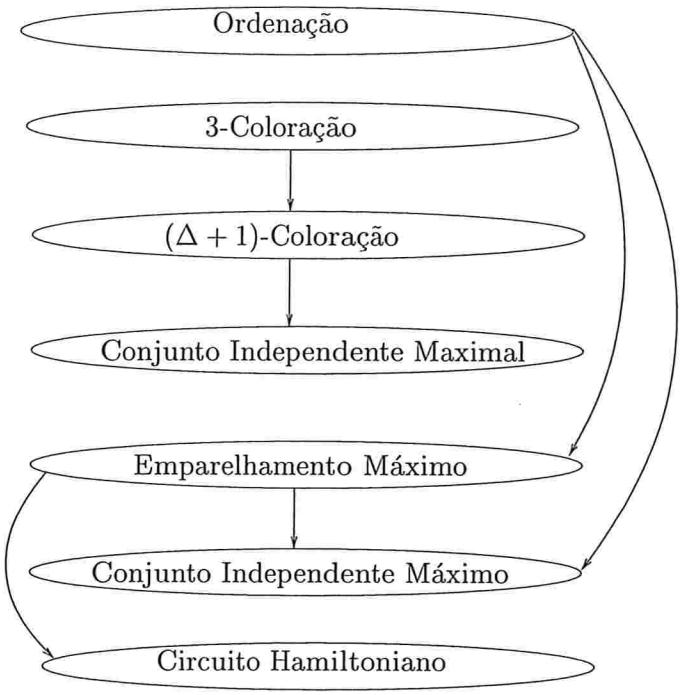


Figura 1.1: Relação entre os algoritmos.

cipais resultados que obtivemos em nossa pesquisa. Na Seção 5.3 fazemos uma correção no algoritmo BSP/CGM de Bose *et al* [BCDL99] para encontrar um emparelhamento máximo em grafos bipartidos convexos. Na Seção 5.4 descrevemos um novo algoritmo seqüencial para encontrar um conjunto independente máximo nesta classe de grafo e estendemos a idéia deste algoritmo formulando um algoritmo para o modelo BSP/CGM. Na Seção 5.5 desenvolvemos um algoritmo seqüencial linear para encontrar um circuito hamiltoniano nesta mesma classe de grafos cuja paralelização usa essencialmente o algoritmo para emparelhamento máximo.

Capítulo 2

Modelos Paralelos Realísticos

Apresentamos neste capítulo alguns modelos de Computação Paralela de propósito geral, denominados *Modelos Realísticos*. Estes modelos buscam refletir sobre as dificuldades inerentes do próprio paralelismo, se esforçam em incorpor mais precisamente os atributos da máquinas paralelas atuais e futuras e buscam estabelecer padrões amplamente aceitos. Além da descrição dos principais modelos realísticos, descrevemos alguns algoritmos básicos de comunicação para ilustrar o funcionamento de cada modelo.

Sob estes modelos, consideraremos que um algoritmo A para um dado problema de tamanho n é *ótimo* se a razão entre $T_A(n,p) \cdot p$ e $T_{A^*}(n)$, onde T_A é o tempo paralelo do algoritmo, p o número de processadores e T_{A^*} é o tempo do melhor algoritmo seqüencial conhecido para o problema, é limitada por uma constante c. Isto é,

$$\frac{T_A(n,p)}{T_{A^*}(n)} \leq \frac{c}{p}.$$

2.1 Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*) proposto por Valiant [Val90], é um dos principais modelos realísticos. Este modelo é pioneiro em incorporar os custos de comunicação, através de parâmetros, como característica do modelo. O objetivo principal deste modelo é servir de *modelo ponte* entre o desenvolvimento de algoritmos e o mundo do *hardware*. Segundo Valiant, este requisito é fundamental para um modelo que deseja desempenhar o mesmo papel do modelo RAM na computação seqüencial.

Uma máquina BSP consiste de p processadores cada um com sua memória local. Os processadores se comunicam através de algum meio de interconexão, gerenciados por um *roteador* com facilidade de sincronização periódica global. Este roteador não possui ne-

nhuma facilidade de duplicação, composição ou *broadcasting*. O modelo possui os seguintes parâmetros:

- L: *periodicidade* é o tempo mínimo entre duas barreiras de sincronização.
- g: descreve a taxa de eficiência entre computação e comunicação, isto é, a razão entre o número de operações computacionais locais realizadas por segundo por todos os processadores e o número total de mensagens de tamanho fixo entregues por segundo pelo roteador.

Um algoritmo BSP consiste de uma seqüência de *superpassos*. Em cada superpasso os processadores operam independentemente realizando computações locais e comunicações globais através de operações de envio e recebimento. Em cada superpasso o roteador realiza uma h-relação, isto é, cada processador envia e recebe no máximo h mensagens de tamanho fixo. Para a realização de uma h-relação, o tempo gasto é proporcional ao tempo gasto para a realização de gh operações computacionais locais. Como na análise seqüencial, tempo e número de operações são relacionados de forma proporcional, abusando da precisão, dizemos que uma h-relação é realizada em tempo gh. Uma mensagem enviada é recebida no próximo superpasso. No final de um superpasso uma barreira de sincronização é realizada. Desta forma, L corresponde ao tempo mínimo de um superpasso. O valor de L pode ser controlado pelo algoritmo, ou mesmo em tempo de execução. Claramente o *hardware* fornece o limite inferior de L e a granularidade de paralelismo do algoritmo fornece o seu limite superior.

O mecanismo de sincronização pode ser desativado de um subconjunto qualquer de processadores de modo que estes processadores não necessitam atrasar sua computação em função de outros. Contudo, eles continuam a poder trocar mensagens com qualquer outro processador.

Valiant discute ainda a simulação de algoritmos PRAM no modelo BSP. Esta simulação pode ser feita de forma ótima considerando uma folga suficiente para os processadores e o parâmetro g como sendo constante. Porém, a simulação não é recomendada quando g é grande. Infelizmente, esta é a maioria dos casos das máquinas paralelas atuais.

2.1.1 BSP*: BSP com Mensagens Longas

Uma extensão do modelo BSP foi proposto por Bäumker *et al.*[BDadH98]. Nesta extensão, chamada BSP*, foi adicionado aos parâmetros p, L e g um parâmetro B, que é o tamanho mínimo que uma mensagem pode ter, de modo a explorar completamente a largura de banda do roteador. As mensagens menores que B são tratadas como sendo de tamanho B. Assim, o modelo gratifica a comunicação em bloco. Isto se deve ao fato de que o envio de grandes mensagens é geralmente suportado pelas máquinas paralelas atuais, e que o tempo de inicialização de uma mensagem geralmente contribui em grande

parte para o custo de comunicação. Sob este modelo, há uma simplificação nos custos de comunicação. Considere, por exemplo, que cada processador envia várias mensagens de tamanho total s para r processadores e recebe várias mensagens de tamanho total s' de r' processadores, então este tempo tem como limitante superior $\max\{g \cdot \max\{s/B + r, s'/B + r'\}, L\}$.

2.1.2 Rotinas Básicas de Comunicação

Broadcast de um Elemento sob o BSP

Considere que um processador deva enviar uma mensagem para n localizações de memória divididas uniformemente entre os p processadores $P_0, P_1, \ldots, P_{p-1}$. Este operação pode ser feita em $\log_d p$ superpassos, usando uma organização lógica dos processadores como uma árvore d-ária balanceada com a raiz em P_0. Para isto, em cada superpasso cada processador que já possui o dado transmite d cópias, uma para cada filho.

O tempo gasto nesta operação é $dg \log_d p$. Além disso, em cada componente são feitas $n/p - 1$ cópias deste elemento em tempo $O(n/p)$. O algoritmo é ótimo se $d = O((n/(gp \lg p) \lg(n/(gp \lg p))$ e $L = O(gd)$. A restrição em d implica $n = \Omega(gp \lg p)$.

Broadcast de um Vetor sob o BSP*

Dado um vetor de tamanho n no processador P_0, queremos enviar este vetor para todos os demais processadores. Para este algoritmo, similar ao exemplo anterior, os processadores são organizados logicamente como uma árvore. Por simplicidade, aqui usamos uma árvore binária. Neste algoritmo, P_0 divide o vetor em $\min\{n, \lg p\}$ pacotes de tamanho no máximo $\lceil n/\lg p \rceil$ e em cada superpasso i envia o i-ésimo pacote a seus filhos. Paralelamente, cada processador que já recebeu o vetor transmite uma cópia para cada filho.

Quanto à complexidade, o algoritmo realiza $O(\lg p)$ superpassos. Em cada superpasso, temos tempo de comunicação $\max\{2g\lceil\lceil n/\lg p\rceil/B\rceil, L\}$ e tempo de processamento $\max\{\lceil n/\lg p\rceil, L\}$. Portanto, o algoritmo toma tempo total $T_{bc}(n) = O(g(n/B + \lg p) + L \lg p + n)$.

Soma Prefixa sob o BSP*

Dados p vetores cada um de tamanho n, onde o vetor A_k está armazenado no processador P_k, queremos computar, para cada $i \in \{0, \ldots, n-1\}$, a soma prefixa $A_0(i) + A_1(i) + \cdots + A_k(i)$ para $0 \leq k < p$. Isto é, queremos a soma prefixa da i-ésima posição do vetor.

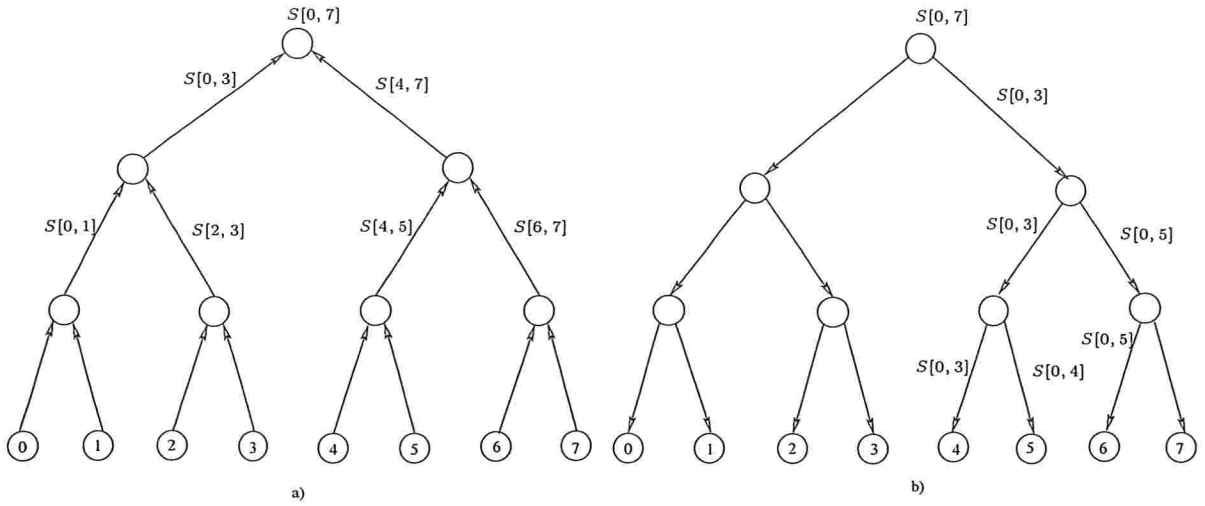


Figura 2.1: Algoritmo Soma Prefixa para 8 processadores. a) Primeira fase e b) Segunda fase. $S[a, b]$ denota a soma $A_a(i) + A_{a+1}(i) + \cdots + A_b(i)$ para $0 \leq i < n$.

Neste algoritmo, os processadores são organizados logicamente como uma árvore binária de p folhas. Desta forma, neste caso cada P_k representa uma folha e um nó interno da árvore. Em cada superpasso um processador realiza as operações para um nó e depois as operações para uma folha. Este algoritmo está dividido em duas fases. Na primeira fase, a soma se move das folhas para a raiz e na segunda alguns cálculos são feitos no sentido da raiz para as folhas. A Figura 2.1 mostra em mais detalhes um exemplo deste algoritmo. Aqui também, o vetor é dividido em $\min\{n, \lg p\}$ pacotes como no algoritmo *Broadcast*.

É fácil ver que este algoritmo sua complexidade $T_{sp}(n)$ igual a do algoritmo anterior, pois são $O(\lg p)$ superpassos com mesmo tempo de processamento e comunicação do *Broadcast* em cada superpasso.

2.2 Modelo CGM

O modelo CGM (*Coarse Granularity Multicomputer*) — Multicomputador com Granularidade Grossa — foi proposto por Dehne *et al* [DFRC93]. Um CGM(n, p) consiste de p processadores, sendo o tamanho total da memória $O(n)$, onde n é o tamanho do problema e cada processador possui $O(n/p)$ memória local. Os processadores podem ser conectados através de um meio de intercomunicação qualquer, rede de interconexão ou memória compartilhada. A memória local é consideravelmente maior que $O(1)$, por exemplo $n/p \geq p$. Um algoritmo CGM consiste de computação local alternada com rodadas de comunicação global. Em uma rodada de comunicação uma única h-relação é roteada ($h = O(n/p)$). Isto é, cada processador envia e recebe mensagens de tamanho total $O(n/p)$. O tempo de

um algoritmo CGM é a soma dos tempos para rodadas de comunicação e de computação.

O modelo CGM pode ser visto como uma variante do modelo BSP. Comparado ao modelo BSP, uma rodada de comunicação/computação no modelo CGM é equivalente a um superpasso no modelo BSP com $h = gn/p$, mas incluindo também a exigência de empacotamento e granularidade grossa (aqui definida como memória local consideravelmente maior que $O(1)$).

Soma Prefixa sob o CGM

As operações mais básicas de comunicação, como *broadcast* e troca de mensagens são feitas facilmente no modelo CGM. Desta forma, vamos analisar o problema da soma prefixa que é uma operação mais complexa. Dados n elementos $x_0, x_1, \ldots, x_{n-1}$ distribuídos de maneira uniforme e consecutiva entre os p processadores, computar $S_i = x_0 \otimes x_1 \otimes \ldots \otimes x_i$, $0 \leq i < n$, para alguma operação associativa $\otimes$. Supondo que cada operação $\otimes$ é feita em tempo constante, um algoritmo seqüencial leva tempo $O(n)$ para resolver o problema. Assumimos que n/p é um inteiro.

Em um algoritmo CGM, cada processador calcula a operação $\otimes$ para cada S_i, $0 \leq i \leq n/p$ com seus n/p elementos, usando o algoritmo seqüencial, e envia a P_0, em uma rodada de comunicação, o resultado local $S_{n/p}$. Em seguida, P_0 calcula a soma prefixa seqüencialmente sobre os p valores recebidos, armazenando em y_k, $0 \leq k < p$, os resultados. Depois disso, P_0 envia, em uma rodada de comunicação, o valor de y_k ao processador P_{k+1}. Finalmente, cada processador opera o valor recebido com cada um de seus resultados parciais S_i obtidos anteriormente. Claramente este algoritmo tem tempo de computação $O(n/p)$ e um número constante de rodadas de comunicação, sendo, portanto, ótimo.

2.3 Modelo LogP

O modelo LogP [CKS+93] busca incorporar mais exatamente os atributos das máquinas existentes. Este modelo é resultado do esforço de diversos grupos de pesquisadores em áreas teóricas, de *software* e de *hardware* no sentido de se produzir um modelo único de computação paralela.

Os parâmetros do modelo são dados por:

- L: limite superior da *latência*, tempo de espera suficiente para a comunicação de uma mensagem ir de sua origem ao seu destino.
- o: *overhead*, tempo que o processador fica comprometido na transmissão e recepção de cada mensagem. Durante este tempo o processador não realiza nenhuma outra operação.

- g: *gap*, intervalo de tempo mínimo entre a transmissão ou recepção de mensagens consecutivas em um processador.

- p: O número de processadores e módulos de memória. O tempo é unitário para operações locais.

O modelo assume que a rede tem capacidade limitada, tal que, no máximo, $\lfloor L/g \rfloor$ mensagens possam estar em transição entre quaisquer dois processadores ao mesmo tempo. Se um processador tenta enviar uma mensagem que extrapole este limite, ele pára e espera até que a mensagem possa ser enviada.

Este modelo é semelhante ao BSP, divergindo do mesmo em dois aspectos. O primeiro é a execução assíncrona. O parâmetro L é usado como medida de latência das mensagens. O segundo é a introdução de um novo parâmetro o, que captura o tempo gasto por um processador para colocar ou receber uma mensagem da rede. Este parâmetro mede o tempo desperdiçado pelo processador que não pode ser medido através de técnica de latência escondida. Devido às variações na latência, as mensagens enviadas por um processador podem não chegar a um mesmo destino na mesma ordem em que foram enviadas.

A escolha dos parâmetros representa um compromisso de capturar as características de máquinas reais e fornecer uma ferramenta razoável para projeto e análise de algoritmos. Os parâmetros não são igualmente importantes em todas as situações. Em alguns casos é possível ignorar um ou mais parâmetros.

Em algoritmos que trocam poucos dados, por exemplo, 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 pela rede, tal que o tempo de transmissão da mensagem é dominado pelos *gaps* inter-mensagens e a latência pode ser desconsiderada. Em algumas máquinas o *overhead* domina o *gap*, assim g pode ser descartado. Apesar do avanço das arquiteturas paralelas, a possibilidade da eliminação do parâmetro o parece ainda prematuro.

O modelo encoraja o escalonamento cuidadoso de computação e a sobreposição de computação e comunicação de acordo com os limites da capacidade da rede. Embora o modelo esteja descrito em termos de máquinas de memória distribuída, os algoritmos não necessitam explicitar operações de troca de mensagens. Algoritmos para memória compartilhada podem ser implementados no modelo através de troca implícita de mensagem.

Tanto o modelo LogP quanto o BSP discutem a possibilidade de introduzir nos algoritmos *folgas paralelas*. Ou seja, um algoritmo seria descrito para v processadores virtuais e rodaria em p processadores físicos, onde v é maior que p. Esta folga permitiria ao compilador e ao processador explorar eficientemente a computação. Por exemplo, com mais de

uma tarefa por processador, caso uma tarefa fosse interrompida por uma espera de dados remotos, o processador seria usado por outra de suas tarefas. Esta técnica, no entanto, é limitada pela banda de comunicação, podendo causar congestionamento de comunicação, e pela espera envolvida na mudança de contexto, que oneraria demasiadamente o Sistema Operacional.

2.3.1 LogGP: LogP com Mensagens Longas

O modelo LogP implicitamente possui um quinto parâmetro, o tamanho da mensagem w. O LogP tem medido com grande precisão o desempenho de algoritmos quando são usadas mensagens curtas. Notamos, por outro lado, que muitas máquinas paralelas atuais fornecem suporte especial para mensagens longas, como largura de banda maiores. O modelo LogGP, que descrevemos nesta seção, captura com mais exatidão as características de máquinas que suportam mensagens longas.

O modelo LogGP [AISS97] estende o modelo LogP com um modelo linear para mensagens longas. Ele busca capturar adequadamente tanto as características de mensagens curtas quanto de mensagens longas. Os modelos de comunicação para mensagens longas existentes, geralmente, modelam o tempo de envio de uma mensagem de n bytes por um modelo linear $t = t_0 + t_b.n$, onde t_0 é o tempo de inicialização e t_b é o tempo por byte [KGGK94]. Estes modelos não refletem precisamente o envio de mensagens curtas, pois juntam o *overhead* e a latência em um único parâmetro t_0.

A importância de incorporar em um modelo mensagens longas e curtas está no seguinte fato: alguns algoritmos podem ter seus desempenhos melhorados de uma forma direta, com uma simples extensão. Porém, como veremos adiante, em alguns casos, para melhorar o desempenho de um algoritmo ótimo para um modelo com mensagens curtas é necessário, não só refazer o padrão de comunicação como criar um novo algoritmo para o problema.

No modelo LogGP, o intervalo mínimo entre mensagens g do LogP é dividido em dois parâmetros. O parâmetro g que é alterado para refletir a recíproca da banda de comunicação por processamento quando são enviadas mensagens curtas. Um novo parâmetro G é incorporado para refletir a recíproca da banda de comunicação quando um processador envia mensagens longas. Os demais parâmetros L, o e p são incorporados sem alterações, além do parâmetro implícito w.

Em máquinas reais é mais claro definir G como tempo por bytes. Já em análises independentes de máquinas pode ser mais interessante definir G como um intervalo por item de tamanho w bytes. O modelo assume que o processador somente tem acesso a uma mensagem após o recebimento completo da mesma. O modelo assume, também, que em um mesmo instante um processador pode receber uma única mensagem.

Como qualquer modelo, o LogGP não captura todas as variações de arquiteturas. Por

exemplo, somente redes com altos graus nos nós e baixo diâmetro possuem os parâmetros G e L constantes. Os resultados experimentais mostram que o modelo LogGP captura os aspectos de comunicação mais importantes das máquinas paralelas atuais.

Veremos na seção a seguir, que o uso de mensagens longas pode mudar a estrutura do próprio algoritmo e levar ao desenvolvimento de um algoritmo completamente diferente do inicial, considerado ótimo.

2.3.2 Rotinas Básicas de Comunicação

Troca de n Mensagens sob o LogP

Uma transferência de uma palavra sob o modelo LogP consome tempo de $L + 2o$. Quando n mensagens são enviadas de um processador para outro o tempo total é $2o + (n+1)g + L$, pois após a primeira mensagem o processador pode receber uma mensagem a cada intervalo g. Esta análise é válida também quando um processador envia n mensagens para n processadores distintos. Neste caso o *overhead* de recebimento é distribuído entre os processadores. Caso n processadores enviem mensagens para um único processador a análise ainda continua válida, exceto que o tempo de recebimento é maior devido a contenção na rede.

Vamos analisar agora um problema mais interessante. Suponha que dois processadores troquem entre si n mensagens cada. Vamos assumir que ambos começam enviando a primeira mensagem ao mesmo tempo (isto implica que houve uma sincronização prévia). Por L unidades de tempo, os processadores enviam uma mensagem a cada intervalo g. Depois disso, o intervalo de envio de cada processador torna-se $\max\{g, 2o\}$, pois cada um precisa também receber mensagens. Após o envio de todas as mensagens, cada processador volta a receber uma mensagem a cada intervalo g. Portanto, o tempo total para n trocas é $2L + 2o + (n - 1 - L/g)\max\{2o, g\}$. Veja a Figura 2.2 onde $tmax = \max\{2o, g\}$. Se a máquina pode transferir w palavras a um custo compensador em uma única mensagem, o tempo total pode ser melhorado para

$$T_{troca}(n) = 2L + 2o + (\lceil n/w \rceil - 1 - L/g)\max\{2o, g\}.$$

Broadcast de n Itens sob o LogP

A versão mais simples deste problema é a difusão de um item. Dado um conjunto de p processadores, a partir de um processador fonte P_0, queremos distribuir p, possivelmente, diferentes itens $i_0, \ldots, i_{p-1}$, cada i_k para o processador destino P_k, $0 \leq k < p$. Sua versão mais geral é a difusão de n itens, onde o processador fonte P_0 possui um conjunto de itens $I_0, \ldots, I_{p-1}$, cada um de tamanho n. O destino do conjunto I_k é o processador P_k.

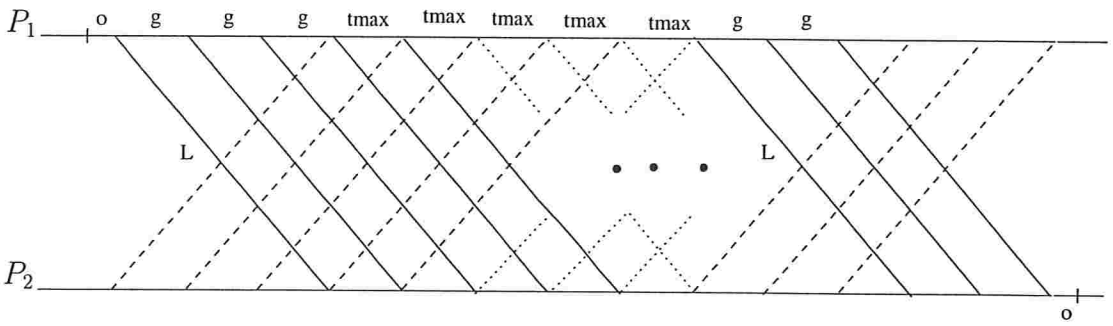


Figura 2.2: Troca de n mensagens entre dois processadores. tmax =max$\{g, 2o\}$.

O algoritmo é bastante simples. O processador P_0 envia cada um dos $(p-1)n$ itens para seu destino como uma mensagem separada. Como esta é a forma mais rápida do processador fonte enviar todos os itens, o algoritmo é ótimo. Este algoritmo tem complexidade $((p-1)n-1)g+L$.

Broadcast de n Itens sob o LogGP

Quando podemos usar envio de mensagens longas, podemos agrupar vários itens em uma única mensagem de forma a explorar melhor a banda de comunicação da rede. Além disso, podemos ter mais processadores ajudando a entregar itens a seus respectivos destinos. Colocamos itens para processadores diferentes em uma única mensagem e o processador receptor se encarrega de entregar as mensagens recebidas que não pertencem a ele aos respectivos destinos.

Considere p potência de 2. Com $n = 1$, o algoritmo funciona da seguinte forma. No início P_0 envia a $P_{p/2}$ todos os itens destinados aos processadores $P_{p/2}, P_{p/2+1}, \ldots, P_{p-1}$. Assim os processadores são divididos em dois grupos de igual tamanho e o problema é reduzido a dois subproblemas com a metade do tamanho. Resolvemos os subproblemas recursivamente, onde P_0 é o processador fonte para o primeiro grupo e $P_{p/2}$ para o segundo. A generalização para $n \geq 2$ é fácil. Em cada passo, o processador fonte simplesmente envia metade de seus conjuntos de itens para o processador receptor, reduzindo assim o problema a dois subproblemas com a metade do tamanho. Este algoritmo tem complexidade $\max\{L, g\}(\lg p - 1) + L + (p-1)n - \lg p$.

Note que este algoritmo melhora o tempo de execução às custas de aumentar o tráfego na rede, uma vez que um dado pode trafegar na rede mais de uma vez. Como os itens devem ser armazenados nos nós intermediários antes da retransmissão, este algoritmo usa significativamente mais memória local que o algoritmo anterior.

Capítulo 3

Algoritmos de Ordenação

Neste capítulo apresentamos dois algoritmos de ordenação usando os modelos realísticos. Na Seção 3.1 descrevemos um algoritmo LogP de ordenação baseado em um algoritmo seqüencial de ordenação por colunas. Na Seção 3.2 relatamos um algoritmo BSP/CGM que utiliza número de rodadas constante e tempo de processamento linear.

Dada uma seqüência $(a_0, a_1, \ldots, a_n)$ dizemos que ela é *crescente* se $a_0 \leq a_1 \leq \cdots \leq a_n$. Esta seqüência é *decrescente* se $a_0 \geq a_1 \geq \cdots \geq a_n$. Dizemos que uma seqüencia está *ordenada* se ela é crescente e *ordenada decrescentemente* se ela é decrescente.

3.1 Algoritmo LogP de Ordenação por Colunas

Inicialmente descrevemos o algoritmo seqüencial que fornece a base do algoritmo LogP apresentado em detalhes a seguir.

3.1.1 Algoritmo Seqüencial

O algoritmo seqüencial e sua correção apresentados nesta seção são descritos por Leighton [Lei85]. Seja Q a matriz $r \times s$ contendo n elementos a serem ordenados, onde $rs = n$, $s|r$ e $r \geq 2(s-1)^2$. Após a execução do algoritmo, a posição i, j de Q conterá o q-ésimo elemento da lista ordenada onde $q = i + jr$, $0 \leq i < r$, $0 \leq j < s$, $0 \leq q < rs$. Dizemos que $rank(a) = q$, se a é o q-ésimo elemento da lista ordenada.

O algoritmo possui oito passos. Nos Passos 1, 3, 5 e 7 os elementos de cada coluna devem ser ordenados. Nos Passos 2, 4, 6 e 8 os elementos da matriz são permutados. No Passo 2, a permutação corresponde a uma "transposição" da matriz da seguinte forma: os elementos de cada coluna j preencherão seqüencialmente as linhas $jr/s, jr/s+1, \ldots, (j+$

$1)r/s-1$. Nesta operação, cada coluna é colocada em r/s linhas da matriz. A operação do Passo 4 é a inversa da operação do Passo 2, ou seja, os elementos das linhas $jr/s, jr/s+1, \ldots, (j+1)r/s-1$ preencherão seqüencialmente a coluna j. Desta forma, os elementos são tomados linha por linha e distribuídos por colunas. Veja a Figura 3.1. A permutação do Passo 6 corresponde a um deslocamento de $\lfloor r/2 \rfloor$ posições para frente nos elementos de Q segundo a ordem da lista ordenada. No Passo 8 é realizada uma operação inversa à do Passo 6. Abaixo segue uma descrição dos 8 passos do algoritmo.

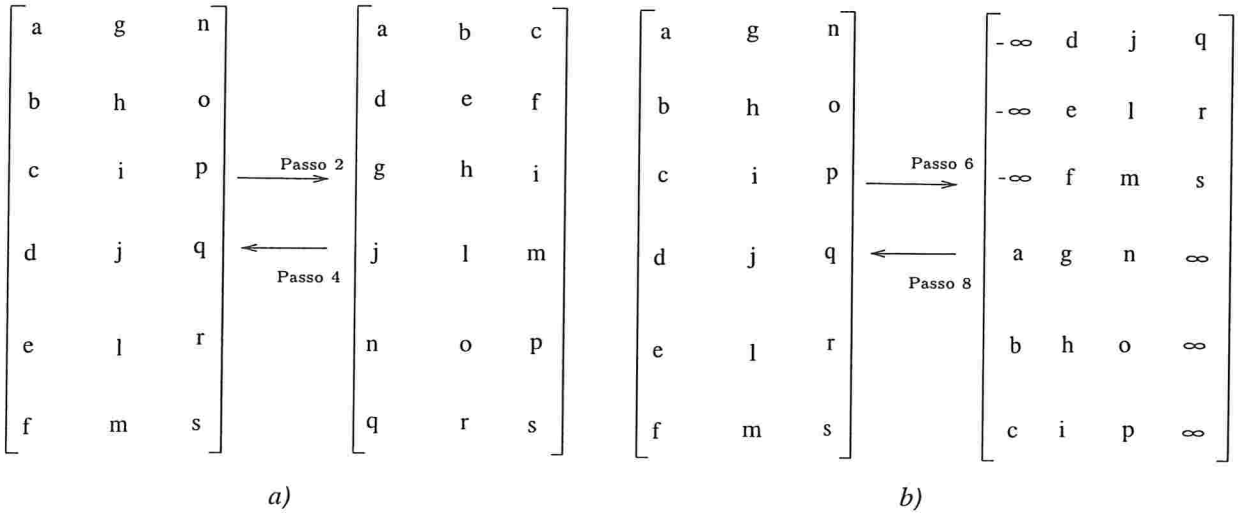


Figura 3.1: As operações de "transposição" *a)* e deslocamento *b)* e suas respectivas inversas.

Algoritmo Ordenação por Colunas

Entrada: $S = (a_0, a_1, \ldots, a_{n-1}) : n = r.s,\ r \geq 2(s-1)^2$.
Saída: A seqüência S ordenada com $rank(a) = i + jr$.

Passo 1 Ordene cada coluna da matriz.

Passo 2 "Transponha" a matriz.

Passo 3 Ordene cada coluna da matriz.

Passo 4 "Destransponha" a matriz.

Passo 5 Ordene cada coluna da matriz.

Passo 6 Desloque os elementos $\lfloor r/2 \rfloor$ posições para frente de acordo com o *rank*.

Passo 7 Intercale, em cada coluna, a primeira com a segunda metade.

Passo 8 Desloque os elementos $\lfloor r/2 \rfloor$ posições para trás de acordo com o *rank*.

A prova de correção do algoritmo é dada em duas partes. Inicialmente vamos mostrar que após o Passo 4, cada elemento está a uma distância $(s-1)^2$, no máximo, de sua posição correta. Em seguida mostraremos que os Passos 5-8 completam a ordenação.

Lema 3.1 *Após o Passo 4 do algoritmo cada elemento está, no máximo, a uma distância* $(s-1)^2$ *de sua posição correta.*

Prova. Considere um elemento x que está na posição i, j de Q após o Passo 3 (segunda ordenação). Após o Passo 4 o elemento x é enviado a uma posição que corresponde ao *rank* de $is+j$ na lista ordenada. Da posição de x depois do Passo 3, sabemos que x é maior ou igual a pelo menos $i+1$ elementos da coluna j. Seja a_k o número desses $i+1$ elementos que vêm originalmente da coluna k da matriz (antes da matriz ser "transposta"). Por definição temos que

$$i+1 = \sum_{k=0}^{s-1} a_k.$$

Note que os elementos da k-ésima coluna depois do Passo 1 que aparecem na j-ésima coluna depois do Passo 2 são compostos do j-ésimo elemento da coluna k e a partir dali de cada s-ésimo elemento da coluna k depois do Passo 1. Assim, temos que x é maior ou igual a pelo menos $(a_k-1)s+j+1$ elementos na coluna k depois do Passo 1. Segue que o *rank* de x é pelo menos

$$\sum_{k=0}^{s-1}[(a_k-1)s+j+1]-1 = \sum_{k=0}^{s-1}(a_k-1)s + \sum_{k=0}^{s-1}[j+1]-1$$

$$= (i+1)s - s^2 + s(j+1) - 1 = si + sj - (s-1)^2.$$

Assim, a posição de x depois do Passo 4 é no máximo

$$is + j - (si + sj - (s-1)^2) = (s-1)^2 - j(s-1) \leq (s-1)^2$$

além de sua posição correta.

De forma similar podemos mostrar que o *rank* de x é no máximo $si+sj$. Assim, sua posição depois do Passo 4 é no máximo

$$si + sj - (is + j) = j(s-1) \leq (s-1)^2$$

aquém de sua posição correta.

□

Lema 3.2 *Dada uma matriz $Q_{r\times s}$ tal que todo elemento está a uma distância de no máximo $\lfloor r/2 \rfloor$ de sua posição correta, os Passos 5-8 do algoritmo ordenam os elementos da matriz Q.*

Prova. Considere uma matriz $Q_{r\times s}$ tal que todo elemento está a uma distância de no máximo $\lfloor r/2 \rfloor$ de sua posição correta. Cada elemento que depois da ordenação final pertence à metade superior da coluna j, $0 \leq j < s$, está, inicialmente, na coluna j ou na metade inferior da coluna $j-1$. Da mesma forma, cada elemento que depois da ordenação final pertence à metade inferior da coluna j está, inicialmente, na coluna j ou na metade superior da coluna $j+1$. Caso contrário, algum elemento poderia estar a uma distância maior que $\lfloor r/2 \rfloor$ de sua posição correta.

Depois do Passo 5 (ordenação) cada elemento que pertence à metade superior da coluna j está na metade superior da coluna j ou na metade inferior da coluna $j-1$. Caso isto não fosse verdade e se tal elemento x estivesse na metade inferior da coluna j, então x, todos os elementos acima de x na coluna j e todos os elementos nas colunas $0, 1, 2, \ldots, j-1$ teriam *rank* menor que $rj + r/2$, o que é impossível já que temos mais que $rj + r/2$ elementos. Caso tal x estivesse na metade superior da coluna $j-1$, então como x pertence à metade superior da coluna j haveria no máximo $r(j-1)+(r/2)-1+(r/2) = rj-1$ elementos com *rank* menor ou igual a $rj-1$, o que é também impossível dado que temos rj elementos.

Analogamente, podemos mostrar que depois do Passo 5, cada elemento que pertence à metade inferior da coluna j está na metade inferior da coluna j ou na metade superior da coluna $j+1$. Ou seja, os elementos que pertencem à metade inferior da coluna j ou à metade superior da coluna $j+1$ estão em uma destas duas metades após o Passo 5. Portanto, os Passos 5-8 são suficientes para ordenar os elementos de Q.

□

Desta forma, temos o seguinte resultado:

Teorema 3.3 *Dada uma matriz $Q_{r\times s}$ tal que $s|r$ e $r \geq 2(s-1)^2$, o algoritmo ordena os elementos da matriz Q em tempo $O(sr \lg r)$.*

Prova. A correção segue dos dois lemas anteriores. As ordenações tomam tempo $O(r \lg r)$ em cada coluna. No Passo 7 a intercalação pode ser feita em tempo $O(r)$ em cada coluna. Os demais passos gastam tempo $O(sr)$. Somando todos os tempos temos o resultado.

□

3.1.2 Implementação sob o LogP

A implementação do algoritmo de Ordenação por Colunas no modelo LogP, realizada por Dusseau *et al.* [DCSM96], alterna passos de ordenação local com passos de comu-

nicação. Dois dos quatro passos de comunicação usam comunicação do tipo todos para todos e dois usam comunicação do tipo um para um. A distribuição dos elementos entre os processadores é direta. Cada processador j recebe a coluna j da matriz $Q_{r\times s}$. Assumimos que $n \geq p^3$.

Nos passos de comunicação temos que o Passo 2, correspondente à "transposição" da matriz, equivale a uma distribuição cíclica dos dados, ou seja, sob esta distribuição cíclica, o primeiro elemento é atribuído ao primeiro processador, o segundo elemento ao segundo processador, e assim por diante. Já a operação inversa, realizada no Passo 4, equivale a uma distribuição em blocos, ou seja, os primeiros r/s elementos são atribuídos ao primeiro processador, os segundos r/s elementos para o segundo processador, e assim por diante. As operações de deslocamento para frente, no Passo 6, são realizadas usando comunicação do tipo um para um, onde o processador P_j envia a metade inferior de seus elementos para o processador $P_{(j+1) \bmod p}$. Além disso, a primeira metade de cada coluna é movida para a segunda metade e os elementos recebidos permanecem na primeira metade da coluna. Na operação de deslocamento para trás, Passo 8, cada processador P_j envia sua metade superior para o processador $P_{(j-1) \bmod p}$ e o processamento local consiste de mover a metade superior para a metade inferior da coluna e colocar os elementos recebidos na metade inferior.

Quanto às operações locais é importante observar que depois da primeira ordenação os elementos estão parcialmente ordenados. Na segunda e na terceira ordenações há p listas ordenadas de tamanho n/p em cada processador, assim um *merge* em p listas pode ser realizado. Na quarta ordenação há duas listas ordenadas de $r/2$ elementos em cada processador e uma intercalação é certamente mais eficiente que uma ordenação.

Complexidade do algoritmo

A complexidade de tempo do algoritmo é a soma dos tempos de seus oito passos incluindo comunicação e computação local.

$$T_{total}(r,s,p) = 3 \cdot T_{sort}(r) + 2 \cdot T_{distr}(r,p) + T_{shift}(r) + T_{merge}(r) + T_{unshift}(r),$$

onde:

$$\begin{aligned}
T_{sort}(r) &= O(r \lg r) \\
T_{distr}(r,p) &= (p-1)T_{troca}(r/p) \\
T_{merge}(r) &= O(r) \\
T_{shift}(r) &= r/2 + 2L + 2o + (\lceil \tfrac{r}{2w} \rceil - 1 - L/g)\max\{g, 2o\} \\
T_{unshift}(r) &= T_{shift}(r).
\end{aligned}$$

Somando todas as parcelas acima temos

$$T_{total}(r,s,p) = O(r \lg r + pL + (r - L/g)(g + o)).$$

Observe que o envio de dados no Passo 2 empacota os elementos a serem enviados a um mesmo processador em blocos contíguos de memória antes da distribuição. As operações de deslocamento consistem em enviar $r/2$ elementos para um processador e receber $r/2$ elementos de um outro processador. Esta operação tem o mesmo custo que uma troca de elementos, T_{troca}, descrita na Seção 2.3.2.

3.2 Algoritmo BSP/CGM de Ordenação

Baseado no algoritmo de Goodrich [Goo96] para o modelo BSP, Chan e Dehne [CD99] desenvolveram um algoritmo paralelo de granularidade grossa determinístico bastante simples para a ordenação de inteiros no intervalo $1, \ldots, n^c$ para alguma constante fixa c. Este algoritmo toma 6 rodadas de comunicações e $O(n/p)$ computações locais, requer $n/p \geq p^2$ e cada processador dever ter memória $O(n/p)$. O algoritmo toma n/p elementos por processador.

Este algoritmo BSP/CGM de ordenação utiliza as idéias de amostragem determinística aliadas ao método do radixsort para a ordenação seqüencial. Primeiramente, cada processador ordena seus n/p elementos usando o radixsort. Em seguida, tira uma amostragem local de $p-1$ elementos em intervalos regulares. Todos os processadores enviam sua amostragem para o processador P_0. O processador P_0, por sua vez, ordena os $p(p-1)$ elementos recebidos, seleciona uma amostragem global de $p-1$ elementos em intervalos regulares e distribui esta amostragem para todos os processadores. Na seqüência, cada processador particiona seus elementos em p blocos de acordo com a amostragem global recebida. Cada processador fica responsável por um bloco. Em uma h-relação, todo processador envia seus blocos aos processadores responsáveis por estes blocos. Finalmente, uma ordenação local dos blocos recebidos é suficiente para completar a operação.

Algoritmo BSP/CGM Ordenação

Entrada: Conjunto de elementos representado por um vetor uniformemente distribuído entre os processadores.

Saída: O vetor ordenado.

Passo 1 Cada P_k ordena localmente seus n/p elementos usando o radixsort.

Passo 2 Cada P_k seleciona de seus elementos uma amostragem local de $p-1$ elementos de *rank* $(k+1)n/(p^2-p)$, $0 \leq k < p-1$. Todos os $p-1$ elementos são enviados para P_0.

Passo 3 P_0 ordena seus $p(p-1)$ elementos recebidos no passo 2 usando o radixsort e seleciona uma amostragem global, o vetor *sample*, de $p-1$ elementos de *rank*

kp, $0 \leq k \leq p-1$. Todos os $p-1$ elementos são enviados para todos os processadores.

Passo 4 Cada P_k, depois de receber a amostra global *sample*, particiona seus n/p elementos em p blocos $B_{k,0}, \ldots, B_{k,p-1}$ de acordo com a amostra do vetor *sample*. Cada bloco $B_{k,i}$, $0 < i < p-1$, contém os elementos de P_k com valores entre $sample(i-1)$ e $sample(i)$. O bloco $B_{k,0}$ contém os elementos de P_k com valores menores que o primeiro elemento de *sample* e o bloco $B_{k,p-1}$ contém os elementos de P_k com valores maiores que último elemento de *sample*.

Passo 5 Cada P_k envia $B_{k,i}$ para o processador P_i em uma h-relação.

Passo 6 Seja R_k o conjunto de elementos recebidos por P_k no passo 5. Cada P_k ordena seqüencialmente R_k usando o radixsort.

Passo 7 Para distribuir os elementos igualmente entre os processadores, uma operação de balanceamento é realizada. Esta operação consiste dos seguintes passos:

7.1 Cada P_k envia para P_0 o valor $|R_k|$.

7.2 P_0 calcula para cada processador P_k um vetor A_k de p elementos indicando quantos de seus elementos serão enviados para o respectivo processador.

7.3 Em uma h-relação, P_0 envia A_k, $0 \leq k \leq p-1$ para cada P_k.

7.4 Cada P_k, baseado em seu A_k recebido pode realizar um balanceamento, enviando $A_k(i)$ de seus elementos para P_i.

Teorema 3.4 *O algoritmo ordena n inteiros, dentro de um intervalo $1, \ldots, n^c$, para alguma constante c, armazenados em p processadores, n/p elementos por processador, onde $n/p \geq p^2$. Além disso, o algoritmo gasta 6 rodadas de comunicação e $O(n/p)$ computação local.*

Prova. A correção do algoritmo segue da combinação do método determinístico de ordenação por amostragem com o radixsort para ordenação seqüencial. Seja $S_{k,i}$ o conjunto de elementos em P_k no fim do Passo 1 com *rank* entre $in/(p^2-p)$ e $(i+1)n/(p^2-p)$, $0 \leq i \leq p-1$. Observe que cada R_k contém no máximo $3p$ conjuntos $S_{i,j}$.

Quanto à complexidade, a quantidade de memória local e de computação local são limitadas pelo radixsort seqüencial. Para a comunicação, observe que os Passos 2 e 3 realizam uma p^2-relação cada. Isto não fere as restrições de memória, pois $n/p \geq p^2$. O Passo 5 realiza uma (n/p)-relação e o Passo 7 realiza duas p^2-relações e uma (n/p)-relação.

□

Chan e Dehne descrevem ainda um algoritmo de ordenação para o caso que $n/p \geq p$. A idéia básica deste algoritmo consiste em particionar os p processadores em $\sqrt{p}$ grupos $G_1, G_2, \cdots, G_{\sqrt{p}}$ com $\sqrt{p}$ processadores cada. A principal tarefa consiste em permutar os elementos de tal forma que todos os elementos armazenados nos processadores do grupo G_i sejam menores ou iguais a todos os elementos armazenados nos processadores dos grupos G_j para todo $i < j$. Em cada grupo podemos, assim, aplicar o algoritmo BSP/CGM que descrevemos nesta seção sem ferir as restriçoes de memória do modelo. Para mais detalhes deste algoritmo, veja [CD99].

No Capítulo 6 descrevemos os resultados experimentais de nossa implementação do algoritmo de ordenação descrito nesta seção.

Capítulo 4

Algoritmos para Grafos de Grau Limitado

Neste capítulo descrevemos dois algoritmos usando o modelo BSP/CGM para problemas em grafos cujo grau máximo é Δ. O primeiro problema, discutido na Seção 4.1, trata de encontrar uma $(\Delta + 1)$-coloração de vértices neste tipo de grafo. O segundo problema está relacionado com a coloração de vértices. Este problema, apresentado na Seção 4.2, trata de encontrar um conjunto independente maximal. Estes problemas quando estudados para um grafo em geral estão entre aqueles que, embora haja um algoritmo seqüencial trivial eficiente, não possuem algoritmos tão simples e eficientes em paralelo.

Os algoritmos descritos nas seções seguintes fazem uso da técnica de *coin tossing* determinística desenvolvida por Cole e Vishkin [CV86]. Esta técnica é um método de atribuir um rótulo a cada elemento de um conjunto de forma determinística usando para isto informações sobre a representação binária dos elementos. Em geral, os algoritmos que usam esta técnica assumem que a operação de encontrar o primeiro bit menos significativo onde dois inteiros diferem pode ser realizada em tempo constante. Neste capítulo usamos a mesma hipótese.

4.1 Coloração de Vértices

Nesta seção tratamos do problema de encontrar uma $(\Delta + 1)$-coloração em um grafo cujo grau máximo é Δ. Os algoritmos apresentados neste capítulo são uma adaptação para o modelo BSP/CGM dos Algoritmos PRAM de Goldberg *et al*[GPS88].

Uma *coloração* de um grafo G é uma atribuição $C : V \rightarrow I$ de inteiros não negativos (cores) para os vértices de V. Uma coloração é *válida* se quaisquer dois vértices adjacentes de G possuirem cores diferentes. Dizemos que uma coloração válida de G é uma k-

coloração se esta coloração pinta todos os vértices do grafo G com no máximo k cores.

Dado um grafo $G = (V, E)$, uma *pseudo-floresta* $G' = (V, E')$ de G é um grafo dirigido, com $E' \subseteq E$, onde cada vértice tem grau de saída no máximo 1. Adicionalmente, é definido para cada vértice $v \in V$ o vértice $pai(v)$ que corresponde ao vizinho de v em G' correspondente ao arco saindo de v. Caso não exista tal arco, dizemos que v é uma raiz e fazemos $pai(v) := v$.

No que segue usamos a seguinte notação:

$$\begin{aligned} \lg^{(1)} x &= \lg x \\ \lg^{(i)} x &= \lg \lg^{(i-1)} x \\ \lg^{*} x &= \min\{i \mid \lg^{(i)} x \leq 2\}. \end{aligned}$$

4.1.1 Etapas do Algoritmo BSP/CGM

Dado um grafo $G = (V, E)$ representado por sua lista de adjacência, cada processador P_k, $0 \leq k < p$, onde p é o número de processadores disponíveis, possui como entrada um conjunto de $O(n/p)$ vértices e as respectivas arestas incidentes a estes vértices, onde n é o número de vértices do grafo G. Por simplicidade, assumimos que o conjunto V é dado como uma seqüência de inteiros de 0 até $n-1$. Denotamos por $V_{P_k} = \{kn/p, \ldots, (k+1)n/p - 1\}$ e por E_{P_k}, respectivamente, os vértices de V e as arestas de E atribuídos ao processador P_k. Seja Δ o grau máximo dos vértices de G.

Em uma fase de pré-processamento, as arestas do grafo de entrada G são todas rearranjadas na forma de arcos em no máximo Δ pseudo-florestas G_i, $0 \leq i < \Delta$. Para determinar uma pseudo-floresta G_i a partir de G, cada P_k para cada v atribuído a P_k insere em G_i a i-ésima aresta $\{v, w\}$ com $v > w$ adjacente a v, se existir tal aresta, como um arco (v, w). Isto é, $pai_i(v) = w$. Caso não exista tal aresta, v é uma raiz e tem $pai_i(v) = v$. Note que para cada par $\{v, w\}$ na lista de adjacência de v com $v < w$ sabemos que $v = pai_i(w)$. Esta informação é usada pelo processador quando do envio da cor de v para o processador cujo w foi atribuído. Este procedimento para determinar as pseudo-florestas pode ser realizado em tempo $O(n/p)$ sem custo de comunicação. Nos algoritmos que seguem assumimos que esta fase de pré-processamento já foi realizada. Denotamos por C_v^i a cor de um vértice v em G_i e por $C_v^i(j)$ o j-ésimo bit da representação binária da cor de v em G_i.

A primeira etapa do algoritmo pinta cada uma destas pseudo-florestas com três cores. Após esta etapa, temos Δ pseudo-florestas distribuídas entre os p processadores, cada uma pintada com três cores. Inicialmente fazemos $G' = G_0$. A partir deste ponto, para cada pseudo-floresta G_i, $1 \leq i < \Delta$ inserimos os arcos de G_i em G', e obtemos uma $3(\Delta + 1)$-coloração de G'. Em seguida, transformamos esta $3(\Delta + 1)$-coloração em uma $(\Delta + 1)$-coloração no grafo combinado $G' \cup G_i$. Repetimos esta operação para toda G_i, $1 \leq i < \Delta$

até determinarmos uma $(\Delta+1)$-coloração de G. Na Seção 4.1.2 descrevemos um algoritmo BSP/CGM para pintar Δ pseudo-florestas cada uma com 3 cores e na Seção 4.1.3 um algoritmo BSP/CGM que dadas Δ pseudo-florestas G_i, $0 \leq i < \Delta$ cada uma com uma 3-coloração determina uma $(\Delta + 1)$-coloração em $G = \bigcup G_i$.

4.1.2 3-Coloração de Pseudo-Florestas

Este algoritmo BSP/CGM pinta Δ pseudo-florestas G_i, $0 \leq i < \Delta$ cada uma com 3 cores. As pseudo-florestas estão distribuídas entre os p processadores conforme descrito na fase de pré-processamento dada na Seção 4.1.1. Primeiro, as pseudo-florestas são pintadas com 6 cores como segue. Para cada G_i, dado um vértice v de cor C_v^i, sejam j_v^i o índice da menor posição da representação binária da cor de v que difere da representação binária da cor de seu pai, e $C_v^i(j_v^i)$ o valor do bit nesta posição. A 6-Coloração das pseudo-florestas é feita, começando com uma coloração válida qualquer e iterativamente reduzindo o número de cores através do par $\langle j_v^i, C_v^i(j_v^i)\rangle$. Este par, um vez calculado para cada vértice de G_i, determina uma nova coloração válida. Estas iterações, realizadas simultaneamente para todas as pseudo-florestas, param quando cada pseudo-floresta atinge uma 6-coloração. Depois disso, a 6-coloração das pseudo-florestas é reduzida para uma 3-coloração em 3 etapas. Em cada etapa uma cor $c = 5$, 4 ou 3 é removida. Para isso, pintamos cada vértice não-raiz, em cada G_i com a cor de seu pai, fazendo com que todos os filhos de um vértice tenham a mesma cor. Em seguida cada vértice de cor c escolhe para si a menor cor ausente em seu pai e em seus filhos. As raízes são pintadas com a menor cor disponível diferente de sua cor corrente.

Algoritmo BSP/CGM 3-coloração de pseudo-florestas

Entrada: Cada P_k com n/p vértices de cada G_i e suas respectivas arestas e $l = \lceil \lg n \rceil$.

Saída: Cada pseudo-floresta G_i com uma 3-coloração.

Passo 1 Cada P_k para cada G_i, $0 \leq i < \Delta$ e $v \in V_{P_k}$ faz $C_v^i := v$.

Passo 2 Cada P_k reduz a coloração executando os passos abaixo enquanto $l > 3$.

- **2.1** Aplique o *Procedimento Reduz_Cores* descrito abaixo obtendo uma nova coloração válida para cada G_i e um novo valor l.
- **2.2** Para cada G_i, $0 \leq i < \Delta$ e $v \in V_{P_k}$ receba a cor de $pai_i(v)$ dos processadores cujo $pai_i(v)$ foi atribuído.

Passo 3 Cada P_k aplica novamente o *Procedimento Reduz_Cores* descrito abaixo obtendo uma nova coloração válida para cada G_i e um novo valor l.

Passo 4 Cada P_k para cada G_i reduz a 6-coloração para uma 3-coloração executando os passos abaixo para $c := 5$, 4 e 3.

4.1 Para cada G_i, $0 \leq i < \Delta$ e $v \in V_{P_k}$ receba a cor de $pai_i(v)$ dos processadores cujo $pai_i(v)$ foi atribuído. Em seguida faça $C^i_{f(v)} := C^i_v$.

4.2 Para cada G_i, $0 \leq i < \Delta$ e $v \in V_{P_k}$ pinte v com a cor de $pai_i(v)$.

4.3 Para cada G_i, $0 \leq i < \Delta$ e $v \in V_{P_k}$ receba a cor de $pai_i(v)$ dos processadores cujo $pai_i(v)$ foi atribuído.

4.4 Para cada vértice v de cor c pinte v em G_i com a menor cor diferente da cor de $pai_i(v)$ e de $C^i_{f(v)}$.

Procedimento Reduz_Cores

Entrada: Cada P_k com n/p vértices de cada G_i e suas respectivas cores e o valor de l.

Saída: Cada pseudo-floresta G_i com nova coloração válida.

1: **para** $v \in V_{P_k}$ e G_i, $0 \leq i < \Delta$ **faça**
2: **se** v é raiz de G_i **então**
3: $j^i_v := 0$
4: $b^i_j := C^i_v(0)$
5: **senão**
6: $j^i_v := \min\{j \mid C^i_v(j) \neq C^i_{pai_i(v)}(j)\}$
7: $b^i_j := C^i_v(j^i_v)$
8: $C^i_v := j^i_v b^i_v$
9: $l := \lceil \lg l \rceil + 1$

Observe que no algoritmo acima, o envio e recebimento de cores pode ser feito em uma rodada de comunicação, uma vez que todos os processadores sabem a localização do pai e dos filhos de cada vértice. Além disso, este algoritmo trabalha simultaneamente com um conjunto de Δ pseudo-florestas, utilizando desta forma, granularidade grossa para a comunicação entre os processadores.

Correção e Complexidade do Algoritmo

Teorema 4.1 *O Algoritmo BSP/CGM 3-coloração de pseudo-florestas produz uma 3-coloração de Δ pseudo-florestas em $O(\lg^* n)$ rodadas de comunicação e tempo de computação $O(n/p(\Delta + \lg^* n))$.*

Prova. Primeiramente demonstramos, por indução, que o Passo 2, produz uma 6-coloração, em seguida mostramos que o Passo 4 reduz a 6-coloração para uma 3-coloração.

Inicialmente, cada $v \in V$ recebe $C_v^i = v$. Trivialmente uma coloração válida. Assim, indutivamente, assumimos que uma coloração C é válida imediatamente antes do Passo 2.1 (*Procedimento Reduz_cores*). Vamos mostrar que a coloração é válida imediatamente após este passo. Sejam v e w dois vértices adjacentes em G_i tal que $v = pai_i(w)$. Pelo *Procedimento Reduz_cores*, v escolhe algum índice t tal que $C_v^i(t) \neq C_{pai_i(v)}^i(t)$. A nova cor de w é $\langle j, C_w^i(j)\rangle$ e a nova cor de v é $\langle t, C_v^i(t)\rangle$. Caso $j \neq t$, as novas cores dos vértices são diferentes. Caso contrário $j = t$, e pela definição de j, $C_v^i(j)$ é diferente de $C_w^i(j)$. Assim, novamente as cores são diferentes. Segue que a coloração é válida.

Vamos mostrar agora que o Passo 2 efetua $O(\lg^* n)$ rodadas. Em cada iteração, no Passo 2.2, cada processador busca $\Delta n/p$ cores, para cada G_i a cor do pai de cada um dos vértices atribuídos ao processador. Seja l_h o número de bits na representação de cores depois da h-ésima rodada. Para $h = 1$ temos

$$l_1 = \lceil \lg l \rceil + 1 \leq 2\lceil \lg l \rceil,$$

se $\lceil \lg l \rceil \geq 1$.

Assuma que para algum h tenhamos $l_{h-1} \leq 2\lceil \lg^{(h-1)} l \rceil$ e $\lceil \lg^{(h)} l \rceil \geq 2$. Então

$$l_h = \lceil \lg l_{h-1} \rceil + 1 \leq \lceil \lg(2 \lg^{(h-1)} l) \rceil + 1 \leq 2\lceil \lg^{(h)} l \rceil.$$

Portanto enquanto $\lceil \lg^{(h)} l \rceil \geq 2$ teremos $l_h \leq 2\lceil \lg^{(h)} l \rceil$. Assim o número de bits decresce por $O(\lg^* n)$ rodadas até l_h ter o valor 3. Com mais a chamada do *Procedimento Reduz_Cores* do Passo 3 temos uma 6-coloração: 3 possíveis valores para o índice j_v^i e 2 possíveis valores para o bit b_v^i.

Para mostrarmos que cada uma das três iterações do Passo 4 produz uma coloração válida, basta notar que com o deslocamento da cor do pai para todos seus filhos, cada vértice será adjacente a vértices com duas cores: a de seu pai e a de seus filhos. Como cada vértice escolhe a menor cor diferente de ambas e a raiz recebe a menor cor que difere de sua cor anterior, portanto a coloração será válida. Como cada iteração realiza uma rodada de comunicação segue o resultado.

□

4.1.3 $(\Delta + 1)$-Coloração

Nesta seção estabelecemos um algoritmo BSP/CGM que, dadas Δ pseudo-florestas $G_i = (V, E^i)$, $0 \leq i < \Delta$ com uma 3-coloração cada, determina uma $(\Delta + 1)$-coloração em $G = \bigcup G_i$. Inicialmente inserimos cada arco de G_0 como uma aresta em um grafo G' que possui a mesma coloração de G_0. A partir deste ponto, para cada pseudo-floresta G_i, $1 \leq i < \Delta$ inserimos os arcos de G_i como arestas em G', e obtemos uma $3(\Delta + 1)$-coloração válida de G' pela concatenação das cores de G_i com as cores de G'. Como cada

cor forma um conjunto de vértices dois a dois não adjacentes em G', podemos pintar cada vértice deste conjunto independentemente dos demais. Isto pode ser feito da seguinte forma: para cada um dos, no máximo, $3(\Delta+1)$ conjuntos de mesma cor escolhemos, para cada um de seus vértices, uma das cores entre $0,\ldots,\Delta$ ausente nos vizinhos de v. Este passo é repetido para toda G_i, $0 < i < \Delta$ de forma a incrementar G' até serem inseridas todas as aresta de G. Assim, com $\Delta+1$ destes passos obtemos uma $(\Delta+1)$-coloração de G.

Algoritmo BSP/CGM $(\Delta+1)$-coloração

Entrada: Cada P_k possui n/p vértices de $G_i, 0 \leq i < \Delta$ e suas respectivas arestas $E^i_{P_k}$.
Saída: O Grafo G' com uma $(\Delta+1)$-coloração.

Passo 1 Cada P_k inicializa $G' := (V_{P_k}, E^0_{P_k})$ com a coloração $C := C^0$.

Passo 2 Cada P_k para cada G_i, $0 < i < \Delta$ realiza os seguintes passos:
Insira os arcos de G_i como arestas em G'.
para $j := 0$ **até** Δ **faça**
  para $l := 1$ **até** 2 **faça**
    para $v \in V_{P_k}$ **faça**
      se $C(v) = j$ e $C^i(v) = l$ **então**
        $C(v) := \max\{\{0,1,\ldots,\Delta\} - \{C(w) | \{v,w\} \in E'\}\}$
    Para cada $v \in V_{P_k}$ receba as cores dos vizinhos de v em G' dos processadores cujo $pai_i(v)$ foi atribuído.

Correção e Complexidade do Algoritmo

Teorema 4.2 *O algoritmo BSP/CGM $(\Delta+1)$-coloração produz uma $(\Delta+1)$-coloração de um grafo com grau máximo Δ em $O(\Delta^2)$ rodadas de comunicação e tempo de computação $O(\Delta^3 n/p)$.*

Prova. A correção segue do fato que em cada rodada cada cor forma um conjunto dois a dois não adjacente em G' e podendo, portanto, cada um destes conjuntos ser pintado independentemente dos demais.

Quanto à complexidade de tempo, o algoritmo realiza no máximo $2(\Delta+1)$ rodadas para cada uma das Δ pseudo-florestas, totalizando no máximo $2(\Delta^2+\Delta)$ rodadas. Como em cada uma destas rodadas o algoritmo faz consultas a, no máximo, Δ vizinhos para cada vértice atribuído ao processador, então o tempo de computação total é $O(\Delta^3 n/p)$.

□

Considerando os Teoremas 4.1 e 4.2 temos o resultado seguinte.

Teorema 4.3 *Podemos produzir uma* $(\Delta + 1)$*-coloração de um grafo com grau máximo* Δ *em* $O(\lg^* n + \Delta^2)$ *rodadas de comunicação e tempo de computação* $O(n/p(\Delta^3 + \lg^* n)$ *usando o modelo BSP/CGM.*

□

4.2 Conjunto Independente Maximal

Um *conjunto independente* de um grafo é um conjunto de vértices no qual não há arestas conectando quaisquer dois vértices deste conjunto. Um conjunto independente é maximal se não houver nenhum outro conjunto independente maior que o contenha. Um algoritmo para encontrar um *Conjunto Independente Maximal* pode ser usado para resolver uma série de problemas em redes, tais como: eleição, solução de *deadlocks* e projetos de redes de comunicação móvel. Nesta seção, mostramos como podemos encontrar um Conjunto Independente Maximal em um grafo de grau limitado no modelo BSP/CGM.

Dada uma $(\Delta + 1)$-coloração de um grafo com n vértices, podemos encontrar um Conjunto Independente Maximal deste grafo fazendo iterações sobre suas cores. Depois da coloração, nós iteramos sobre o conjunto de $\Delta + 1$ cores selecionando os vértices remanecentes da cor corrente, adicionando-os ao conjunto independente e removendo todos os seus vizinhos do grafo.

Teorema 4.4 *Dado um grafo de grau* Δ*, um Conjunto Independente Maximal pode ser encontrado no modelo BSP/CGM em* $O(\lg^* n + \Delta^2)$ *rodadas de comunicação e tempo de computação* $O(n/p(\Delta^3 + \lg^* n))$.

Prova. Usando o Teorema 4.3 o grafo pode ser colorido com $\Delta+1$ cores com $O(\lg^* n+\Delta^2)$ rodadas de comunicação e tempo de computação $O(n/p(\Delta^3 + \lg^* n)$. Depois da coloração, nós iteramos sobre o conjunto de $\Delta + 1$ cores selecionando os vértices remanecentes da cor corrente, adicionando-os ao conjunto independente e removendo todos os seus vizinhos do grafo. Pode-se mostrar por indução que o conjunto assim produzido é um Conjunto Independente Maximal. Em cada iteração as remoções e inclusões de vértices podem ser feitas em tempo $O(n/p)$ com um número constante de comunicação. Assim, como o número de cores é constante, segue o teorema.

□

Capítulo 5

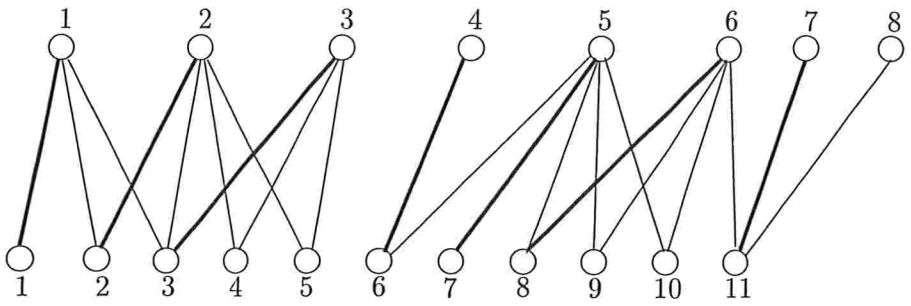
Algoritmos para Grafos Bipartidos Convexos

Seja $G = (V, W, E)$ um grafo bipartido, onde V e W formam a bipartição dos vértices e E é o conjunto de arestas da forma (v, w), onde $v \in V$ e $w \in W$. Um grafo bipartido é *convexo* se há uma ordenação dos elementos de W que satisfaz a seguinte propriedade: se $(v, w') \in E$ e $(v, w'') \in E$, com $w' < w''$, então $(v, w) \in E$ para todo $w' < w < w''$. Neste capítulo vamos denotar $|V| = n$ e $|W| = m$, além disso, por simplicidade, assumimos que os conjuntos V e W são dados como seqüências de inteiros de 1 até n e de 1 até m, respectivamente. Vamos também considerar, como é feito usualmente no desenvolvimento de algoritmos para grafos bipartidos convexos, que a ordenação dos inteiros que representam os vértices de W coincide com a ordenação de W mencionada acima. Ademais, denotaremos o conjunto de inteiros $\{i, i+1, \ldots, j\}$ por $[i, j]$.

O fato de estarmos assumindo que o conjunto W é dado de forma ordenada pressupõe que o grafo sofreu um pré-processamento. Uma forma natural de obtermos o conjunto W ordenado é através de algoritmos de reconhecimento de grafos bipartidos convexos. Algoritmos seqüenciais e paralelos (CGM) para o problema de reconhecimento podem ser encontrados em [BL76] e em [CCDP00], respectivamente.

Dessa forma, cada grafo bipartido convexo pode ser representado por um conjunto de $|V|$ triplas da forma $(v, begin(v), end(v))$, onde $v \in V$, $begin(v)$ e $end(v)$ são, respectivamente, o menor e o maior vértice de W adjacente a v. Assumimos que não há vértices isolados em G. Esta representação é chamada de representação *compacta* de G. Na Figura 5.1 ilustramos um grafo bipartido convexo, sua representação compacta e um emparelhamento máximo.

Neste capítulo apresentamos algoritmos BSP/CGM paralelos para alguns problemas em grafos bipartidos convexos. A Seção 5.1 trata de fornecer algumas preliminares. Alguns algoritmos seqüenciais são apresentados na Seção 5.2. Na Seção 5.3 descrevemos um



V	*begin*	*end*
1	1	3
2	2	5
3	3	5
4	6	6
5	6	10
6	8	11
7	11	11
8	11	11

Figura 5.1: Um grafo bipartido convexo, sua representação compacta e um emparelhamento máximo indicado por arestas em negrito

algoritmo para encontrar um emparelhamento máximo, na Seção 5.4 um algoritmo para o Problema do Conjunto Independente Máximo e na Seção 5.5 tratamos do Problema do Circuito Hamiltoniano. Descrevemos ainda, na Seção 5.6, um algoritmo para encontrar todos os caminhos mínimos em grafos bipartidos biconvexos.

5.1 Preliminares

Um *emparelhamento* M em um grafo G é um subconjunto de arestas de G tal que quaisquer duas arestas de M não são incidentes em um mesmo vértice. Um emparelhamento é dito *máximo* se tiver cardinalidade máxima. Um vértice que seja ponta de alguma aresta de M é dito estar *emparelhado* por M. Um vértice que não seja ponta de nenhuma aresta de M é chamado de *livre* em M. Se $(v, w) \in M$, dizemos que v está emparelhado com w e vice-versa.

Definição 5.1 *Um emparelhamento M em um grafo bipartido convexo é dito guloso se tem as seguintes propriedades:*

1. *se $(v, w) \in M$ então, para cada $w' \in W$, com $begin(v) \leq w' \leq w - 1$, existe $v' \in V$ tal que $(v', w') \in M$ e $end(v') \leq end(v)$;*

2. *se $w \in W$ é adjacente a um vértice livre $v' \in V$, então existe $v \in V$, com $end(v) \leq end(v')$, tal que $(v, w) \in M$.*

O nome guloso vem do fato de que podemos obter um emparelhamento percorrendo ordenadamente os elementos de W e, para cada elemento $w \in W$, dentre os vértices de V ainda não emparelhados adjacentes a w, escolhe-se um v de menor $end(v)$ e acrescenta-se (v, w) a M. Este algoritmo é conhecido como Algoritmo de Glover [Glo67], que provou que, de fato, o emparelhamento assim obtido é máximo. Dado um grafo bipartido convexo

$G = (V, W, E)$ na forma compacta, um algoritmo para o problema que roda em tempo $O(m + n\alpha(n))$ foi descrito por Lipski e Preparata em [LP81]. Este algoritmo pode ser melhorado para rodar em tempo $O(m+n)$ usando uma versão especial dos algoritmos para a união de conjuntos disjuntos (*union-find*) [GT85]. Um algoritmo com complexidade de tempo independente do tamanho de W foi relatado por Gallo [Gal84]. Este algoritmo tem complexidade de tempo $O(n \lg n)$. Na Seção 5.2.3 desenvolvemos um outro algoritmo que roda também em tempo $O(n \lg n)$. Outro algoritmo com complexidade de tempo independente de m é devido a Steiner e Yeomans [SY96] com tempo $O(n)$. Este algoritmo porém utiliza uma tabela de tamanho mn.

Na computação paralela, Dekel e Sahni [DS84] desenvolveram um algoritmo EREW PRAM para o problema que roda em tempo $O(\lg^2 n)$ usando $O(n)$ processadores. Os emparelhamentos encontrados pelos algoritmos acima são gulosos. Um algoritmo BSP/CGM descrito por Bose *et al.* [BCDL99] para um tipo de grafo bipartido convexo especial é apresentado em detalhes na Seção 5.3.1. Esse algoritmo é usado no caso geral, que está descrito na Seção 5.3.3.

O algoritmo da Seção 5.3.3 é, de fato, uma correção no algoritmo BSP/CGM de Bose *et al.* [BCDL99] para encontrar um emparelhamento máximo em grafos bipartidos convexos. Na Seção 5.3.2 descrevemos sucintamente os problemas encontrados neste artigo.

5.2 Algoritmos Seqüenciais para Emparelhamento

Nesta seção descrevemos alguns dos algoritmos seqüenciais para encontrar um emparelhamento em um grafo bipartido convexo G. Os algoritmos descritos aqui foram aqueles utilizados em nossas implementações, exceto o Algoritmo de Glover que é descrito para efeito de comparação. O Algoritmo de Steiner e Yeomans [SY96] apesar de ter complexidade de tempo $O(n)$ utiliza uma tabela de tamanho nm. Desta forma não pode ser aplicado em nossos estudos, pois o conjunto W atribuído a algum processador pode ser tão grande quanto $\Omega(m)$, ferindo a restrição de memória do modelo BSP/CGM que é $O((n + m)/p)$. O *Algoritmo Emparelhamento por Intervalos* da Seção 5.2.3 foi preferido ao Algoritmo de Gallo [Gal84] por ser de mais fácil implementação.

Os algoritmos a seguir, para garantir a complexidade, assumem que os vértices de V já estão ordenados pela função *end*. Isto implica em uma fase de pré-processamento. Nesta fase a ordenação dos vértices de V pode ser executada em tempo $O(n)$ usando Radixsort [CLR90, Cap. 9], caso $m = O(n^c)$, para alguma constante c. Caso contrário, podemos usar um algoritmo de ordenação padrão de tempo $O(n \lg n)$.

5.2.1 Algoritmo de Glover

O Algoritmo de Glover [Glo67] consiste em percorrer um a um os vértices do conjunto W. Em cada passo determina V_r, os vértices do conjunto V ainda não emparelhados e adjacentes ao $w \in W$ corrente. Caso V_r não seja vazio, seja v tal que $v \in V_r$ e $end(v) = \min_{i \in V_r}\{end(i)\}$. Isto é, v é o vértice de V ainda não emparelhado cujo *end* é o menor possível. O vértice $w \in W$ é emparelhado com $v \in V$ e v é removido do grafo. Caso V_r seja vazio, w é livre. Em seguida o algoritmo considera o próximo vértice de W.

A tarefa de maior custo deste algoritmo é organizar o conjunto V_r e determinar o elemento v de menor *end* entre os adjacentes ao vértice corrente w de W. Isto envolve para cada w percorrer todos os vértices de V adjacentes a w. Claramente esta tarefa toma tempo $O(|E|)$. Este tempo pode ser melhorado para $O(m + n \lg \lg n)$ usando estruturas de dados especiais, conforme observado em [LP81], que traz ainda uma prova simples da correção deste algoritmo.

5.2.2 Algoritmo de Lipski e Preparata

O Algoritmo de Lipski e Preparata [LP81] percorre os vértice de V em ordem crescente de *end*. Para cada $v \in V$ o algoritmo encontra o menor vértice $w \in W$ ainda não emparelhado para ser emparelhado com v. Em seguida w é removido e v incrementado de 1.

Considere $W_w := \{v \in V \mid begin(v) = w\}$, $0 \leq w \leq m$. Ou seja, W_w é o conjunto dos vértices de V cujo *begin* é w. Para escolher o vértice $w \in W$ ainda livre para ser emparelhado com v, o algoritmo mantém uma lista duplamente ligada, onde a cada posição está associada uma árvore de conjuntos disjuntos W_w (também conhecido como algoritmo *union-find*). Esta estrutura suporta as operações $find(v)$ que determina o conjunto contendo v e $union(v, w)$ que efetua a união do conjunto que contém v com o conjunto que contém w. Para cada v, o vértice w escolhido deve ser o $begin(v)$ do grafo corrente, isto é $find(v)$. Devido a remoção de vértices de W, no decorrer do algoritmo a função *begin* sofre alterações. A remoção de w do grafo é feita por meio do incremento de $begin(v)$ para todo v com $begin(v) = w$. Ou seja, é realizada a união de w e $suc(w)$, onde $suc(w)$ é o sucessor de w na lista ligada, e w é removido da lista. Como algum $v \in V$ pode tornar-se isolado, devemos verificar se $find(v) \leq end(v)$ em cada iteração antes de realizar o emparelhamento.

Considerando que a ordenação pode ser feita em tempo linear, este algoritmo toma tempo $O(m+n)$ utilizando uma versão especial para o *union-find* [GT85]. Uma prova da correção deste algoritmo pode ser encontrada em [LP81].

5.2.3 Algoritmo Emparelhamento por Intervalos

Utilizando a estrutura de dados de *árvore de intervalo* [CLR90, Cap.21], desenvolvemos um algoritmo que roda em tempo $O(n \lg n)$ independente do tamanho de m. Uma *árvore de intervalo* é uma árvore onde cada nó está relacionado a um intervalo. Nesta árvore, além dos campos *esq* e *dir* indicando os filhos esquerdo e direito, cada nó possui os campos f, o primeiro elemento do intervalo, l, o último elemento do intervalo, e max, o valor de l do intervalo com maior final da subárvore com raiz naquele nó.

Inicialmente uma árvore de intervalo T é criada com um único nó contendo o intervalo $[1, m]$. O algoritmo percorre os vértices de V por ordem crescente de *end*. Para cada $v \in V$ o algoritmo encontra o menor vértice de W ainda não emparelhado. Isto significa encontrar o menor intervalo y de T que intercepta o intervalo $[begin(v), end(v)]$. Caso não exista tal intervalo, o vértice v é livre. Caso contrário, emparelha v com $w = \max\{f(y), begin(v)\}$ e remova w do intervalo y. Após esta remoção, o intervalo y deixa de existir, devendo ser removido da árvore. Em seu lugar são inseridos os intervalos $y' = [f(y), w - 1]$ e $y'' = [w + 1, l(y)]$. Note que caso w seja um dos extremos de y, y' ou y'' é vazio e apenas um intervalo é inserido em T.

As operações de inserção e remoção de intervalos na árvore podem ser encontradas em [CLR90, Cap.21] e podem ser realizadas em tempo $O(\lg n)$.

A operação de encontrar o menor intervalo interceptando $[begin(v), end(v)]$ é bastante simples e descrita a seguir.

Procedimento Busca Intervalo Mínimo

Entrada: A raiz T da árvore de intervalo e o intervalo $[begin(v), end(v)]$

Saída: O intervalo mínimo y

1: $x := T$
2: $y := NIL$
3: **enquanto** $x \neq NIL$ e $max(x) \geq begin(v)$ **faça**
4: **enquanto** $x \neq NIL$ e $(end(v) < f(x)$ **ou** $l(x) < begin(v))$ **faça**
5: **se** $esq(x) \neq NIL$ e $max(esq(x)) \geq begin(v)$ **então**
6: $x := esq(x)$
7: **senão**
8: $x := dir(x)$
9: **se** $x \neq NIL$ **então**
10: $y := x$
11: $x := esq(x)$
12: Devolva y

É fácil de ver que este procedimento tem tempo $O(\lg n)$. Desta forma o algoritmo todo toma tempo $O(n \lg n)$ independente do conjunto W. A correção do algoritmo é direta,

pois a escolha do menor vértice livre de W ainda não emparelhado segue da correção do Algoritmo de Glover e claramente a árvore de intervalos, como descrita acima, implementa esta idéia.

5.3 Algoritmo BSP/CGM para Emparelhamento

Seja M um emparelhamento em um grafo bipartido $G = (V, W, E)$. Associamos a cada emparelhamento M uma função M tal que para cada $(x, y) \in M$, $\mathsf{M}(x) = y$ e $\mathsf{M}(y) = x$. Além disso, se $x \in V \cup W$ é livre em M, então $\mathsf{M}(x) = 0$. Estendemos essa função definindo, para $X \subseteq V \cup W$, $\mathsf{M}(X)$ como sendo o conjunto de vértices de X emparelhados por M.

Definimos V_{P_i} e W_{P_i} como sendo, respectivamente, os subconjuntos de vértices de V e de W atribuídos ao processador P_i. Essa definição é dinâmica no sentido em que os conjuntos V_{P_i} e W_{P_i} se alteram durante a execução dos algoritmos. A não ser quando especificado de outra forma, o conjunto $W_{P_i} \subseteq W$ é o conjunto de todos os vértices adjacentes a vértices em V_{P_i}.

Para cada vértice $v \in V_{P_i}$, vamos supor que o processador P_i tem acesso aos valores da tripla $(v, begin(v), end(v))$ da representação compacta do grafo. Também suporemos que o processador tem acesso ao valor de $\mathsf{M}(v)$ para os emparelhamentos envolvidos nos algoritmos. Em termos de implementação isso corresponde a alocar um registro para cada vértice de V com campos especificando os valores descritos acima. Sempre que houver uma redistribuição de vértices de V entre os processadores, devemos supor que os registros correspondentes são redistribuídos.

5.3.1 Caso Especial

Trataremos nesta seção de computar um emparelhamento máximo usando o modelo BSP/CGM para um caso especial de grafo bipartido convexo $G = (V, W, E)$ onde cada $v \in V$ tem $begin(v) = l$, para algum inteiro positivo l fixo. Note que, se os vértices de V estão ordenados pela função *end*, um algoritmo seqüencial resolve o problema em tempo $O(n)$. Basta percorrer seqüencialmente os vértices de V e, para cada $v \in V$, tentar colocar (v, w) em M, onde w é o menor vértice livre de W. Caso $w > end(v)$, então v será um vértice livre em M.

O algoritmo apresentado nesta seção foi descrito por Bose *et al.* [BCDL99] e roda em tempo de computação $T_s(n/p, m/p)$ com $O(1)$ rodadas de comunicação, onde $T_s(n, m)$ é o tempo do algoritmo de ordenação seqüencial. Este algoritmo será usado no caso geral que está descrito na Seção 5.3.3.

Algoritmo BSP/CGM Emparelhamento Especial

Entrada: Um conjunto de vértices V de um grafo bipartido convexo para o qual, para cada $v \in V$, vale que $begin(v) = l$. Os vértices são uniformemente distribuídos entre os p processadores $P_0, P_1, \ldots, P_{p-1}$.

Saída: Um emparelhamento guloso $M = \cup_{i=0}^{p-1} M_i$ distribuído entre os processadores.

Passo 1 Em paralelo, ordene os vértices de V de acordo com a função *end* usando um algoritmo paralelo BSP/CGM. Empates são quebrados pelos números dos vértices. Ao final deste passo os vértices de V estão uniformemente distribuídos entre os processadores por ordem crescente de *end*.

Passo 2 Cada P_i, $0 \leq i \leq p-1$, define $max\_end := \max\{end(v) | v \in V_{P_i}\}$ e, se $i < p-1$, P_i envia esse valor para P_{i+1}.

Passo 3 Cada P_i, $1 \leq i \leq p-1$, define max_end_ant como o valor recebido no passo anterior. Cada P_i, $0 < i \leq p-1$, define $my\_begin := max\_end\_ant + 1$. O processador P_0 define $my\_begin := l$. Cada P_i define $W_{P_i} := [my\_begin, max\_end]$.

Passo 4 Cada P_i determina seqüencialmente um emparelhamento máximo guloso para seus vértices, considerando como *begin* de cada vértice o valor my_begin. Sejam $free\_V_i$ e $free\_W_i$ os números de vértices livres de V_{P_i} e W_{P_i} respectivamente.

Passo 5 Cada P_i informa os valores de $free\_V_i$ e $free\_W_i$ para os demais processadores. Com os valores recebidos cada processador monta os vetores $Free\_V[0, p-1]$ e $Free\_W[0, p-1]$.

Passo 6 Cada P_i determina seqüencialmente o valor de *get*, o número adicional de vértices de W, além daqueles em $[my\_begin, max\_end]$, disponíveis para serem emparelhados com vértices em V_{P_i}. Esse valor é calculado em cada P_i usando o *Procedimento Calcula_Get* abaixo.

Passo 7 Cada P_i redefine W_{P_i} como sendo $W_{P_i} := [my\_begin - get, max\_end]$.

Passo 8 Cada P_i, calcula seqüencialmente um emparelhamento máximo guloso para seus vértices, considerando como *begin* de cada vértice o valor $my\_begin - get$, obtendo o emparelhamento M_i.

Procedimento Calcula_Get

Entrada: Os vetores $Free\_V[0, p-1]$ e $Free\_W[0, p-1]$ de P_i

Saída: *get*: o número adicional de vértices de W disponíveis para serem emparelhados com vértices em V_{P_i}

1: **se** $i = 0$ **então**
2: **devolva** 0
3: $get\_ant := Calcula\_Get(Free\_V, Free\_W, i-1)$
4: **se** $get\_ant \geq Free\_V[i-1]$ **então**
5: $min := Free\_V[i-1]$
6: **senão**
7: $min := get\_ant$
8: $get := Free\_W[i-1] + get\_ant - min$
9: **devolva** get

Lema 5.2 *A complexidade de tempo do Procedimento Calcula_Get é $O(p)$.*

Prova. Cada passo recursivo é feito em tempo constante e, portanto, o procedimento todo é executado em tempo $O(i) = O(p)$.

□

No lema abaixo, considere my_begin_i, my_end_i e get_i os valores de my_begin, my_end e get determinados pelo processador P_i durante a execução do algoritmo.

Lema 5.3 *O emparelhamento produzido pelo algoritmo BSP/CGM emparelhamento especial é um emparelhamento guloso.*

Prova. Inicialmente, os vértices de V são ordenados por *end*. Com isso, como o emparelhamento é guloso, os vértices em V_{P_i} terão maior prioridade a serem emparelhados do que vértices em $V_{P_{i+1}}$. Por construção, os vértices em V_{P_i} são os vértices de menor *end* adjacentes a vértices em $[my\_begin_i, max\_end_i]$ e, portanto, são os vértices a serem preferencialmente emparelhados a vértices em $[my\_begin_i, max\_end_i]$. Segue que os vértices de V_{P_i} emparelhados pelo algoritmo no Passo 4 continuarão a ser emparelhados, mas talvez com vértices menores de W.

Vamos mostrar, por indução em i, que os vértices em V_{P_i} devem ser emparelhados a partir do vértice $my\_begin_i - get_i$ de W, e que portanto o emparelhamento M_i, determinado no Passo 8 do algoritmo, é correto.

Para $i = 0$, como os vértices de V_{P_0} são os de menor *end*, o emparelhamento M_0 está correto pois são emparelhados a partir do menor vértice de W, o vértice l.

Vamos considerar agora um $i > 0$. Por indução, os vértices em $V_{P_{i-1}}$ foram emparelhados corretamente a partir do vértice $my\_begin_{i-1} - get_{i-1}$. Portanto, get_{i-1} vértices adicionais de W foram disponíveis para serem emparelhados em vértices de $V_{P_{i-1}}$. Desse modo, exatamente $\min\{get_{i-1}, free\_V_{i-1},\}$ novos vértices de V_{P_i} serão emparelhados, restando $free\_W_{i-1} + get_{i-1} - \min\{get_{i-1}, free\_V_{i-1},\}$ vértices livres em $[my\_begin_{i-1} -$

$get_{i-1}, max\_end_{i-1}]$ que poderão ser emparelhados com os vértices em V_{P_i}. É exatamente esse valor que é calculado como get_i pelo processador P_i no *Procedimento Calcula_Get*, e segue que os vértices em V_{P_i} devem ser emparelhados a partir do vértice $my\_begin_i - get_i$ de W.

□

Teorema 5.4 *O Algoritmo BSP/CGM Emparelhamento Especial encontra um emparelhamento máximo em tempo $T_s(n/p, m/p)$ e $O(1)$ rodadas de comunicação, onde $T_s(n, m)$ é o tempo de ordenação seqüencial com $n/p > p$.*

Prova. A correção do algoritmo segue do Lema 5.3. Não há rodadas de comunicação nos Passos 3, 4 e 8. Nos demais passos o número de rodadas é $O(1)$. Em cada passo o total de mensagens trocadas é $O(n/p)$. A ordenação do Passo 1 pode ser feita em tempo de computação $O(n/p)$ com número de rodadas constante usando o algoritmo BSP/CGM para ordenar n inteiros no intervalo $[1, m]$ de [CD99] descrito na Seção 3.2, caso $m = O(n^c)$, para alguma constante c. Caso não haja limite para m, a ordenação pode ser feita no modelo BSP/CGM em tempo de computação $O(n \lg n/p)$ com número de rodadas constante usando o Algoritmo de Goodrich [Goo96]. A computação local do Passo 6 é $O(p)$, pelo Lema 5.2. Para os demais passos a computação local é $O(n/p)$ e, como $n/p \geq p$, segue o teorema.

□

5.3.2 O Algoritmo de Bose *et al.*

Como o algoritmo da Seção 5.3.3 faz uma correção no algoritmo BSP/CGM de Bose *et al.* [BCDL99], nesta seção descrevemos sucintamente o algoritmo de Bose *et al.* e apresentamos, como exemplo, dois casos que não são tratados adequadamente pelo algoritmo.

Seguindo a notação de Bose *et al.*, para um dado grafo bipartido convexo $G = (V, W, E)$, seja $I(G)$ o cojunto de $|V|$ intervalos contendo para cada $v \in V$ um intervalo $I_i = (l_i, r_i)$, onde l_i e r_i são, respectivamente $begin(v)$ e $end(v)$ em W. Uma *atribuição ótima* consiste de atribuir, para um número máximo de intervalos, um rótulo inteiro para cada intervalo, de forma que cada rótulo esteja dentro daquele intervalo e que quaisquer dois intervalos não possuam o mesmo rótulo.

Note que o problema do emparelhamento máximo em G pode ser reduzido a encontrar um atribuição ótima em $I(G)$. A seguir, fazemos a descrição do *Algoritmo 3*, para encontrar uma atribuição ótima em um conjunto de intervalos. Este algoritmo faz chamadas ao *Algoritmo 1* que corresponde ao *Algoritmo BSP/CGM Emparelhamento Especial* descrito na Seção 5.3.1.

Algoritmo 3

Entrada: Um conjunto de intervalos $I(G)$ distribuído entre os p processadores.

Saída: Uma atribuição ótima.

Passo 1 Os intervalos são ordenados por l_i usando um algoritmo BSP/CGM.

Passo 2 Em seguida, os intervalos com mesmo l_i são combinados em grupos. Todos os grupos atribuídos a um único processador são combinados em um único grupo, gerando $\gamma \leq 2p + 1$ grupos.

Passo 3 A cada grupo é associado um intervalo de controle (l_I, r_I) $I = 1, \ldots, \gamma$, onde l_I é o menor l_i do grupo e $r_I = l_{I+1} - 1$, para $0 < I < \gamma$ e r_γ é o maior r_i.

Passo 4 Cada processador resolve seqüencialmente o problema para grupos de intervalos que estão inteiramente dentro de um único processador.

Passo 5 Cada processador aplica o *Algoritmo 1* para os grupos que são atribuídos a mais de um processador.

Passo 6 Os intervalos são classificados em três tipos: M, os intervalos rotulados com rótulos menores ou iguais a r_I; I, os intervalos não rotulados e com $end \leq l_I$; e T os demais intervalos. Sejam M_L e T_L os intervalos do grupo da esquerda, e sejam M_R e T_R os intervalos do grupo da direita, com intevalos de controle (l_L, r_L) e (l_R, r_R), respectivamente. Usando o *Algoritmo 1*, resolva o problema para $T_L \cup M_R$ no intervalo (l_R, r_R) obtendo M_n e T_n. Defina o grupo com $M := M_n \cup M_R$, $T := T_n \cup T_R$ e intervalo de controle (l_L, r_R).

Passo 7 Repita o Passo 6 γ vezes até obter um único grupo com um conjunto de intervalos rotulados.

Passo 8 Com um processo inverso aos passos anteriores, para cada grupo existente, seja M o conjunto de intervalos rotulados. Sejam $V = \{v \mid v \in M \text{ e } begin(v) < l_L\}$ e $W = V \cup M_L$. Para obter dois grupos, divida M em M'_L e M'_R, onde M'_L consiste apenas dos primeiros $\min\{|W|, l_R - l_L\}$ de W e $M'_R = M - M'_L$.

Passo 9 Repita o Passo 8 γ vezes até obter os γ grupos do Passo 3.

Passo 10 Execute o algoritmo seqüencial para aqueles grupos atribuídos a um único processador. Para grupos atribuídos a mais de um processador, ajuste o l_i de todos os intervalos para o início do grupo e execute o *Algoritmo 1*.

Teorema 2 *O Algoritmo 3 resolve o problema de atribuição ótima de rótulos em* $O(\lg p)$

rodadas de comunicação e tempo de computação $O(T_{seq}(n/p, m/p) + n/p \lg p)$, *onde* T_{seq} *é o tempo seqüencial para o problema.*

□

Na análise do *Algoritmo 3* descrita no Teorema 2 que aparecem em [BCDL99], há a afirmação de que ocorrem $O(1)$ chamadas do algoritmo de emparelhamento seqüencial justificando o tempo $O(T_{seq}(n/p, m/p))$. De fato, há duas chamadas do algoritmo seqüencial.

A primeira chamada do algoritmo seqüencial ocorre no Passo 4 do *Algoritmo 3*. Devido ao Passo 1, o número de intervalos em cada processador é $O(n/p)$. Porém a quantidade de vértices de W nestes intervalos pode ser tão grande quanto $\Omega(m)$, ferindo a restrição de memória do modelo BSP/CGM que é $O((n+m)/p)$. Para resolver este problema, basta usar um algoritmo de emparelhamento que dependa apenas de n/p. Este é o caso do Algoritmo de Gallo [Gal84] que tem completidade $O(n/p \lg(n/p))$.

A segunda chamada do algoritmo seqüencial ocorre no Passo 10. Neste passo vemos dois problemas. Primeiro, o número de intervalos que é redistribuído para um processador poder ser tão grande quanto $\Omega(n)$ ferindo a restrição de memória do modelo. Além disso, novamente a quantidade de vértices de W nestes intervalos pode ser $\Omega(m)$.

Decrevemos um exemplo que ilustra os dois problemas. Considere o seguinte grafo com 5000 intervalos. Para $i = 1, \ldots, 1000$, $(l_i, r_i) = (1, 5000)$. Para $i = 2, \ldots, 1000$, $(l_i, r_i) = (2, 5000)$. Para $i = 3, \ldots, 1000$, $(l_i, r_i) = (3, 5000)$. Para $i = 4, \ldots, 1000$, $(l_i, r_i) = (4, 5000)$. Para $i = 5, \ldots, 1000$, escolha intervalos arbitrários (l_i, r_i), com cada $l_i \geq 5$ e $r_i \leq 5000$.

Agora, suponha que temos 5 processadores para resolver o problema ($P_1, \ldots, P_5$, confome notação do artigo). Pelo Passo 1, os primeiros 1000 intervalos são atribuídos ao processador 1, os 1000 intervalos seguintes para o processador 2 e assim por diante. Assim, temos para o Passo 3 os seguintes intervalos de controle (*controlled ranges*): $(l_1, r_1) = (1, 1)$, $(l_2, r_2) = (2, 2)$, $(l_3, r_3) = (3, 3)$, $(l_4, r_4) = (4, 4)$, $(l_5, r_5) = (5, 5000)$.

Desta forma o intervalo de controle para o processador 5 é muito grande, não respeitando a restrição de memória do modelo BSP/CGM, tornado o Passo 4 inviável. Continuando a simulação do algoritmo, vemos que no Passo 10, o processador 5 precisa resolver o problema do emparelhamento seqüencial para 4996 intervalos. Além disso, o intervalo de controle deste intervalos é $(5, 5000)$.

5.3.3 Caso Geral

Seja $G = (V, W, E)$ um grafo bipartido convexo. Dado como entrada o conjunto V uniformemente distribuído entre os p processadores $P_0, P_1, \ldots, P_{p-1}$ representando G, o

algoritmo para encontrar um emparelhamento máximo guloso vai particionar V e resolver recursivamente o problema para cada uma das partes. O conjunto V é particionado em duas partes $V_l := \cup_{i=0}^{k-1} V_{P_i}$ e $V_r := \cup_{i=k}^{p-1} V_{P_i}$, onde $k := \lceil p/2 \rceil$, de forma que cada vértice em V_l tenha o *begin* menor ou igual a cada vértice de V_r.

Uma vez encontrados os emparelhamentos M_l e M_r, respectivamente de V_l e V_r, temos de obter o emparelhamento M para o grafo todo. Embora os vértices de V emparelhados por M_l e M_r sejam disjuntos o mesmo não ocorre com os vértices de W, pois podem existir vértices de W emparelhados por M_l e M_r e, portanto, $M_l \cup M_r$ pode não ser um emparelhamento. Para corrigir isso, depois de encontrados M_l e M_r o algoritmo aplica o *Procedimento Ajusta Emparelhamento.*

O *Algoritmo BSP/CGM Emparelhamento Máximo*, descrito abaixo, faz a ordenação dos vértices de V e chama o *Procedimento Emparelhamento Recursivo.*

O *Procedimento Emparelhamento Recursivo*, por sua vez, determina M_l e M_r. Antes de o *Procedimento Ajusta Emparelhamento* ser chamado, mais alguns cálculos são realizados. Um conjunto T é determinado, consistindo de vértices de V emparelhados por M_l a vértices de W que também podem estar emparelhados por M_r. Mais precisamente, seja R_l, o menor vértice de W emparelhado por M_r. Então T é o conjunto de vértices de V emparelhados por M_l com vértices de W maiores ou iguais a R_l. Além disso, um conjunto de arestas M_s é determinado usando o *Algoritmo Emparelhamento Especial* para vértices em $V' = \mathsf{M_r}(V_r) \cup T$. Conforme resultado descrito por Dekel e Sahni [DS84], e enunciado no Teorema 5.5, $(\mathsf{M_l}(V_l) \backslash T) \cup \mathsf{M_s}(V')$ contém exatamente os vértices de V de um emparelhamento máximo guloso. O problema é que M_s não é um emparelhamento legítimo em G. Podem existir arestas em M_s que não estão em E, pois para encontrarmos M_s os valores da função *begin* são alterados para valores menores do que os reais e alguns vértices de V podem acabar ficando "emparelhados" com vértices aos quais não são adjacentes.

Para corrigir essa situação, o *Procedimento Ajusta Emparelhamento* é usado. Este procedimento usa várias propriedades discutidas em detalhes na prova de correção do algoritmo.

Algoritmo BSP/CGM Emparelhamento Máximo

Entrada: Um conjunto de vértices V de um grafo bipartido convexo uniformemente distribuído entre os p processadores $P_0, P_1, \ldots, P_{p-1}$.

Saída: Um emparelhamento guloso $M = \cup_{i=0}^{p-1} M_i$ distribuído entre os processadores.

Passo 1 Em paralelo, ordene os vértices de V de acordo com a função *begin* usando um algoritmo BSP/CGM. Empates são quebrados de acordo com a função *end.* Persistindo o empate, o desempate é decidido pelos números dos vértices. Ao final deste passo os vértices de V estão uniformemente distribuídos entre os processadores por ordem crescente de *begin.*

Passo 2 Em paralelo, obtenha um emparelhamento máximo guloso M_i usando o *Algoritmo Emparelhamento Recursivo* descrito a seguir.

O *Algoritmo BSP/CGM Emparelhamento Recursivo* e o *Procedimento Ajusta Emparelhamento* são detalhados a seguir. É importante observar aqui que o valor de p deve ser entendido como um parâmetro dessas funções. Na primeira chamada do *Algoritmo BSP/CGM Emparelhamento Recursivo*, p é o número total de processadores disponíveis. Porém, nas chamadas recursivas, p é o tamanho do grupo de processadores determinado no Passo 2 do algoritmo.

Algoritmo BSP/CGM Emparelhamento Recursivo

Entrada: Um conjunto de vértices V de um grafo bipartido convexo. Os vértices de V são uniformemente distribuídos entre os processadores $P_0, P_1, \ldots, P_{p-1}$ e estão ordenados de acordo com a função *begin*.

Saída: Um emparelhamento guloso $M := \cup_{i=0}^{p-1} M_i$ distribuído entre os processadores.

Passo 1 Se $p = 1$, então o processador resolve o problema seqüencialmente e devolve o emparelhamento M_i encontrado. Para resolver o problema seqüencialmente, se $|W_{P_i}| \leq c \cdot |V_{P_i}|$, para alguma constante c previamente fixada, o Algoritmo de Lipski e Preparata [LP81] é usado; caso contrário, o algoritmo da Seção 5.2.3 é usado.

Passo 2 Se $p > 1$, os processadores são divididos em 2 grupos: $P_0, P_1, \ldots, P_{k-1}$ e P_k, $P_{k+1}, \ldots, P_{p-1}$, onde $k := \lceil p/2 \rceil$. Sejam $V_l := \cup_{i=0}^{k-1} V_{P_i}$ e $V_r := \cup_{i=k}^{p-1} V_{P_i}$. Em paralelo, cada grupo chama recursivamente este algoritmo, obtendo emparelhamentos $M_l := \cup_{i=0}^{k-1} M_{l_i}$ e $M_r := \cup_{i=k}^{p-1} M_{r_i}$ distribuídos entre os processadores.

Passo 3 Seja R_l o número do menor vértice de W emparelhado por M_r. O valor de R_l é calculado por P_k e divulgado entre os processadores.

Passo 4 Cada P_i, $i = 0, \ldots, k-1$ determina um conjunto T_i definido como o conjunto de vértices de V emparelhados por M_{l_i} com vértices de W com números maiores ou iguais a R_l. Seja $T := \cup_{i=0}^{k-1} T_i$.

Passo 5 Em paralelo, os processadores aplicam o *Algoritmo Emparelhamento Especial* para o conjunto $T \cup \mathsf{M}_r(V_r)$ considerando como *begin* de cada vértice o valor R_l, obtendo o emparelhamento $M_s := \cup_{i=0}^{p-1} M_{s_i}$ distribuído entre os processadores.

Passo 6 Em paralelo, os processadores aplicam o *Procedimento Ajusta Emparelhamento* descrito abaixo tendo como entrada T, M_r, M_s e o valor de R_l. Como resultado, cada P_i obtém um emparelhamento guloso M_{f_i}. Cada P_i define $M_i := M_{f_i} \cup \overline{M_{l_i}}$, onde $\overline{M_{l_i}}$ é o emparelhamento resultante da remoção de arestas em M_{l_i} contendo vértices de T.

A Figura 5.2 ilustra o funcionamento do *Algoritmo BSP/CGM Emparelhamento Máximo.*

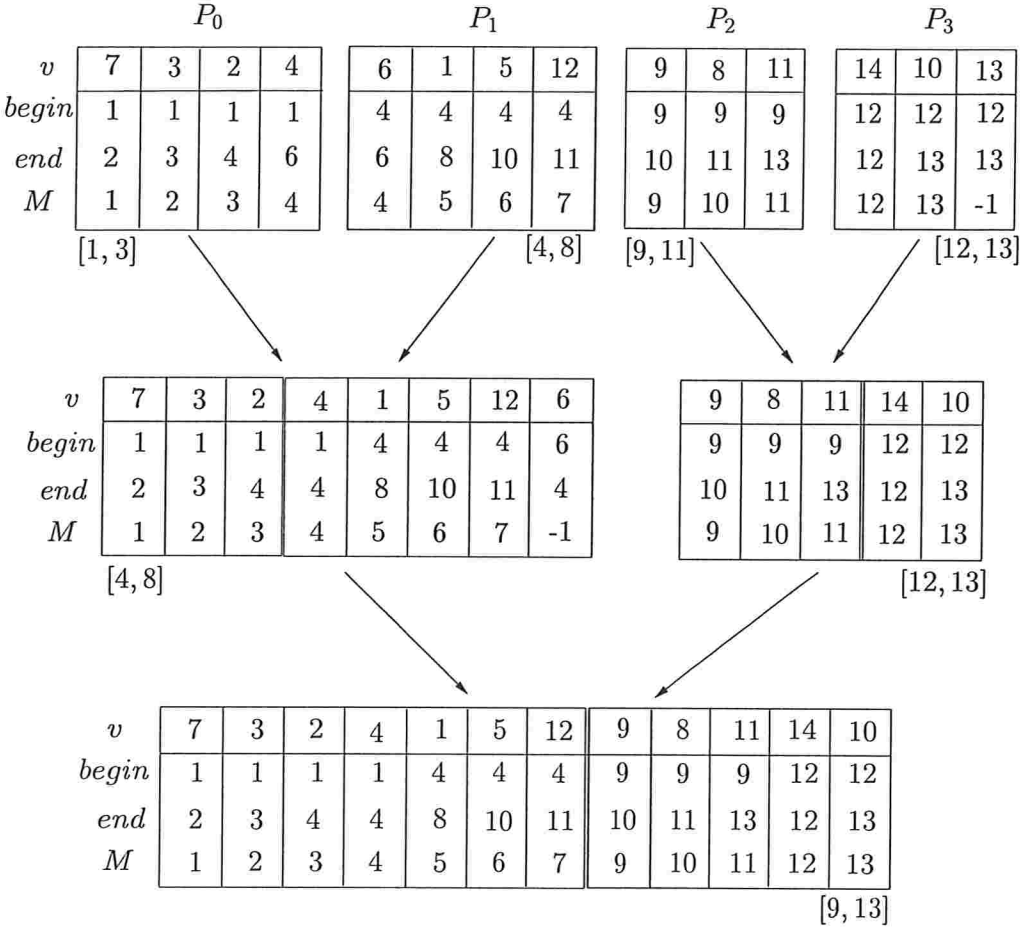


Figura 5.2: Ilustração do *Algoritmo BSP/CGM Emparelhamento Máximo* executado com 4 processadores. Os valores entre colchetes definem o R_l e o $R_{l+1} - 1$ de cada grupo.

Procedimento Ajusta Emparelhamento

Entrada: Descrita no Passo 6 do *Algoritmo Emparelhamento Recursivo.*

Saída: Um emparelhamento $M_f := \cup_{i=0}^{p-1} M_{f_i}$ distribuído entre os processadores.

Passo 1 Cada P_i define $slice := (|M_r| + |T|)/p + 1$ e $W_{P_i} = [R_l + i \cdot slice, R_l + (i+1) \cdot slice - 1]$.

Passo 2 Para cada $v \in M_r \cap M_s$ atribuído a P_i, se $\mathsf{M_s}(v) \leq \mathsf{M_r}(v)$ envie v para o processador P_k tal que $\mathsf{M_r}(v) \in W_{P_k}$.

Passo 3 Cada P_i calcula S_{i_j}, $0 \leq j \leq i$, o conjunto de vértices de V_{P_i} que podem ser emparelhados com vértices em W_{P_j} mas não com vértices em $W_{P_{j-1}}$. Isso pode ser feito da seguinte forma: para cada $v \in V_{P_i}$ se $v \in T$ inclua v em S_{i_0}, senão inclua v em S_{i_k}, onde $k := (\max\{\mathsf{M_r}(v), \mathsf{M_s}(v)\} - R_l)/slice$. Distribua o tamanho destes conjuntos para cada um dos processadores. Com os valores recebidos, cada processador monta uma matriz A $p \times p$ triangular onde $A[i,j]$, $j \leq i$, tem o valor de $|S_{i_j}|$.

Passo 4 Cada P_i aplica o procedimento a seguir sobre a matriz A obtendo quantos dos vértices que lhe podem ser enviados ele de fato consegue emparelhar.

para $j := 0$ **até** $p-1$ **faça**
  $hole := slice$
  para $k := j$ **até** $p-1$ **faça**
    se $hole \geq A[k,j]$ **então**
      $hole := hole - A[k,j]$
    senão
      $A[k,j+1] := A[k,j+1] + A[k,j] - hole$
      $A[k,j] := hole$
      $hole := 0$

Passo 5 Cada P_i redistribui os vértices de V_{P_i} enviando para cada P_j, $0 \leq j \leq i$, $A[i,j]$ vértices na seguinte ordem. Percorrendo ordenadamente por *end* a lista de vértices de V_{P_i}, envie os primeiros $A[i,j]$ vértices que podem ser emparelhados com vértices em W_{P_j} para P_j.

Passo 6 Cada P_i resolve o problema seqüencialmente e determina o emparelhamento M_{f_i}. Para resolver o problema seqüencialmente, se $|W_{P_i}| \leq c \cdot |V_{P_i}|$, para alguma constante c previamente fixada, o Algoritmo de Lipski e Preparata [LP81] é usado; caso contrário, o algoritmo da Seção 5.2.3 é usado.

A Figura 5.3 detalha o funcionamento do *Procedimento Ajusta Emparelhamento* nos processadores P_3 e P_4 para o exemplo da Figura 5.2.

Correção e Complexidade do Algoritmo Emparelhamento Máximo

Da forma como o Algoritmo de Glover opera, é fácil ver que arestas em M_l que não conflitam com arestas de M_r continuam a fazer parte de um emparelhamento máximo guloso. Lembrando que T é o conjunto de vértices de V_l que podem estar emparelhados por M_l com vértices já emparelhados por M_r, seja $V' = T \cup \mathsf{M_r}(V_r)$. Para determinarmos os vértices de V' que são emparelhados por um emparelhamento guloso, basta usarmos o procedimento descrito em Dekel e Sahni [DS84] ou seu correspondente paralelo que

Entrada
$[P_2, P_3]$

v	9	8	11	14	10
begin	9	9	9	12	12
end	10	11	13	12	13
M_r	9	10	11	12	12

$[9, 13]$

Passo 2

	P_2		P_3		
v	9	8	14	11	10
begin	9	9	12	9	12
end	10	11	12	13	13
M_s	9	10	11	12	13

$[9, 11]$ $[12, 13]$

Passo 5

	P_2		P_3		
v	9	8	11	14	10
begin	9	9	9	12	12
end	10	11	13	12	13
M_r	9	10	11	12	13

$[9, 11]$ $[12, 13]$

Passo 6

	P_2		P_3		
v	9	8	11	14	10
begin	9	9	9	12	12
end	10	11	13	12	13
M_f	9	10	11	12	13

$[9, 11]$ $[12, 13]$

Figura 5.3: Principais passos do *Procedimento Ajusta Emparelhamento* para o grafo da Figura 5.2 nos processadores P_2 e P_3 (durante a primeira chamada do procedimento).

descrevemos na Seção 5.3.1, considerando o *begin* de cada vértice em V' como sendo R_l. Citamos o resultado obtido por Dekel e Sahni [DS84].

Teorema 5.5 *O emparelhamento M_s contém exatamente os vértices de V' emparelhados pelo Algoritmo de Glover.*

□

Sabemos, portanto, que cada vértice de $\mathsf{M_s}(V')$ pode ser emparelhado por um emparelhamento guloso com um vértice distinto de W. O problema é que não sabemos exatamente com qual vértice de W cada vértice de $\mathsf{M_s}(V')$ será emparelhado. Em particular, pode ser o caso que, para algum $v \in V'$, $begin(v) > \mathsf{M_s}(v)$. A tarefa de determinar o emparelhamento correto é feita pelo *Procedimento Ajusta Emparelhamento.*

Seja M^* um emparelhamento máximo guloso para os vértices de V', considerando que, para cada $v \in V'$, $begin(v) \geq R_l$. Vamos provar que $M_f = M^*$. Para tanto, vamos estabelecer relações entre os emparelhamento M_r, M_s e M^*. Antes, porém, precisamos introduzir a noção de *prioridade.*

Se dois vértices $v, v' \in V$ são adjacentes a um vértice livre w, dizemos que v tem maior *prioridade* do que v' se num emparelhamento guloso v seria emparelhado com w e

v' não. Mais precisamente, para exprimir essa relação, definimos a função *prior* tal que, se $prior(v) > prior(v')$, então (i) $end(v) < end(v')$; ou (ii) $end(v) = end(v')$ e $v < v'$. A segunda condição tem a finalidade de facilitar a descrição dos nossos algoritmos quando um dos dois vértices, v ou v', pode ser emparelhado com w num emparelhamento guloso: o vértice a ser emparelhado com w será o de maior prioridade.

Lema 5.6 *Para cada* $v \in \mathsf{M_r}(V_r) \cap \mathsf{M^*}(V')$, $\mathsf{M^*}(v) \geq \mathsf{M_r}(v)$.

Prova. A prova é por contradição. Escolha o vértice $v \in \mathsf{M_r}(V_r) \cap \mathsf{M^*}(V')$ com o menor valor de $\mathsf{M_r}(v)$ tal que $\mathsf{M^*}(v) < \mathsf{M_r}(v)$.

Seja $w = \mathsf{M^*}(v)$. Como v não foi escolhido por M_r para ser emparelhado com w, então existe $v' \in M_r$ tal que $\mathsf{M_r}(v') = w$ e $prior(v') > prior(v)$. Além disso, pela nossa escolha de v, $\mathsf{M^*}(v') \geq \mathsf{M_r}(v)$ ou $v' \notin M^*$. Em ambos os casos segue que quando w foi emparelhado com v por M^*, v' estava livre e na vizinhança de w, e portanto $prior(v') < prior(v)$, uma contradição.

□

Corolário 5.7 *Para cada* $v \in \mathsf{M_r}(V_r)$, *fazendo* $begin(v) := \mathsf{M_r}(v)$ *o emparelhamento guloso* M^* *não é alterado.*

□

Lema 5.8 *Sejam* $v, v' \in V$ *vértices adjacentes a algum vértice* $w \in W$. *Se após o Passo 5 do Procedimento Ajusta Emparelhamento,* $v \in V_{P_i}$, $w \in W_{P_i}$ *e* $v' \in V_{P_j}$, *com* $i < j$, *então* $prior(v) > prior(v')$.

Prova. Na entrada do *Procedimento Ajusta Emparelhamento*, todos os vértices estão distribuídos aos processadores ordenados pelos valores da função *end* e, portanto, ordenados decrescentemente por prioridades.

Duas distribuições de vértices alteram essa ordem relativa no procedimento: no Passo 2 e no Passo 5.

Vamos mostrar que após a execução do Passo 2 a conclusão do lema é válida. Seja $max\_end_{P_i}$ o maior *end* dos vértices em V_{P_i}. Vamos imaginar que neste passo vértices são enviados um a um para os processadores. Conforme observado, inicialmente o lema é válido. Suponha que após um número arbitrário de envios de vértices o lema ainda continue válido e um vértice $x \in V$ é enviado a um processador. Note que para x ser enviado a um processador x tem que pertencer a M_r e o processador destino tem índice maior do que o do processador ao qual x está atribuído. Lembrando o Corolário 5.7, vamos considerar $begin(x) = \mathsf{M_r}(x)$. Se x faz o papel de $v \in V_{P_i}$ do enunciado, a ordem relativa

entre os vértices distribuídos ao processador P_j não se altera, pois $i < j$. Portanto, se v' não é um vértice que foi enviado a P_j por outro processador, vale que $prior(v) > prior(v')$ pois essa condição valia inicialmente. Mas se v' foi enviado a P_j em algum momento, então não é possível que v' seja adjacente a nenhum $w \in W_{P_i}$, pois $max\_end_{P_i} < begin(v') = \mathsf{M_r}(v')$. Se x faz o papel de $v' \in V_{P_j}$, também não é possível que x seja adjacente a w para algum $w \in W_{P_i}$, pois $max\_end_{P_i} < begin(x)$. Portanto, depois do Passo 2 o lema continua válido.

No Passo 5 vértices atribuídos a P_k somente podem ser enviados a processadores com índices menores do que k, devido ao fato de poderem ser emparelhados com vértices menores do que os vértices em W_{P_k} e de haver vértices livres de W atribuídos a processadores de menor índice do que k. Como a tentativa é enviar os vértices na ordem em que aparecem em P_k e P_k já está ordenado por prioridade, sempre que possível a ordem relativa dos vértices é mantida. A exceção — ocorre inversão de ordem relativa — fica por conta dos vértices $x \in V_{P_k}$ que não podem ser enviados a um processador $P_{k'}$, com $k' < k$, pois $begin(x) > max\_end_{P_k}$. Usando as mesmas observações do parágrafo anterior com respeito a x, podemos concluir que as prioridades se mantêm.

□

Lema 5.9 $M_f = M^*$.

Prova. A prova é por contradição. Supondo $M_f \neq M^*$, podemos escolher o menor $w \in W$ para o qual M_f e M^* não coincidem. Precisamos considerar 3 possibilidades.

Caso 1: para algum v, $w = \mathsf{M_f}(v)$ e w é livre em M^*.

A entrada do *Procedimento Ajusta Emparelhamento* é um conjunto de vértices V definido pelo *Algoritmo Emparelhamento Especial* e portanto todos os vértices de V são emparelháveis por um emparelhamento máximo. Como M^* é emparelhamento máximo, $\mathsf{M^*}(v) = w'$ para algum $w' \neq w$. Pela escolha de w, $w' > w$, o que mostra que M^* não é um emparelhamento guloso pois o item 1 da Definição 5.1 não é respeitado. Uma contradição.

Caso 2: para algum v e v', com $v \neq v'$, $w = \mathsf{M^*}(v) = \mathsf{M_f}(v')$.

Seja k o índice do processador ao qual foi atribuído a fatia contendo o vértice w. Seja i tal que $v \in V_{P_i}$.

Subcaso 1 $i = k$.

Como $v, v' \in V_{P_k}$, podemos concluir do Passo 6 do procedimento que w foi emparelhado por M_f com um vértice v' de V_{P_k} com $prior(v') > prior(v)$. O fato de que $\mathsf{M^*}(v') \neq w$ contradiz a hipótese de M^* ser um emparelhamento guloso.

Subcaso 2 $i < k$.

Pela minimalidade de w, M^* e M_f coincidem em todos os vértices emparelhados de V_{P_i}. Portanto, v é um vértice livre em M_f. Como a cardinalidade de V_{P_i} não é maior do que *slice*, temos que existe algum vértice livre w' em W_{P_i}. Como v não foi emparelhado com w', e v é adjacente a $w > w'$, podemos concluir que v não é adjacente a w' e que $\mathsf{M_r}(v) > w'$. Segue, pelo Lema 5.10, que v é emparelhado por M_f, uma contradição.

Subcaso 3 $i > k$.

Pela escolha de w sabemos que $\mathsf{M}^*(v') > w$. Como v e v' são adjacentes a w e M^* é guloso, temos que $prior(v) > prior(v')$. Por outro lado, como $k < i$, $v' \in V_{P_k}$ e $v \in V_{P_i}$, pelo Lema 5.8 temos que $prior(v') > prior(v)$, uma contradição.

Caso 3: para algum v, $w = \mathsf{M}^*(v)$ e w é livre em M_f.

Seja k o índice do processador ao qual foi atribuído a fatia contendo o vértice w. Seja i tal que $v \in V_{P_i}$.

Subcaso 1 $i = k$.

Como $v \in V_{P_k}$, e $w \in W_{P_k}$, ambos os vértices foram atribuídos ao processador P_k, mas não foram emparelhados por M_f. Então, v não pode também ser livre em M_f, pois nesse caso v teria sido emparelhado com w no Passo 6 do procedimento. Pela escolha de w, v foi emparelhado com um vértice maior do que w, o que mostra que M_{f_k} não é um emparelhamento guloso pois o item 1 da Definição 5.1 não é respeitado. Uma contradição.

Subcaso 2 $i < k$.

Idêntico ao Subcaso 2 do Caso 2.

Subcaso 3 $i > k$.

No Passo 4 é feita a tentativa de se enviar *slice* vértices pelos processadores P_j, com $j \geq k$, ao processador P_k. Podemos supor que essa tentativa foi bem sucedida, pois caso contrário v seria enviado a P_k. Portanto, como w é livre em M_f, existe um vértice livre $x \in V_{P_k}$ em relação a M_f, com x não adjacente a w. Se $x \in M_r$ e $\mathsf{M_r}(x) > w$, o Lema 5.10 implica que x é emparelhado por M_f, uma contradição.

Vamos supor, então, que $end(x) < w$. Como todo vértice em $\mathsf{M_s}(V)$ é emparelhável, $\mathsf{M}^*(x) = y > 0$. Note que, como $end(x) < w$, temos que $y < w$. Como, para todo vértice de W menor do que w os emparelhamentos M_f e M^* coincidem, $\mathsf{M_f}(x) = \mathsf{M}^*(x) = y$, uma contradição pois x é livre em M_f.

□

Lema 5.10 *Seja $V_{P_i} \subseteq V$ o conjunto de vértices atribuídos ao processador P_i após o Passo 5 do Algoritmo Ajusta Emparelhamento. Seja $w \in W_{P_i}$ um vértice livre em relação a M_f. Então, para cada $v \in \mathsf{M_r}(V_{P_i})$, com $\mathsf{M_r}(v) > w$, vale que v é emparelhado por M_f.*

Prova. Seja $max\_end_{P_i}$ o maior *end* dos vértices em V_{P_i}. Observe que:

(i) como w é livre, nenhum vértice de V_{P_i} emparelhado por M_f a vértices em $[w + 1..max\_end_{P_i}]$ é adjacente a w;

(ii) de *(i)* obtemos que nenhum vértice em $[w + 1..max\_end_{P_i}]$ pode ser emparelhado com algum vértice $t \in T$, pois se isso ocorresse t seria adjacente a w;

(iii) pela distribuição dos vértices no Passo 5 do algoritmo, cada vértice de M_r enviado a V_{P_i} tem $M_r(x) \leq max\_end_{P_i}$;

Segue de *(ii)* e *(iii)* que cada vértice de M_r enviado a V_{P_i} com $\mathsf{M_r}(v) > w$ pode ser emparelhado com esse mesmo vértice por M_f. Como M_{f_i} é emparelhamento máximo, concluímos que todos esses vértices são emparelhados por M_f.

□

Desta forma temos que o *Procedimento Ajusta Emparelhamento* de fato toma os dois emparelhamentos M_s e M_r e o conjunto T_l e encontra corretamente um emparelhamento guloso de $T \cup \mathsf{M_r}(V_r)$. Assim, os teoremas abaixo resumem os resultados obtidos. No restante desta seção denotamos por $T_s(n, m)$ o tempo de ordenação seqüencial e por $T_m(n, m)$ o tempo de encontrar um emparelhamento máximo seqüencialmente.

Teorema 5.11 *O Procedimento Ajusta Emparelhamento encontra um emparelhamento guloso em $T \cup \mathsf{M_r}(V_r)$ sobre o intervalo de W começando em R_l em $O(1)$ rodadas de comunicação e tempo $O(\max_{P_i}\{T_m(|V_{P_i}|, |W_{P_i}|)\})$ de computação local usando p processadores com $n/p > p^2$.*

Prova. A correção segue do Lema 5.9. A complexidade de tempo é dada pela soma dos passos. O Passo 2 toma tempo de computação $O(n/p)$ e 1 rodada de comunicação. É fácil ver que o Passo 3 toma tempo $O(n/p + p^2)$ com 1 rodada de comunicação e o Passo 4 toma tempo $O(p^2)$. O Passo 5 pode ser realizado em tempo $O(n/p)$ com 1 rodada de comunicação. Senão vejamos: criamos uma lista auxiliar e percorremos a lista dos vértices de V_{P_i}. Note que a soma prefixa $I_k = |S_{i_0}| + |S_{i_1}| + \ldots + |S_{i_k}|$ corresponde ao número de vértices de P_i a serem enviados ao processador P_j, $j \leq k$, e portanto indica a posição

inicial na lista auxiliar dos vértices a serem enviados para P_{k+1}. Cada vez que um vértice é determinado para ser enviado a um dado P_k, o resultado da soma I_k é incrementado de um de forma a sabermos a posição corrente da lista que será inserido o próximo vértice. De fato, a complexidade é dominada pelo Passo 6, a saber $O(\max_{P_i}\{T_m(|V_{P_i}|, |W_{P_i}|)\})$. Como temos $n/p > p^2$ seque o resultado.

□

Teorema 5.12 *O Algoritmo BSP/CGM Emparelhamento Máximo resolve o problema do emparelhamento máximo com computação local* $O((\max_{P_i}\{T_m(|V_{P_i}|, |W_{P_i}|)\} + n/p)\lg p)$ *e* $O(\lg p)$ *rodadas de comunicação com* $n/p > p^2$.

Prova. Com o Lema 5.5 e o Teorema 5.11 a correção pode ser facilmente mostrada por indução no nível da árvore de computação. Quanto a complexidade de tempo, no Passo 1 a ordenação pode ser feita em tempo de computação $T_s(n/p, m/p)$ e $O(1)$ rodadas de computação usando um algoritmo BSP/CGM. O *Algoritmo Emparelhamento Recursivo* pode ser chamado $O(\lg p)$ vezes, pelo Passo 2. Em cada chamada os Passos 5 e 6, que chamam o *Algoritmo Emparelhamento Especial* e *Procedimento Ajusta Emparelhamento* respectivamente, realizam $O(1)$ rodadas de comunicação com tempo de computação $O((\max_{P_i}\{T_m(n/p, |W_{P_i}|)\} + n/p)$. Portanto, como o número total de rodadas de comunicação é $O(\lg p)$ temos o resultado.

□

5.4 Conjunto Independente Máximo

Nesta seção tratamos do problema de encontrar um conjunto independente máximo — *MIS* — em um grafo bipartido convexo. Considerando $m = O(n^c)$, dado um grafo G bipartido convexo na forma compacta e um emparelhamento máximo M de G descrevemos um algoritmo seqüencial de complexidade $O(n)$. Este algoritmo é mais rápido que os previamente conhecidos. Baseado nesse algoritmo descrevemos um importante resultado, a saber, um algoritmo paralelo BSP/CGM ótimo que gasta computação local $O(n/p)$ e rodadas de comunicação $O(1)$, com $n/p \geq p$, onde p é o número de processadores [SS01].

5.4.1 Preliminares

Relembrando da Seção 4.2, um *conjunto independente* de um grafo é um conjunto de vértices no qual não há arestas conectando quaisquer dois vértices deste conjunto. Um *conjunto independente máximo* — MIS — é um conjunto independente de cardinalidade

máxima. Note a diferença entre um conjunto independente máximo e um conjunto independente maximal.

Dado um grafo bipartido convexo G na forma compacta e um emparelhamento máximo guloso M de G, um algoritmo seqüencial $O(n+m)$ para encontrar um MIS em G foi descrito por Lipski e Preparata em [LP81]. Na computação paralela, Czumaj *et al.* [CDP96] desenvolveram um algoritmo PRAM ótimo que roda em tempo $O(\lg n)$.

Seja $G = (V, W, E)$ um grafo bipartido convexo e M um emparelhamento em G. Um *caminho alternante* em M é um caminho em G com início em um vértice livre de V em M e cujas arestas estão alternadamente em M e em $E \setminus M$. Ou seja, se e e e' são arestas consecutivas de um caminho alternante, então $e \in E$ e $e' \in E \setminus M$, ou vice-versa. Um vértice v é *atingível* se existe um caminho alternante que termina em v. Note que por essa definição, todos os caminhos alternantes começam em vértices livres de V.

É bem conhecido que um MIS pode ser derivado de um emparelhamento máximo usando técnicas de caminhos alternantes desenvolvidas por Kuhn [Kuh55]. Se M é um emparelhamento máximo de G, sejam $V_A \subseteq V$ e $W_A \subseteq W$ os conjuntos de vértices atingíveis em G. O conjunto $I = V_A \cup (W \setminus W_A)$ é um conjunto independente máximo de G. Desta forma o problema é reduzido a encontrar todos os vértices atingíveis de G. Por simplicidade, assumiremos que o grafo dado não tem vértices isolados. Claramente isso não é problema quando se trata de determinar um *MIS* de um grafo, pois todos os vértices isolados sempre fazem parte de qualquer MIS do grafo.

Antes da descrição de nosso algoritmo paralelo, a Seção 5.4.2 descreve uma versão seqüencial do algoritmo, cuja complexidade é $O(n)$. Este algoritmo, além de ser mais rápido que os previamente conhecidos apresenta características mais relevantes para a paralelização. Na Seção 5.4.3, descrevemos nosso algoritmo BSP/CGM para o problema, com $O(1)$ rodadas de comunicação e computação local $O(n/p)$, onde n é o número de vértices de V e p é o número de processadores. O algoritmo paralelo é baseado fortemente no algoritmo desenvolvido na Seção 5.4.2.

5.4.2 Algoritmo Seqüencial

Seja $G = (V, W, E)$ um grafo bipartido convexo com $V = [1, n]$ e $W = [1, m]$. Abusando da notação V denotará também um vetor onde o elemento v do vetor, $1 \leq v \leq n$, será associado à quádrupla $(v, begin(v), end(v), \mathsf{M}(v))$ que será a representação de G na forma compacta juntamente com um emparelhamento guloso M. A função M que representa o emparelhamento guloso de G é tal como definida na Seção 5.3.

O algoritmo descrito abaixo inicialmente rotula cada $v \in V$ com rótulo $\mathsf{M}(v)$ se v estiver emparelhado, ou com rótulo $end(v)$ se v for vértice livre. Com isso, se o emparelhamento M é máximo e dois vértices têm o mesmo rótulo, pelo menos um deles é

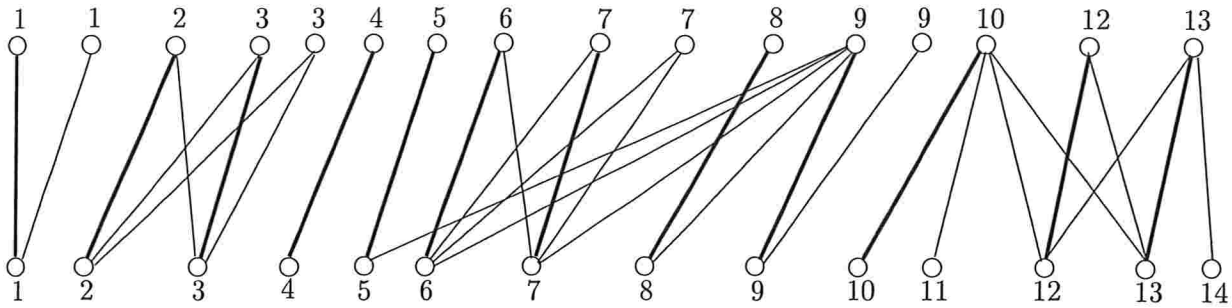


Figura 5.4: Ilustração de um emparelhamento guloso e uma rotulação associada a ele

livre. A Figura 5.4 fornece um emparelhamento guloso em um grafo bipartido convexo e uma rotulação associada ao emparelhamento. O algoritmo ordena o vetor V de acordo com os rótulos dados. Empates são quebrados de forma que vértices emparelhados venha primeiro. Em seguida o algoritmo chama o *Procedimento MIS* com entrada $V[1:n]$. Por fim, o algoritmo retorna os conjuntos V_A e W_A representando os vértices atingíveis em V e W, respectivamente.

Algoritmo MIS

Entrada: Um vetor $V[1:n]$ representando um grafo bipartido convexo $G = (V, W, E)$, onde a cada $V[v]$ está associada a quádrupla $(v, begin(v), end(v), \mathsf{M}(v))$.

Saída: Os conjuntos V_A e W_A representando os vértices atingíveis em V e W, respectivamente.

1: Crie um novo campo denominado *rótulo* no vetor V.
2: **para** $v := 1$ **até** n **faça**
3: **se** $\mathsf{M}(v) > 0$ **então** $rótulo(v) := \mathsf{M}(v)$
4: **senão** $rótulo(v) := end(v)$
5: Ordene crescentemente o vetor V pelo valor de $rótulo(v)$. Empates são quebrados de forma que o vértice emparelhado seja menor que os demais.
6: Chame o *Procedimento MIS* com entrada $V = [1, n]$ obtendo os conjuntos V_A e W_A.
7: Devolva os conjuntos V_A e W_A

O *Procedimento MIS* dado abaixo é quem, de fato, determina o conjunto de vértices atingíveis do grafo. Neste procedimento, o conjunto de vértices atingíveis por caminhos alternantes a partir de vértices livres de V será dado por um conjunto de intervalos. Para cada k, os intervalos $[begin\_V(k), end\_V(k)] \subseteq [1, n]$ e $[begin\_W(k), end\_W(k)] \subseteq [1, m]$ correspondem a vértices atingíveis em V e em W. Desta forma, V_A e W_A são os conjuntos obtidos pelas uniões dos intervalos $[begin\_V(k), end\_V(k)]$ e $[begin\_W(k), end\_W(k)]$, respectivamente. Isto é, $V_A = \cup_k [begin\_V(k), end\_V(k)]$ e $W_A = \cup_k [begin\_W(k), end\_W(k)]$. No que segue, consideraremos que os vértices de V estão em uma ordem lexicográfica $<$

de forma que para quaisquer $v, v' \in V$, $v < v'$ se, e somente se, a posição de v' após a ordenação é posterior a posição de v.

Como notado no Fato 5.13, apresentado mais adiante, cada intervalo k de vértices atingíveis de W é da forma $[a, end(v)]$, onde v é um vértices livre de V. O fato chave explorado por nosso algoritmo é que considerando a ordenação de acordo com os rótulos, o mesmo vale para o conjunto V. Isto é, se V está ordenado pelos rótulos, os vértices de V atingíveis por caminhos alternantes são consecutivos formando um subintervalo de $[1, n]$. Para computar estes intervalos, o conjunto V é examinado, começando da posição final b. Quando um vértice livre v é encontrado, os valores de $begin\_V(k)$ e de $begin\_W(k)$ são inicializados com v e $begin(v)$, respectivamente. Os vértices de V são examinados em ordem decrescente de rótulo, até que seja encontrado o maior $v' \neq v$, $a \leq v' \leq v$ para o qual: (1) $rótulo(v') = begin\_W(k)$; (2) $v' - 1 < a$ ou $rótulo(v'\text{-}1) < begin\_W(k)$; (3) para todo v'', $begin\_V(k) \leq v'' \leq end\_V(k)$, $begin(v'') \geq begin\_W(k)$. Para satisfazer essas condições, o algoritmo vai alterando os valores de $begin\_V(k)$ e de $begin\_W(k)$ se necessário.

Procedimento MIS

Entrada: Um vetor $V[a : b]$ representando um grafo bipartido convexo $G = (V, W, E)$, onde a cada $v \in V$ está associado a quíntupla $(v, begin(v), end(v), M(v), rótulo(v))$. O vetor $V[a : b]$ está ordenado pelos rótulos.

Saída: Os conjuntos V_A e W_A representando os vértices atingíveis de V e W, respectivamente.

1: $v := b$, $k := 0$
2: **enquanto** $v \geq a$ **faça**
3: se $\mathsf{M}(v) = 0$ **então** {encontrado vértice livre}
4: $end\_V(k) := v$
5: $begin\_W(k) := begin(v)$
6: $end\_W(k) := rótulo(v)$
7: **repita**
8: $begin\_V(k) := v$ {Vértice v é inserido em $[begin\_V(k), end\_V(k)]$}
9: $begin\_W(k) := \min\{begin(v), begin\_W(k)\}$
10: **Invariante 1** Cada vértice em $[begin\_V(k), end\_V(k)] \subseteq V$ é atingível
11: **Invariante 2** Cada vértice em $[begin\_W(k), end\_W(k)] \subseteq W$ é atingível
12: $v := v - 1$
13: **até** $v = a$ **ou** $rótulo(v) < begin\_W(k)$
14: $k := k + 1$ { Acrescenta o intervalo $[begin\_V(k), end\_V(k)]$ ao conjunto V_A e o intervalo $[begin\_W(k), end\_W(k)]$ ao conjunto W_A}
15: **Invariante 3** Cada vértice em $[begin\_V(k), end\_V(k)]$ tem seu rótulo no intervalo $[begin\_W(k), end\_W(k)]$

16: **senão**
17: $v := v - 1$
18: Retorne os conjuntos V_A e W_A

Como observado anteriormente o conjunto independente máximo é dado por $I = V_A \cup (W \setminus W_A)$. As proposições a seguir tratam de mostrar a correção do algoritmo. O fato abaixo foi observado em [CDP96], e usa o fato do emparelhamento ser guloso.

Fato 5.13 Se $v \in V$ é vértice livre em M, então os vértices de W atingíveis por caminhos alternantes a partir de v são consecutivos, formando um subintervalo de $[1, end(v)]$.

□

Lema 5.14 As Invariantes 1 e 2 do Procedimento MIS são corretas.

Prova. Seja v um valor para o qual a condição do **if** da Linha 3 é verdadeira.

Vamos mostrar, por indução no número de vezes que o comando **repita** é executado, que as invariantes são verdadeiras.

Na primeira execução do comando **repita**, temos que $[begin\_V(k), end\_V(k)] = [v, v]$ e $[begin\_W(k), end\_W(k)] = [begin(v), end(v)]$. Como i é livre e portanto atingível, pela definição dos caminhos alternantes, também são atingíveis os vértices no intervalo $[begin(v), end(v)]$.

Considere agora uma iteração qualquer do comando **repita**, e suponha que na iteração anterior as invariantes eram verdadeiras. Considere o valor de v e $begin\_W(k)$ no início dessa nova iteração. Por indução sabemos que os vértices em $[v + 1, end\_V(k)]$ são atingíveis. Note que percorremos V em ordem decrescente. Para inserir o vértice v no intervalo $[begin\_V(k), end\_V(k)]$, o valor de $begin\_V(k)$ é alterado. Note que se $rótulo(v) \geq begin\_W(k)$ então $(v, rótulo(v)) \in M$ ou v é livre. Em ambos os casos, temos que v também é atingível. Como estamos numa nova iteração, a condição de parada do **repita** é falsa e vale que $rótulo(v) \geq begin\_W(k)$. Portanto, $rótulo(v) \in [begin\_W(k), end\_W(k)]$. Como, por indução, cada vértice em $[begin\_W(k), end\_W(k)]$ é atingível, o vértice v também é atingível e segue que cada vértice em $[v = begin\_V(k), end\_V(k)]$ é atingível. Uma vez que v é atingível, também são atingíveis os vértices em $[begin(v), end(v)]$. Além disso, como sabemos que $rótulo(v) \in [begin\_W(k), end\_W(k)]$, cada vértice de W em $[begin\_W(k) = \min\{begin(v), begin\_W(k)\}, end\_W(k)]$ é atingível.

□

Lema 5.15 A Invariante 3 do Procedimento MIS está correta.

Prova. Quando a condição do comando **se** da Linha 3 é verdadeira, o comando **repita** é executado pela primeira vez com o valor inicial de $end\_W(k)$ igual a $rótulo(end\_V(k))$. O valor de $end\_W(k)$ não é alterado na execução do **repita**. Na parada do comando **repita**, temos que $rótulo(begin\_V(k)) \geq begin\_W(k)$.

Durante a execução do **repita** são considerados valores decrescentes de v. Como o vetor V está ordenado por rótulos, as observações acima implicam que $begin\_W(k) \leq rótulo(begin\_V(k)) \leq rótulo(end\_V(k)) = end\_W(k)$ no final do **repita**, o que prova o lema.

□

No que segue, I é o conjunto $I = V_A \cup W \setminus W_A$.

Lema 5.16 *Sejam $V_L \subset V$ e $W_L \subset W$ os conjuntos de vértices livres de G em M. Então, $|I| \geq |M| + |V_L| + |W_L|$.*

Prova. Note que $V_L \subset V_A$, pois vértices livres são considerados e inseridos em V_A ou na Linha 3 ou durante a execução do comando **repita** do algoritmo.

Também é verdade que $W_L \subset W \setminus W_A$, pois, pela Invariante 2, cada vértice em W_A é atingível por caminhos alternantes e portanto não pode ser livre: um caminho alternante com término em vértice livre indica a existência de um emparelhamento de cardinalidade maior do que a cardinalidade do emparelhamento máximo M.

Vamos argumentar agora que cada vértice de M tem pelo menos uma ponta em I. Suponha que $(v, w) \in M$ e $w \notin I$. Então, pela definição de I, $w \in W_A$ e, como conseqüência, existe k tal que $w \in [begin\_W(k), end\_W(k)]$. Pelo Lema 5.15, o vértice v, cujo rótulo é w, está no intervalo $[begin\_V(k), end\_V(k)]$, e, portanto, está em $V_A \subseteq I$.

Logo, o lema é verdadeiro pois os conjunto de vértices emparelhados, V_L e W_L são dois a dois disjuntos.

□

Lema 5.17 O conjunto I é independente.

Prova. Vamos mostrar que, para todo k, os vértices de W que são adjacentes a vértices em $[begin\_V(k), end\_V(k)]$ estão todos no intervalo $[begin\_W(k), end\_W(k)]$. Como $I = V_A \cup W \setminus W_A$, a afirmação fica provada.

Suponha que exista k para o qual exista $v \in [begin\_V(k), end\_V(k)]$ com algum vizinho fora de $[begin\_W(k), end\_W(k)]$. Então, ou $begin(v) < begin\_W(k)$ ou $end(v) > end\_W(k)$.

O primeiro caso não pode acontecer pois, na Linha 8 quando v foi inserido no intervalo $[begin\_V(k), end\_V(k)]$, o comando seguinte atribui como $begin\_W(k)$ o mínimo entre $begin(v)$ e $begin\_W(k)$. Como o valor de $begin\_W(k)$ nunca aumenta durante a execução do **repita**, quando termina uma de suas execuções, $begin(v)$ continua menor ou igual a $begin\_W(k)$.

Vamos examinar o segundo caso, ou seja $end(v) > end\_W(k)$. Se isso ocorresse, como v é atingível, existiriam vértices de W, atingíveis a partir de um vértice livre q, com valores maiores do que $rótulo(q) = end\_W(k)$. Mas, pelo Fato 5.13, isso também não pode ocorrer.

□

Lema 5.18 *Seja $G = (V, W, E)$ um grafo bipartido, I um conjunto independente de G e M um emparelhamento de G. Então $|I| \leq |V_L| + |W_L| + |M|$, onde $V_L \subseteq V$ e $W_L \subseteq W$ são os conjuntos de vértices livres de G em M.*

Prova. Observe inicialmente que $|V|+|W| = |V_L|+|W_L|+2|M|$, pois cada vértice de G ou é livre ou é extremidade de uma aresta de M. Disso decorre que, se $|I| > |V_L|+|W_L|+|M|$, então I contém mais do que $|M|$ vértices emparelhados. Portanto, existe pelo menos uma aresta de M com ambas as extremidades em I, o que contraria a definição de conjunto independente.

□

Finalmente, agora temos a prova do teorema que completa a correção do algoritmo.

Teorema 5.19 *Seja $G = (V, W, E)$ um grafo bipartido convexo sem vértices isolados, Então, o conjunto $I = V_A \cup W \setminus W_A$ é um conjunto independente máximo, onde V_A e W_A são os conjuntos definidos pelo Algoritmo MIS.*

Prova. Decorre diretamente dos Lemas 5.17, 5.16 e 5.18 que I é um conjunto independente máximo.

□

Para finalizar esta seção, descrevemos a complexidade de tempo do *Algoritmo MIS*. A inicialização do campo de rótulos do vetor V é claramente executada em tempo $O(n)$. A ordenação do vetor V pode ser executada em tempo $O(n)$ usando Radixsort [CLR90, Cap. 9], caso $m = O(n^c)$, para alguma constante c. Caso contrário, podemos usar um algoritmo de ordenação padrão de tempo $O(n \lg n)$. Para verificar que o comando **enquanto** é também executado em tempo $O(n)$ basta observar que o valor de v, inicializado com n,

sempre decresce de pelo menos uma unidade a cada iteração do **enquanto** ou a cada iteração do comando **repita**. Portanto, o algoritmo roda em tempo $O(n)+T_s(n,m)$, onde $T_s(n,m)$ é o tempo de ordenação de n inteiros no intervalo $[1,m]$. Um outro algoritmo seqüencial conhecido para este problema é devido a Lipski e Preparata [LP81]. Este algoritmo roda em tempo $O(n+m)$. Note que nosso algoritmo melhora a complexidade de pior caso para o problema. Quando $m = O(n^c)$ para algum $c > 1$, o Algoritmo de Lipski e Preparata roda em tempo $O(n^c)$, enquanto o nosso é linear em n. Caso contrário, digamos se $m = \Omega(n^c)$ para alguma constante $c > 1$, o Algoritmo de Lipski e Preparata roda em tempo $\Omega(n^c)$, enquanto o nosso roda em tempo $O(n \lg n)$.

5.4.3 Algoritmo BSP/CGM

O algoritmo BSP/CGM opera de forma similar ao algoritmo seqüencial. A entrada do algoritmo é o conjunto $V = [1,n]$ e a quádrupla $(v, begin(v), end(v), \mathsf{M}(v))$ associada a cada $v \in V$. Este conjunto é igualmente distribuído entre os processadores. Da mesma forma que no algoritmo seqüencial, os vértices de V são rotulados, ordenados de acordo com os rótulos e redistribuídos aos processadores. A rotulação e ordenação são feitas facilmente no modelo BSP/CGM.

Vamos descrever o funcionamento do algoritmo supondo a existência de p processadores, P_0, P_1, ..., P_{p-1}. Seja $V[a : b]$ a parte do vetor recebido pelo processador P_k, $0 \leq k < p$, após a ordenação dos vértices pelos rótulos. Vamos chamar os vértices em $V[a : b] \cup [rótulo(a), rótulo(b)]$ de *vértices atribuídos* ao processador P_k. No final do processamento, P_k deverá determinar quais vértices em $[a,b]$ são atingíveis. Também deverá determinar os vértices atingíveis em $[rótulo(a), rótulo(b)] \subseteq W$. Embora seja possível que algum vértice em W não seja atribuídos a nenhum processador, o fato é que todo vértice atingível de W é atribuído a algum processador.

No algoritmo seqüencial, o processamento dos vértices de V é feito em ordem decrescente de rótulos. O problema a ser resolvido no algoritmo BSP/CGM é a determinação, pelo processador P_k, da existência de caminhos alternantes começando em vértices atribuídos aos processadores P_{k+1}, P_{k+2}, ..., P_{p-1}. E, em caso positivo, saber quais os vértices atribuídos a P_k que são atingíveis por esses caminhos. Neste caso, basta a informação do menor vértice de W atingível por tais caminhos. Vamos chamar tal vértice de *min_ating*. A partir daí, o processador P_k pode prosseguir com o processamento, usando o valor de *min_ating* no papel de $begin\_W(k)$ do algoritmo seqüencial. O problema é como determinar o valor de *min_ating* sem depender do encadeamento de resultados obtidos pelo processadores P_{p-1}, ..., P_{k+2}, P_{k+1}.

Se ao processador P_{k+1} é atribuído um vértice livre a partir do qual vértices atribuídos a P_k são atingidos, basta o processador P_{k+1} determinar e informar ao processador P_k o número do menor vértice de W atingível por esses caminhos. Esse vértice, que chamaremos

de *min_rel*, é um candidato a *min_ating*. A determinação desse vértice de W, que pode ser feita em paralelo, deverá ser informada aos demais processadores. Cada processador montará um vetor $Min\_rel[k+1:p-1]$ contendo o número desses vértices.

Embora essa informação seja necessária, ela não é suficiente. Pode ser que existam caminhos alternantes atingindo vértices atribuídos a P_k cuja origem não esteja nos vértices atribuídos a P_{k+1}. Neste caso, todos os vértices atribuídos a P_{k+1} seriam também atingidos. Para exemplificar, vamos supor que nenhum vértice livre de V atribuído a P_{k+1} atinge vértices atribuído a P_k, mas existem vértices livres atribuídos a P_{k+2} cujos caminhos alternantes atingem vértices atribuídos a P_k. Neste caso, o valor de *min_ating* será o menor dentre $Min\_rel[k+2]$ e o mínimo dos $begin(v)$, para todo $v \in V$ atribuído a P_{k+1}. Esse último mínimo será chamado de *min_abs*. Sua determinação também pode ser feita em paralelo e informada aos demais processadores. Cada processador montará um vetor $Min\_abs[k+1:p-1]$ contendo o número desses vértices.

Porém, temos ainda um outro complicador. Pode ser que os caminhos alternantes com início em vértices atribuídos ao processador P_{k+2} terminem todos em vértices atribuídos ao processador P_{k+1}. Dessa forma, nenhum vértice atribuído a P_k seria vértice atingível a partir de vértices livres atribuídos a outros processadores.

Para resolver esse problema, o processador P_{k+1} pode informar se existem vértices atribuídos a ele que são candidatos a término de intervalos de vértices atingíveis. A procura de tais vértices se justifica pelo Lema 5.13, que garante que vértices atingíveis a partir de um mesmo vértice formam um subintervalo de W. Esse vértice será procurado em W e será chamado de *batente*. O processador P_{k+1} deverá procurar por um vértice tal que todo vértice v atribuído a P_{k+1} com rótulo maior ou igual a *batente* tem seus vizinhos em W maiores do que *batente*. Ou seja, $begin(v) \geq batente$, para todo v tal que $rótulo(v) \geq batente$. Se caminhos alternantes com origem em P_{k+2} atingem somente vértices atribuídos a P_{k+1} maiores ou iguais a *batente*, então nenhum vértice de P_k seria atingido por esses caminhos alternantes. Portanto, os valores desses batentes, que serão determinados em paralelo, também serão informados aos demais processadores. Cada processador montará um vetor $Batente[k+1:p-1]$ contendo o número desses vértices. Caso existam mais de um candidato a batente num mesmo processador, basta informar o menor deles.

Resumindo, cada processador P_k, depois de receber o vetor $V[a:b]$ ordenado pelos rótulos, deverá determinar localmente e informar aos outros processadores:

1. o valor de *min_rel*, o número do menor vértice de W atingível por caminhos alternantes com origem em vértices atribuídos a P_k;

2. o valor de *min_abs*, o número do menor vértice de W com vizinhos em algum vértice atribuído a P_k;

3. o valor de *batente*, o número do menor vértice de W atribuído a P_k que é candidato

a término de intervalo de vértices atingíveis.

Esses valores são encontrados pelo *Procedimento Resumo* dado a seguir.

Procedimento Resumo

Entrada: Um vetor $V[a : b]$ ordenado pelos rótulos representando um grafo convexo $G = (V, W, E)$.

Saída : Os valores de *batente*, *min_rel* e *min_abs*.

1: {Cálculo de *batente* e de *min_abs*}
2: *batente* :=*rótulo*(b) +1
3: $ind = b + 1$
4: $v := b$
5: **enquanto** $v \geq a$ **faça**
6: *candidato* := *begin*(v)
7: **enquanto** $v \geq a$ **e** *rótulo*(v) $\geq$ *candidato* **faça**
8: *candidato* := min{*candidato*, *begin*(v)}
9: $v := v - 1$
10: **se** *candidato* = *begin*($v + 1$) **e** *candidato* $\geq$*rótulo*(a) **e** M($v + 1$) $\neq 0$ **então**
11: *batente* := *candidato*
12: $ind = v + 1$
13: *min_abs* := $\min_{v \in [a,b]}$\{*begin*(v)\}
14: {Cálculo de *min_rel*}
15: $v := ind - 1$
16: *min_rel* := $+\infty$
17: **enquanto** $v \geq a$ **faça**
18: **se** M(v) = 0 **então** {encontrado vértice livre}
19: *candidato* := *begin*(v)
20: **repita**
21: *candidato* := min{*candidato*, *begin*(v)}
22: $v := v - 1$
23: **até** $v < a$ **ou** *rótulo*(v) $<$ *candidato*
24: **se** *candidato* $<$*rótulo*(a) **ou** (M(a) = 0 **e** *candidato* =*rótulo*(a)) **então**
25: *min_rel* := *candidato*
26: **senão**
27: $v := v - 1$
28: Retorne *batente*, *mim_rel* e *min_abs*

No nosso exemplo, apenas consideramos caminhos alternantes com início em P_{k+1} e P_{k+2}. Porém, com as informações acima, cada processador P_k é capaz de determinar, para qualquer $q > k$, se existem caminhos alternantes com início em vértices atribuídos a P_q que atingem vértices atribuídos a P_k. Para tanto, o processador P_k inicialmente constrói os vetores $Min\_Rel[k + 1 : p - 1]$, $Min\_Abs[k + 1 : p - 1]$ e $Batente[k + 1 : p - 1]$.

Os vetores $Min\_Rel[k+1:p-1]$ e $Batente[k+1:p-1]$ são examinados do final para o início de forma a identificar algum vértice livre de V, que foi atribuído a algum processador P_q, com $q > k$, o qual é início de caminhos alternantes que atingem vértices atribuídos a algum $P_l < P_q$. A partir deste ponto, P_k percorre $Min\_Abs[k:l]$ para identificar o processador que possui o término deste intervalo através do vetor $Batente[k+1:l]$. Este processo é repetido até encontrar um vértice livre atribuído a algum $P_q > P_k$ que atinge algum vértice atribuído a P_k.

Qualquer processador poderá determinar para qual processador foi atribuído o vértice de W que é término do intervalo. Mas somente o processador responsável pelo vértice saberá exatamente o final do intervalo, uma vez que havendo mais de um candidato a *batente* num mesmo processador, somente o menor deles é informado. Para essa determinação, e para a determinação dos intervalos de vértices atingíveis por caminhos alternantes com início e término num mesmo processador, cada processador rodará o *Algoritmo MIS* seqüencial. O conjunto independente máximo obtido é dado por $I = \cup_k V_A^k \cup (W \backslash W_A^k)$.

Algoritmo BSP/CGM MIS

Entrada: Um vetor $V[1:n]$ representando um grafo convexo $G = (V, W, E)$ sem vértices isolados. O processador P_k, $0 \leq k < p$, recebe o vetor $V[a:b]$, onde $a = k\lceil n/p \rceil + 1$ e $b = \min\{n, (k+1)\lceil n/p \rceil\}$.

Saída: Cada P_k determina os conjuntos V_A^k e W_A^k de vértices atribuído a P_k.

Passo 1 Cada P_k, $0 \leq k < p$ rotula cada $v \in [a, b]$ com $rótulo(v) := \mathsf{M}(v)$, se $\mathsf{M}(v) \neq 0$ e com $rótulo(v) := end(v)$, caso contrário.

Passo 2 Em paralelo, ordene o vetor V pelo valor do campo $rótulo(v)$. Empates são quebrados de forma que o vértice emparelhado apareça antes que os demais.

Passo 3 Cada P_k calcula *batente*, *min_rel* e *min_abs* usando o *Procedimento Resumo* e distribui *batente*, *min_rel* e *min_abs* para os processadores com identificadores menores do que k.

Passo 4 Cada P_k constrói os vetores $Min\_rel[k:p-1]$, $Min\_abs[k:p-1]$ e $Batente[k:p-1]$ com as informações recebidas no passo anterior.

Passo 5 Cada P_k calcula $min\_ating \in W$, o menor vértice de W atingível por vértices de V distribuídos aos processadores com identificadores maiores do que k, de acordo com o código abaixo. Caso não exista tal vértice, então $min\_ating > rótulo(b)$.

$i := p - 1$
$min\_ating := +\infty$

enquanto $i > k$ **faça**
  enquanto $i > k$ e $Min\_rel[i] = +\infty$ **faça**
    $i := i - 1$
  se $i > k$ **então**
    $min\_ating := Min\_rel[i]$
    $i := i - 1$
  enquanto $i > k$ e $min\_ating < Stopper[i]$ **faça**
    $min\_ating := \min\{min\_ating, Min\_Abs[i]\}$
    $i := i - 1$

Passo 6 Cada P_k calcula para cada $v \in [a, b]$ $begin(v) := \max\{rótulo(a), begin(v)\}$.

Passo 7 Cada P_k verifica se $min\_ating \leq rótulo(b)$. Caso positivo: (1) adiciona o vértice $b+1$ em V com $begin(v+1) := \max\{min\_ating, rótulo(a)\}$, $end(b+1) := end(b)$, $\mathsf{M}(b+1) = 0$ e $rótulo(b+1) := rótulo(b)$; (2) aplica o algoritmo seqüencial com entrada $V[a : b+1]$, obtendo os conjuntos V_A^k e W_A^k e remove o vértice $V[b+1]$ de V_A^k. Caso contrário, aplica o algoritmo seqüencial com entrada $V[a : b]$, obtendo os conjuntos V_A^k e W_A^k

Passo 8 Cada P_k retorna os conjuntos V_A^k e W_A^k.

Agora vamos analisar a complexidade de tempo do algoritmo BSP/CGM.

A criação dos rótulos gasta tempo $O(n/p)$. A ordenação pode ser feita em tempo de computação $O(n/p)$ com número de rodadas constante usando o algoritmo BSP/CGM para ordenar n inteiros no intervalo $[1, m]$ de [CD99] descrito na Seção 3.2, caso $m = O(n^c)$, para alguma constante c. Caso não haja limite para m, a ordenação pode ser feita no modelo BSP/CGM em tempo de computação $O(n \lg n/p)$ com número de rodadas constante usando o Algoritmo de Goodrich [Goo96]. É fácil ver que o *Procedimento Resumo* roda em tempo $O(n/p)$. Em uma rodada de comunicação o Passo 3 distribui $O(p)$ dados. O laço do Passo 5 gasta tempo $O(p)$. As chamadas do algoritmo seqüencial na Passo 7 gastam tempo $O(n/p)$. Portanto, o algoritmo BSP/CGM efetua um número de rodadas de comunicação constante e o tempo de computação total do algoritmo é $O(n/p) + T_s(n, m, p)$, onde $T_s(n, m, p)$ é o tempo da ordenação de n inteiros no intervalo $[1, m]$ no modelo BSP/CGM. No modelo BSP o algoritmo realiza $O(1)$ superpassos com custo de computação local igual ao CGM e custo de comunicação $O(gn/p)$.

5.5 Circuitos Hamiltonianos

Um *circuito hamiltoniano* em um grafo $G = (V, E)$ de n vértices é um circuito simples $(v_1, \ldots, v_n, v_1)$, contendo cada vértice $v_i \in V$. Isto é, $\{v_i, v_{i+1}\} \in E$, para $1 \leq i \leq n - 1$

e $\{v_1, v_n\} \in E$. Um grafo que contenha um circuito hamiltoniano é chamado *grafo hamiltoniano*. O problema de decidir se um grafo é hamiltoniano é um dos tópicos mais conhecidos em teoria dos grafos. O problema é NP-completo [GJ79]. O problema permanece NP-difícil mesmo para grafos bipartidos [Kri75] e grafos bipartidos cordais [Mul96]. Uma hierarquia de classes de grafos freqüentemente considerada é: grafos bipartidos de permutação $\subset$ grafos bipartidos duplamente convexos $\subset$ grafos bipartidos convexos $\subset$ grafos bipartidos convexos circulares $\subset$ grafos bipartidos cordais [Gol80]. Neste trabalho tratamos do problema de encontrar um circuito hamiltoniano em grafos bipartidos convexos.

Apresentamos um algoritmo seqüencial para encontrar um circuito hamiltoniano em grafos bipartidos convexos que roda em tempo $O(n)$. Müller menciona em [Mul96] um algoritmo $O(n^2)$ para o mesmo problema. Para uma classe mais ampla de grafos, os grafos bipartidos convexos *circulares*, Liang e Blum [LB95] apresentam um algoritmo seqüencial $O(n)$ para encontrar circuitos hamiltonianos.

O nosso algoritmo seqüencial tem uma fácil paralelização a qual é apresentada na Seção 5.5.2. Esta versão paralela se baseia em algoritmos paralelos eficientes para o problema de emparelhamento máximo.

5.5.1 Algoritmo Seqüencial

Embora todo circuito hamiltoniano num grafo bipartido seja a união de dois emparelhamentos perfeitos disjuntos, nem toda união de dois emparelhamentos perfeitos disjuntos é um circuito hamiltoniano. Esse fato é ilustrado pelo grafo G_A da Figura 5.5. A união dos dois emparelhamentos perfeitos $\{(v_1, w_1), (v_2, w_2), (v_3, w_3), (v_4, w_4)\}$ e $\{(v_1, w_2), (v_2, w_1), (v_3, w_4), (v_4, w_3)\}$ é a união de dois circuitos disjuntos. Também, pode ser o caso de que a escolha de um emparelhamento perfeito deixe o grafo sem outro emparelhamento perfeito disjunto do primeiro. Esse fato é ilustrado pelo grafo G_B da Figura 5.5. O emparelhamento $\{(v_1, w_1), (v_2, w_2), (v_3, w_3)\}$ deixa o grafo sem outro emparelhamento perfeito disjunto.

Porém, em grafos bipartidos convexos hamiltonianos podemos encontrar um circuito hamiltoniano que essencialmente consiste da união disjunta de dois emparelhamentos. O algoritmo para tal é descrito a seguir.

Como antes, para M um emparelhamento num grafo bipartido convexo $G = (V, W, E)$, para cada $x \in V \cup W$, $\mathsf{M}(x) = y > 0$ se $\{x, y\} \in M$, e $M(x) = 0$ se x é um vértice livre em M. Denotaremos por $N(x)$ os vértices adjacentes a x e por $N(\{x_i, \ldots, x_j\})$ ou simplesmente $N(x_i, \ldots, x_j)$ o conjunto $\cup_{k=i}^{j} N(v_k)$.

No algoritmo consideramos o grafo representado na sua forma compacta. Como identificamos os vértices de $W := \{w_1, w_2, \ldots, w_n\}$ com inteiros, vamos abusar da notação e

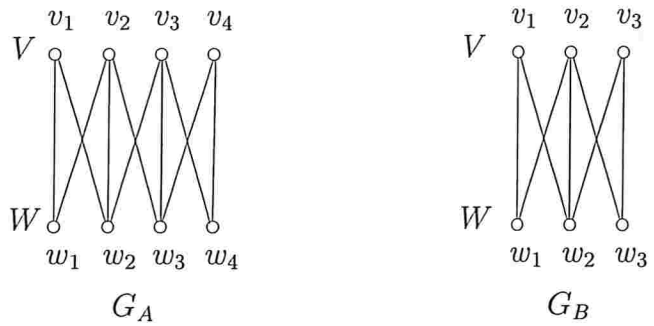


Figura 5.5: Dois grafos bipartidos convexos hamiltonianos.

usar, quando conveniente, expressões do tipo $w_i = w_{i-1} + 1 = w_{i+1} - 1$ onde $w_i \in W$.

Algoritmo Circuito Hamiltoniano

Entrada: Um grafo $G = (V, W, E)$ representado na forma compacta.

Saída: Um circuito hamiltoniano caso G tenha um.

Passo 1 Escolha como v_1 qualquer vértice de V adjacente a w_1.

Passo 2 $G_1 := G - v_1 - w_n$.

Passo 3 Encontre um emparelhamento guloso M em G_1.

Passo 4 Caso $|M| < n - 1$, devolva que G não é hamiltoniano.

Passo 5 $G_2 := G - w_1$.

Passo 6 Para $v_i \in V \backslash \{v_1\}$ faça $begin(v_i) = \mathsf{M}(v_i) + 1$.

Passo 7 Encontre um emparelhamento guloso M' em G_2.

Passo 8 Caso $|M'| < n-1$ ou $w_n \notin N(v_k)$, onde v_k é o vértice de V livre em M', devolva que G não é hamiltoniano. Caso contrário, devolva $M \cup M' \cup (v_1, w_1) \cup (v_k, w_n)$.

As Figuras 5.6, 5.7, 5.8 e 5.9 descrevem um exemplo da aplicação do algoritmo começando pela a escolha do vértice v_1 no Passo 1, o emparelhamento guloso M encontrado em G_1 no Passo 3, o emparelhamento guloso M' encontrado em G_2 no Passo 7 e o circuito hamiltoniano resultante.

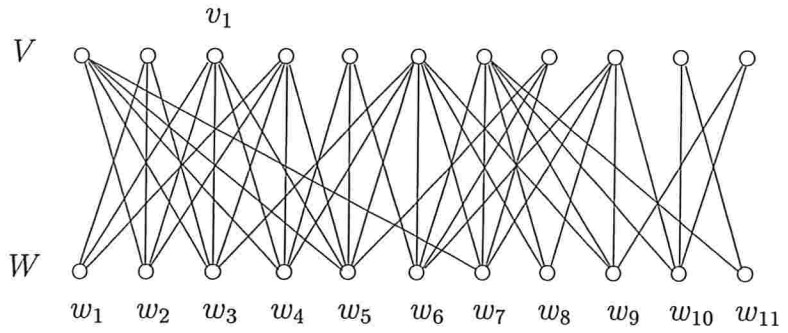


Figura 5.6: Grafo G com $n = 11$ mostrando o vértice escolhido como v_1.

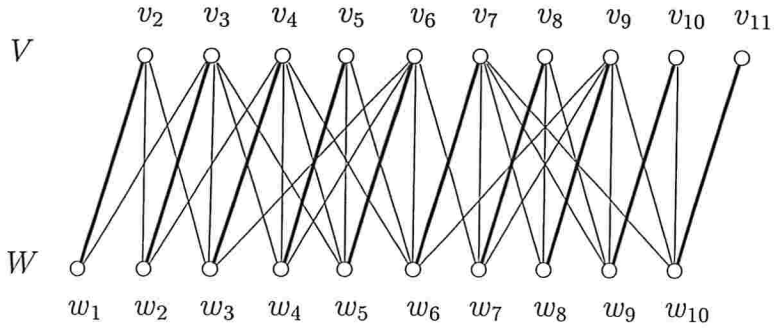


Figura 5.7: Grafo G_1 mostrando as arestas de M em negrito. Os vértices de V já estão ordenados por M.

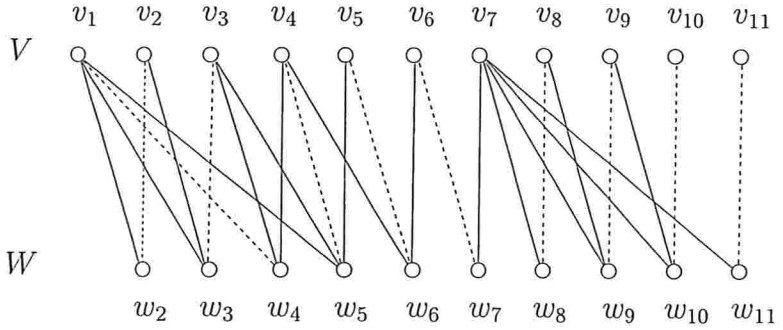


Figura 5.8: Grafo G_2 mostrando as arestas de M' tracejadas.

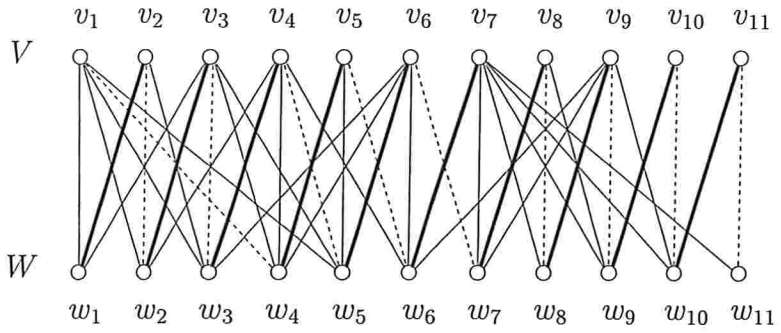


Figura 5.9: Grafo G mostrando o circuito hamiltoniano composto pelas arestas de M em negrito, de M' tracejadas e arestas (v_1, w_1) e (v_7, w_{11}).

O circuito hamiltoniano encontrado vai ser formado por uma das seqüências abaixo:

$$(w_1, v_1, \mathsf{M}'(v_1), \mathsf{M}(\mathsf{M}'(v_1)), \mathsf{M}'(\mathsf{M}(\mathsf{M}'(v_1))), \ldots, v_k, w_n,$$
$$\mathsf{M}'(w_n), \mathsf{M}(\mathsf{M}'(w_n)), \mathsf{M}'(\mathsf{M}(\mathsf{M}'(w_n))), \ldots, \mathsf{M}(w_1), w_1)$$

ou

$$(w_1, v_1, \mathsf{M}'(v_1), \mathsf{M}(\mathsf{M}'(v_1)), \mathsf{M}'(\mathsf{M}(\mathsf{M}'(v_1))), \ldots, w_n, v_k,$$
$$\mathsf{M}(v_k), \mathsf{M}'(\mathsf{M}(v_k)), \mathsf{M}(\mathsf{M}'(\mathsf{M}(v_k))), \ldots, \mathsf{M}(w_1), w_1).$$

Os Lemas que seguem ajudarão a demonstrar a correção do algoritmo.

Lema 5.20 *Se $G = (V, W, E)$ é bipartido hamiltoniano, então para qualquer S subconjunto próprio não-vazio de V temos $|S| < |N(S)|$.*

Prova. Seja C um circuito hamiltoniano em G. Como S é subconjunto próprio não-vazio de V, levando em conta somente as arestas de C, fica definida uma vizinhança de S de cardinalidade maior do que $|S|$.

□

Lema 5.21 *Se G é hamiltoniano, então G_1 tem um emparelhamento perfeito.*

Prova. Seja C um circuito hamiltoniano em G. Sejam P o caminho de v_1 para w_n em C, e P' o caminho de w_n para v_1 em C. Note que, como o grafo é bipartido, tanto P quanto P' têm um número ímpar de arestas. Os caminhos $P - v_1 - w_n$ e $P' - v_1 - w_n$ são completamente disjuntos e suas arestas contêm um emparelhamento perfeito em $G_1 = G - v_1 - w_n$.

□

No que segue consideraremos que os vértices de V estão ordenados de acordo com M, de forma que para cada i, $1 < i \leq n$, $\mathsf{M}(v_i) = w_{i-1}$.

Lema 5.22 *Se G é hamiltoniano, então $\mathsf{M}(v_j) < end(v_j)$ para cada j, $1 < j \leq n$.*

Prova. Suponha, por contradição, que G é hamiltoniano e existe $1 < j \leq n$ tal que $\mathsf{M}(v_j) = w_{j-1} \geq end(v_j)$. Como v_j é emparelhado com w_{j-1}, só pode ser o caso que $\mathsf{M}(v_j) = end(v_j)$. Se $N(v_1, \ldots, v_{j-1}) \subseteq \{w_1, \ldots, w_{j-1}\}$, temos que, pelo Lema 5.20, G não é hamiltoniano. Então, existe $r < j$ tal que $end(v_r) > end(v_j)$. Vamos escolher r o maior possível, de forma que, para cada i, $r < i \leq j$, temos que $end(v_i) \leq end(v_j)$. Da observação anterior, e lembrando que M é guloso, vale que $begin(v_i) > \mathsf{M}(v_r)$ para cada $r < i \leq j$. Portanto, segue que $N(v_{r+1}, \ldots, v_j) = \{\mathsf{M}(v_{r+1}), \ldots, \mathsf{M}(v_j)\}$, e, pelo Lema 5.20 G, não é hamiltoniano. Uma contradição.

□

Lema 5.23 *Se G é hamiltoniano, então cada vértice em $W - \{w_1\}$ é emparelhado por M'.*

Prova. Como $M = \{(v_2, w_1), (v_3, w_2), \ldots, (v_n, w_{n-1})\}$ é um emparelhamento, usando o Lema 5.22 e a convexidade de G, $M'' = \{(v_2, w_2), (v_3, w_3), \ldots, (v_n, w_n)\}$ também é um emparelhamento. Note que M'' é um emparelhamento em G_2 que emparelha cada vértice em $W - \{w_1\}$. Como M' é emparelhamento máximo, M' também emparelhará cada vértice em $W - \{w_1\}$.

□

Note que, caso o grafo G seja hamiltoniano, após o Passo 7 do algoritmo o grafo G possui dois emparelhamentos M e M', ambos de tamanho $n - 1$. Como, em G_2 temos $|V| = n$ e $|W| = n - 1$, então existe exatamente um vértice em V livre em relação a M'.

Lema 5.24 *Seja v_k o vértice livre de V em M'. Se G é hamiltoniano, então v_k é adjacente a w_n.*

Prova. Por contradição, suponha que v_k não é adjacente a w_n. Seja q tal que $end(v_k) = \mathsf{M}(v_q) = w_{q-1}$. Como pelo Lema 5.22, $end(v_k) > \mathsf{M}(v_k)$, temos $k > q$.

Primeiro, vamos mostrar que para cada i, $1 \leq i < q$, temos que $\mathsf{M}'(v_i) \leq \mathsf{M}(v_q)$. Por construção de G_2, os únicos vértices de V que podem ser adjacentes a vértices em $W^* := \{w_2, w_3, \ldots, w_{q-1}\}$ em G_2 são os vértices em $V^* := \{v_1, v_2, \ldots, v_{q-1}\}$. Como v_k é um desses vértices e, pelo Lema 5.23, todos os vértices de $W - \{w_1\}$ estão emparelhados por M', então M' induz uma bijeção entre entre W^* e $V^* - v_k$.

Como V^* é não-vazio podemos usar o Lema 5.20 para argumentar que $|N(V^*)| > |\{w_1, w_2, \ldots, w_{q-1}\}|$ e que, portanto, existe um $r < q$ tal que $end(v_r) > w_{q-1}$. Escolha r com essa propriedade de forma a maximizar $w_j := \mathsf{M}'(v_r)$. Como $end(v_k) < end(v_r)$, v_k é livre em M' e M' é guloso, temos que w_j não é adjacente a v_k em G_2. Pelo observado no parágrafo anterior, podemos concluir que $w_j < w_k$, pois os vizinhos de v_k em G_2 são $w_k, w_{k+1}, \ldots, w_{q-1}$.

Queremos mostrar agora que $j = r$. Suponha que não, que $j > r$. Então, como v_j é adjacente a w_j em G_2 e M' é guloso, podemos concluir que $end(v_j) \geq end(v_r)$, o que contraria a escolha de r, pois $\mathsf{M}'(v_j)$ seria maior que $\mathsf{M}'(v_r)$.

Note que existem $r - 1$ vértices menores do que v_r emparelhados por M'. Portanto, algum deles, digamos v_s, é emparelhado a algum vértices fora de $\{w_2, w_3, \ldots, w_{r-1}\}$. Segue que $\mathsf{M}'(v_s) > \mathsf{M}'(v_r) = w_r$. Como v_s não foi emparelhado por M' com w_r, temos que $end(v_s) \geq end(v_r)$. Novamente, isso contradiz a escolha de r, mostrando que não existe tal r, uma contradição.

□

Dos lemas acima, segue o seguinte teorema.

Teorema 5.25 *O Algoritmo Circuito Hamiltoniano resolve o problema do circuito hamiltoniano para grafos bipartidos convexos.*

Prova. Vamos mostrar que o conjunto de arestas $E_C := M \cup M' \cup (v_1, w_1) \cup (v_k, w_n)$ são exatamente as arestas de um circuito hamiltoniano de G.

Observe inicialmente que, por construção, (v_1, w_1) é uma aresta do grafo. A existência dos emparelhamentos e da aresta (v_k, w_n) estão garantidas pelos lemas acima.

Por inspeção, podemos verificar que E_C induz um subgrafo de G onde cada vértice tem grau par. Portanto, E_C induz uma coleção de circuitos disjuntos em G. Precisamos mostrar que E_C induz exatamente 1 circuito, e portanto, um circuito hamiltoniano.

Seja C um dos circuitos induzidos por E_C. Vamos mostrar que a aresta (v_k, w_n) é usada por C. Como a escoha de C é arbitrária e os circuitos são disjuntos, então E_C tem que induzir somente 1 circuito.

Como C tem pelo menos 4 arestas, C tem que usar pelo menos uma aresta de M. Podemos então considerar que $C = (w_{j_1}, v_{i_1}, w_{j_2}, v_{i_2}, \ldots, w_{j_1})$, onde $\mathsf{M}(w_{j_1}) = v_{i_1}$.

Vamos provar que a seqüência $j_1, i_1, j_2, i_2, \ldots$ é crescente até que a aresta (v_k, w_n) seja usada.

Lembrando que os vértices de V foram ordenados por M de forma que para cada i, $\mathsf{M}(v_i) = w_{i-1}$, temos que $i_1 = j_1 + 1$. Isso prova que $j_1 < i_1$. Continuando, vamos provar

que $i_1 \leq j_2$. Caso $(v_{i_1}, w_{j_2}) = (v_k, w_n)$, nada mais há a provar. Como $i_1 > 1$, também não pode ser o caso que $(v_{i_1}, w_{j_2}) = (v_1, w_1)$. Portanto, vamos considerar que $\mathsf{M}'(v_{i_1}) = w_{j_2}$. Como M' é emparelhamento em G_2, onde para cada i, $begin_{G_2}(v_i) = \mathsf{M}(v_i) + 1$, temos que $\mathsf{M}'(v_{i_1}) = w_{j_2} \geq begin_{G_2}(v_{i_1}) = \mathsf{M}(v_{i_1}) + 1 = w_{i_1-1} + 1 = w_{i_1}$. Segue que $i_1 \leq j_2$.

Aplicando o mesmo raciocício do parágrafo anterior, podemos provar que $j_2 < i_2 \leq j_3$, a menos que a aresta (v_k, w_n) seja usada. Como o circuito termina em w_{j_1}, a seqüência de índices não pode ser sempre crescente e a aresta (v_k, w_n) precisa ser usada por C.

□

Complexidade

A complexidade de tempo do algoritmo é dominada pelos Passos 3 e 7, os quais consistem de encontrar um emparelhamento guloso em um grafo bipartido convexo. Usando o algoritmo de Steiner e Yeomans [SY96], estas operações podem ser realizadas em tempo $O(n)$. Em suma, temos o seguinte teorema.

Teorema 5.26 *O Algoritmo resolve o problema do circuito hamiltoniano em um grafo bipartido convexo em tempo $O(n)$.*

□

5.5.2 Algoritmos Paralelos

O algoritmo seqüencial da seção anterior consiste basicamente de encontrar dois emparelhamentos gulosos em um grafo bipartido convexo. No modelo PRAM, Dekel e Sahni [DS84] desenvolveram um algoritmo EREW PRAM para o problema do emparelhamento máximo que roda em tempo $O(\lg^2 n)$ usando $O(n)$ processadores. Na Seção 5.3.3 descrevemos um algoritmo BSP/CGM com $O(\lg p)$ rodadas de comunicação e $O(T_s(n/p)+n/p \lg p)$ tempo de computação local, onde $T_s(n)$ é a complexidade tempo seqüencial para o problema. Os emparelhamentos máximos encontrados pelos algoritmos acima são gulosos. Desta forma, temos os teoremas abaixo.

Teorema 5.27 *O problema do circuito hamiltoniano em um grafo bipartido convexo é resolvido em tempo $O(\lg^2 n)$, usando n processadores no modelo EREW PRAM.*

□

Teorema 5.28 *O problema do circuito hamiltoniano em um grafo bipartido convexo é resolvido no modelo BSP/CGM com $O(\lg p)$ rodadas de comunicação e $O(T_s(n/p)+n/p \lg p)$ de tempo de computação local, onde $T_s(n)$ é a complexidade de tempo seqüencial para o problema do emparelhamento máximo.*

□

5.6 Caminhos Mínimos em Grafos Bipartidos Biconvexos

Um grafo bipartido $G = (V, W, E)$ é biconvexo se há uma ordenação dos vértices de W tal que cada vértice de V tem seus vizinhos consecutivos em W e o mesmo ocorre em V, ou seja há também uma ordenação em V tal que cada vértice de W tem seus vizinhos consecutivos em V. Nesta seção tratamos do problema de encontrar os caminhos mínimos entre todos os pares de vértices em grafos bipartidos biconvexos usando o modelo BSP/CGM. O algoritmo descrito a seguir é uma adaptação para o modelo BSP/CGM baseado no algoritmo PRAM de Chen [Che99].

Seja $M_{n \times m}$ a matriz de adjacência de um grafo bipartido $G = (V, W, E)$, onde $|V| = n$ e $|W| = m$. Uma matriz-(0,1) satisfaz a propriedade dos uns consecutivos para as linhas (ou colunas) se suas colunas (ou linhas) podem ser permutadas tal que a matriz resultante possa ter uns consecutivos em cada uma de suas linhas (ou colunas). Podemos notar que, um grafo bipartido é biconvexo se, e somente se, sua matriz de adjacência satisfaz a propriedade dos uns consecutivos tantos para linhas quanto para colunas. Note ainda que para cada vértice $v \in V \cup W$, $begin(v)$ e $end(v)$, coincidem respectivamente, com o índice da menor e da maior linha (ou coluna) de $M_{n \times m}$ correspondente a v.

Seja $G = (V, W, E)$ um grafo bipartido biconvexo. Considere dois vértices v_i e v_{i+1} de V. Estes dois vértices estão em uma mesma componente se, e somente se, há interseção entre os intervalos $[begin(v_i), end(v_i)]$ e $[begin(v_{i+1}), end(v_{i+1})]$. Desta forma, torna-se bastante simples determinar as componentes de G no modelo BSP/CGM. A saber, considere V globalmente ordenado por *begin* e uniformemente distribuídos entre os processadores. Cada P_k percorre os vértices de V_{P_k} e determina o número de componentes para V_{P_k}. Em seguida, cada P_k distribui este valor, juntamente com seu maior *end* e seu menor *begin*, para todo $P_j > P_k$. Com estas informações cada P_k determina qual componente cada um dos vértices atribuídos a ele pertence. Assim vamos considerar, sem perda de generalidade, que o grafo bipartido convexo é conexo.

Conforme observado por Lipski e Preparata[LP81] temos: dado um grafo bipartido biconvexo, então existem dois inteiros, u e l, $1 \leq u \leq l \leq n$, tais que valem as seguintes condições:

1. A seqüência $[begin(v_1), begin(v_u)]$ é decrescente e $[end(v_1), end(v_u)]$ é crescente.
2. As seqüências $[begin(v_{u+1}), begin(v_{l-1})]$ e $[end(v_{u+1}), end(v_{l-1})]$ são ambas crescentes ou ambas decrescentes.
3. A seqüência $[begin(v_l), begin(v_n)]$ é crescente e $[end(v_l), end(v_n)]$ é decrescente.

Assim, a matriz pode ser dividida em partes superior, do meio e inferior. Veja a Figura 5.6 com exemplo. Podemos fazer as duas seqüências da parte do meio crescentes,

$$G = \begin{array}{c|ccccccc} & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 \\ \hline v_1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ v_2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ v_3 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ v_5 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ v_7 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ v_8 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ v_9 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figura 5.10: Matriz de entrada $M_{9\times 7}$ representando um grafo G bipartido biconvexo propriamente ordenado. Neste caso $u = 4$ e $l = 9$ com $begin(v_4) = w_1$, $end(v_4) = w_4$, $begin(v_9) = w_6$ e $end(v_9) = w_6$.

se ela existe, pela inversão da ordem das colunas se necessário. Neste caso temos um grafo bipartido biconvexo propriamente ordenado. Em um grafo bipartido biconvexo, pode haver vários pares (u, l) satisfazendo as três condições acima. Denotaremos por $d(x, y)$ a distância, número de arestas, do caminho mínimo de x a y.

O Algoritmo BSP/CGM

Como entrada o algoritmo recebe uma matriz $n/p \times m$ propriamente ordenada onde cada linha desta matriz informa quais os vértices de W são adjacentes a cada um dos n/p vértices de V_{P_k}. O algoritmo que descrevemos obtém o valor final da linha correspondente ao vértice v_i. Uma árvore, T_i, dirigida de tamanho $n + m$ é então construída, tendo v_i como raiz e cada nó corresponde a um vértice do grafo de entrada. Esta árvore tem a propriedade de que o caminho entre a raiz e qualquer outro nó representa o caminho mínimo entre estes dois nós no grafo de entrada. A árvore é construída baseada nas seguintes regras:

1. A raiz da árvore é v_i
2. Se w_j é adjacente a v_i, então v_i é pai de w_j
3. Se v_i está na parte superior ($i \leq u$), então

   3.1 Se w_j não é adjacente a v_i, então $v_{begin(w_j)}$ é pai de w_j

   3.2 Se $j \neq i$ e $begin(v_1) \in [begin(v_j), end(v_j)]$, então $w_{begin(v_1)}$ é pai de v_j

   3.3 Se $begin(v_1) \notin [begin(v_j), end(v_j)]$, então $w_{begin(v_j)}$ é pai de v_j

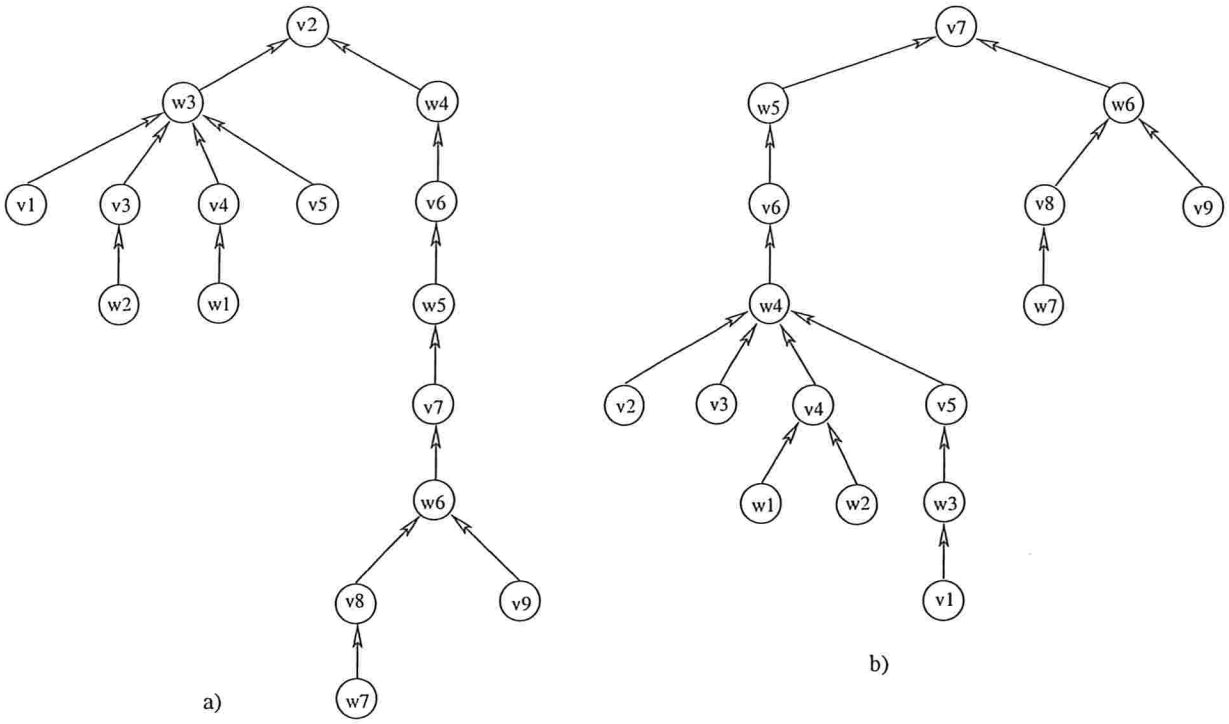


Figura 5.11: Exemplo de árvores de G para v_2 em a) e v_7 em b)

4. Se v_i está na parte do meio ($u < i < l$), então

   4.1 Se w_j não é adjacente a v_i e $end(w_j) < i$, então $v_{end(w_j)}$ é pai de w_j

   4.2 Se w_j não é adjacente a v_i e $end(w_j) > i$, então $v_{begin(w_j)}$ é pai de w_j

   4.3 $j < i$, então $w_{end(v_j)}$ é pai de v_j

   4.4 $j > i$, então $w_{begin(v_j)}$ é pai de v_j

5. Se v_i está na parte inferior ($i \geq l \neq u$), então a construção é análoga ao caso 3.

Chen [Che99] demonstrou que a árvore construída com estas regras, determina exatamente quantos e quais são os filhos de cada nó do grafo. No procedimento abaixo, cada processador P_k calcula os *begins* e *ends* dos vértices de V_{P_k}.

Procedimento Calcula begin_end

Entrada: Matriz de incidência M $n/p \times m$ propriamente ordenada.

Saída: *begin* e *end* de cada $v_i \in V_{P_k}$

1: **para** $i := 0$ **até** $n/p - 1$ **faça**

2: **para** $j := 0$ **até** m **faça**
3: **se** $M(i,j) = 0$ **e** $M(i,j+1) = 1$ **então**
4: $begin(v_i) := j + 1$
5: **se** $M(i,j) = 1$ **e** $M(i,j+1) = 0$ **então**
6: $end(v_i) := j$

O cálculo para os *begins* e *ends* dos vértices de W pode ser feito com o procedimento percorrendo todas as colunas da matriz para cada vértice de V_{P_k}. Se o *begin* e o *end* de cada w_i forem inicializados com $-\infty$ e $+\infty$, respectivamente, com um *broadcast* de cada valor calculado podemos escolher o maior e o menor valor, respectivamente, como sendo o *begin* e o *end* de w_i.

Após calcularmos os *begins* e *ends* devemos determinar os valores de u e l, construir a árvore T_i usando as regras 1 a 5 e calcular $d(i,v)$ de cada $v \in V(T_i)$. Para encontrar todos os pares de caminhos mínimos é necessário construir uma árvore para cada v_i. Uma vez calculado os *begins* e *ends* de cada vértice de G, podemos, com uma rodada de comunicação, armazenar em cada P_k os *begins* e *ends* de todos os vértices de G consumindo memória $O(m+n)$. Desta forma, podemos calcular u e l seqüencialmente [Che99] em tempo $O(n+m)$. A partir daí, cada árvore pode ser construída sem comunicação e com tempo de computação $O(n+m)$. Calcular $d(i,v)$ de cada $v \in V(T_i)$ é realizar um busca em largura em T_i e toma tempo $O(n+m)$. Como em cada P_k temos n/p árvores para construir e realizar as buscas, o tempo total é $O(n(n+m)/p)$.

Capítulo 6

Resultados das Implementações

Neste capítulo relatamos os resultados de tempo dos algoritmos que implementamos para os problemas de ordenação, emparelhamento, conjunto independente máximo e circuito hamiltoniano em grafos bipartidos convexos. Na Seção 6.1 comentamos sobre como foram gerados os grafos utilizados nos testes. Na Seção 6.2 tratamos das características das máquinas onde foram medidos os desempenhos dos algoritmos. Na Seção 6.3 descrevemos o padrão MPI, o qual foi utilizado para implementar os programas. Por fim, na Seção 6.4 detalhamos os resultados obtidos pelos diversos algoritmos através de gráficos e tabelas.

6.1 Tipos de Grafos

Para cada grafo bipartido convexo $G = (V, W, E)$ usado nos testes foi gerada sua representação compacta da seguinte forma. Escolhido o tamanho de W, para cada vértice $v \in V$ escolhemos aleatoriamente o meio do intervalo $[begin(v), end(v)]$ e com uma densidade pré-definida estabelecemos o tamanho do intervalo baseado na distribuição de Poisson.

Para testar a implementação dos diversos problemas criamos grafos com intervalos tendo como referência uma densidade de 8%, isto é, criamos intervalos com tamanho médio de 8% do tamanho de W. Este valor foi escolhido com o intuito de testar algumas peculiaridades do algoritmo para encontrar um MIS e do algoritmo para determinar se o grafo possui um circuito hamiltoniano. No primeiro caso (*Algoritmo MIS*), a densidade de 8% produz um número relativamente grande de subintervalos de W com vértices atingíveis. No segundo caso (*Algoritmo Circuito Hamiltoniano*), a densidade de 8% produz, em geral, um grafo não hamiltoniano e, com alta probabilidade, apenas um emparelhamento pode definir que o grafo não é hamiltoniano. Em virtude disto, produzimos também grafos com uma densidade de 45%, forçando o algoritmo a chamar duas vezes o algoritmo para

emparelhamento.

6.2 Características das Máquinas

Utilizamos duas máquinas para a realização de nossos testes de desempenho: o Biowulf/IME do Instituto de Matemática e Estatística (IME-USP) e o I-cluster do Instituto francês ID-INRIA. Estas máquinas têm em comum o sistema de processamento paralelo do tipo Beowulf, que consiste de um conjunto de PCs ou *workstations* interconectados por uma rede de alto desempenho dedicado para rodar aplicações paralelas. Neste sistema as máquinas interligadas estão livres de perturbações externas e se conectam com o mundo exterior por meio de um único nó. Este sistema tem ganho popularidade atualmente por apresentar bons resultados computacionais a custo relativamente baixo, além de sua expansibilidade.

6.2.1 A Máquina Biowulf/IME

Máquina Biowulf/IME é um *cluster* com 16 PCs conectados por um *switch* Fast Ethernet. A Figura 6.1 mostra a estrutura de conexão entre os processadores. Além do *Switch* tipo Superstack III 3300 10/100, no sistema cada PC possui a seguinte configuração:

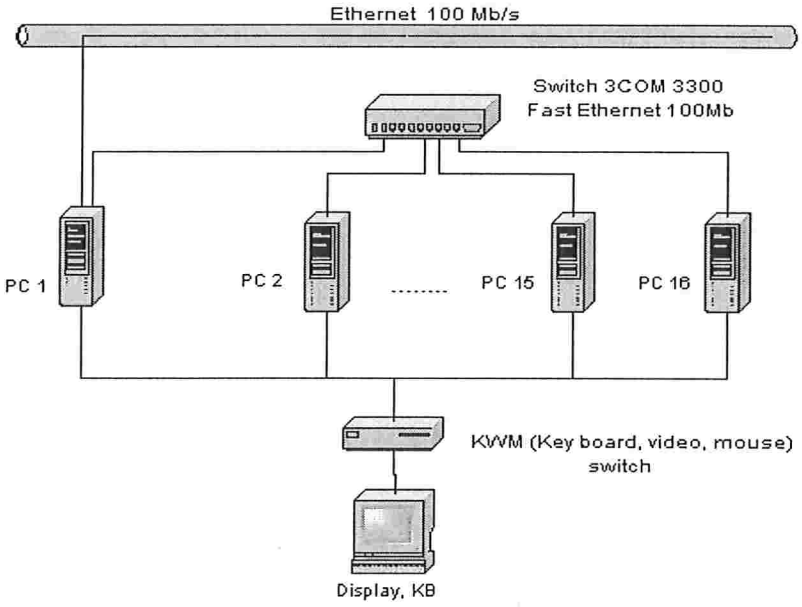
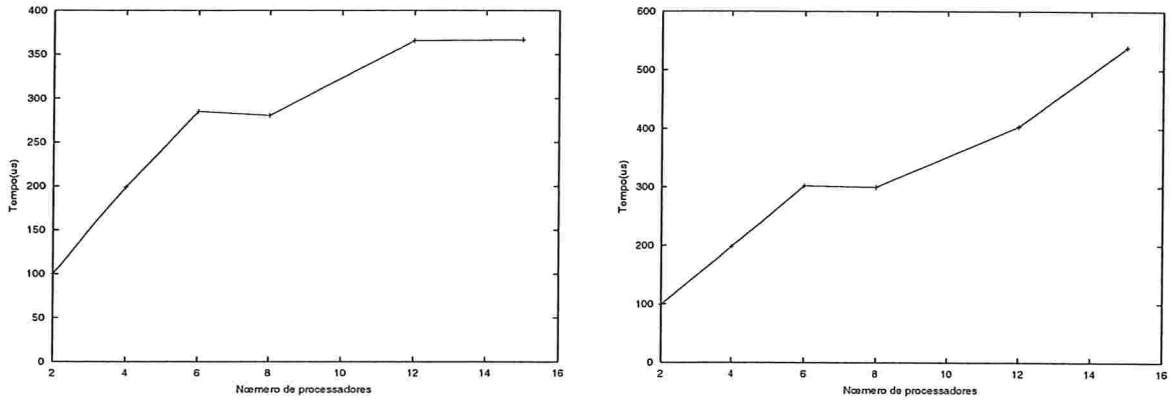


Figura 6.1: Configuração da máquina Biowulf/IME

Processador: AMD Athlon 1.2 GHz, cache L2 com 256 KB
Memória: 768 MB PC133 SDRAM

Quanto à configuração do Software, o Sistema Operacional é o Debian Linux 2.2.

As Figuras 6.2(a) e 6.2(b) mostram, respectivamente, o desempenho da máquina Biowulf/IME com relação ao tempo de *broadcast* de uma mensagem de 1KByte e da barreira de sincronização quando variado o número de processadores.



(a) *Broadcast* de 1KByte

(b) Barreira de sincronização

Figura 6.2: Medidas de tempo na Biowulf/IME

6.2.2 A Máquina I-Cluster/INRIA

A Figura 6.3 mostra a estrutura de conexão entre os processadores da I-Cluster/INRIA.

A máquina I-Cluster do INRIA possui 216 PCs, três *switches* Fast Ethernet com 40 nós cada e dois *switches* com 48 nós cada. Estes *switches*, por sua vez, estão interligados entre si com uma largura de banda de 1 GB. Além dos *Switches* tipo HP procurve 4000 100MB/1GB, no sistema cada PC possui a seguinte configuração:

Processador: Intel Pentium III 733 MHz
Memória: 256 MB PC133 SDRAM

Quanto à configuração do Software, o Sistema Operacional usado é o Mandrake Linux 2.2.

As Figuras 6.4(a) e 6.4(b) mostram, respectivamente, o desempenho da máquina I-Cluster/INRIA com relação ao tempo de *broadcast* de uma mensagem de 1KByte e da barreira de sincronização quando variado o número de processadores.

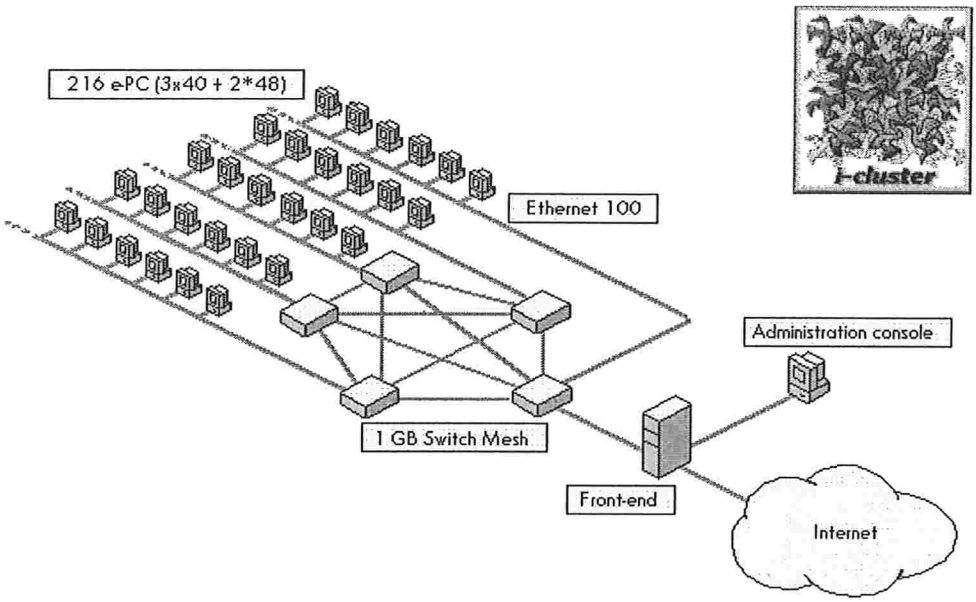
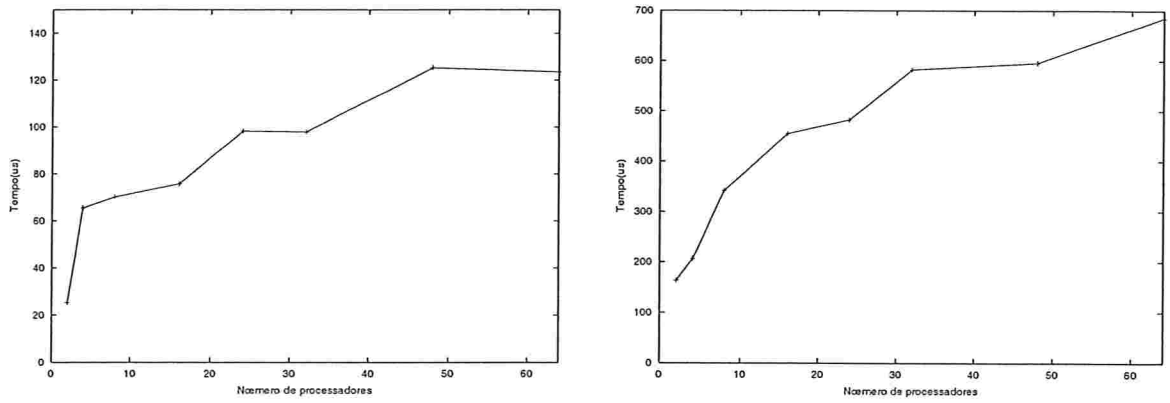


Figura 6.3: Configuração da máquina I-Cluster/INRIA



(a) *Broadcast* de 1KByte

(b) Barreira de sincronização

Figura 6.4: Medidas de tempo na I-Cluster/INRIA

6.3 MPI

MPI — Message-Passing Interface — é uma biblioteca de funções e macros que pode ser utilizada em programas em C, C++ e Fortran. O objetivo do MPI, como o nome indica, é ser usado em programas que exploram a existência de múltiplos processadores através da troca de mensagens. O padrão MPI foi desenvolvido por um grupo de pesquisadores nas áreas acadêmica, industrial e governamental. Este padrão possui várias implementações. As mais conhecidas são MPICH, LAM/MPI e Chimb.

Em MPI, os processadores envolvidos na execução de um programa paralelo são identificados por uma seqüência de inteiros não-negativos. Se há p processadores executando o programa, cada um será identificado por um valor dentre $0, 1, \ldots, p-1$. O que ocorre quando um programa é executado em uma máquina paralela, essencialmente, é o seguinte:

1. O usuário envia uma diretiva para o sistema operacional que se encarrega de colocar o programa executável na memória local de cada uma dos processadores.
2. Cada processador inicia a execução de sua cópia do executável.
3. Diferentes processadores podem executar diferentes comandos através de expressões condicionais do programa, baseadas nos identificadores dos processadores, por exemplo.

Desta forma, o MPI se adequa perfeitamente ao paradigma SPMD — *Single Program Multiple Data* — que é a forma mais utilizada para escrever programas MIMD — *Multiple Instruction Multiple Data*. Além disto, o MPI possibilita também o uso de diferentes programas em diferentes processadores.

Um programa MPI típico em C tem o seguinte formato:

```
#include ''mpi.h'' /* contém as macros e protótipos de funções     */
                   /* necessárias para compilar um programa MPI    */
...

main(int argc, char** argv){

   ...

   MPI_Init(NULL,NULL); /* funções MPI devem ser chamadas depois daqui */

    <comandos>
```

```
    MPI_Finalize(); /* funções MPI devem ser chamadas antes daqui */

    ...

} /* main */
```

O MPI possui centenas de funções que possibilitam e facilitam a comunicação entre dois ou mais processadores. Abaixo destacamos as quatro principais e sua sintaxe.

`int MPI_Comm_rank(MPI_Comm comm, int rank)`

Retorna o `rank`, o identificador, do processador em seu segundo argumento. Seu primeiro argumento é um *communicator*, estrutura de dados que especifica o escopo de uma operação de comunicação, isto é, um grupo de processadores envolvidos no contexto da comunicação. O *communicator* predefinido MPI_COMM_WORLD consiste de todos os processadores rodando o programa quando a execução começa. O `rank` é interpretado com respeito ao grupo de processadores associados com o *communicator*.

`int MPI_Comm_size(MPI_Comm comm, int rank)`

Determina o número de processadores executando o programa em um *communicator*. Seu primeiro argumento é um *communicator*. Seu segundo argumento retorna o número de processadores.

```
int MPI_Send(void *message, int cont, MPI_Datatype dtype,
             int dest, int tag, MPI_Comm comm)
```

Envia a mensagem contida em um bloco de memória e apontada por `message` para o processador `dest`. Esta mensagem possui tamanho `cont` de tipo `dtype`. Este tipo pode ser um dos predefinidos pelo MPI, ou construído pelo usuário. O inteiro `tag` especificado pelo usuário pode ser usado para distinguir mensagens diferentes enviadas por um mesmo processador-emissor e recebidas por um mesmo processador-receptor.

```
int MPI_Recv(void *message, int cont, MPI_Datatype dtype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Recebe a mensagem no bloco de memória apontada por `message` vinda do processador `source`. Aqui os parâmetros `source` e `tag` podem ser as constantes predefinidas MPI_ANY_SOURCE e MPI_ANY_TAG respectivamente, possibilitando que a mensagem tenha qualquer origem e qualquer `tag`. Para tratar esta situação, além dos parâmetros da função `MPI_Send`, `MPI_Recv` possui o parâmetro `status` que aponta para um registro com um campo definindo o processador-emissor e outro definindo a `tag`.

6.4 Resultados Obtidos

Para a implementação de nossos algoritmos utilizamos grafos, gerados conforme descrito na Seção 6.1, com $|V| = |W|$. No que segue denotamos $G1$, $G5$ e $G10$ como sendo a média dos tempos dos grafos com $|V| = 1 \times 10^6$, $|V| = 5 \times 10^6$ e $|V| = 1 \times 10^7$, respectivamente. Para a exposição dos resultados dos algoritmos que apresentaram melhores resultados práticos utilizamos o diagrama de ganho de tempo (*speedup*). Utilizamos o diagrama tempo $\times$ processadores para a exposição dos resultados de tempo obtidos pelos algoritmos que não apresentaram bons resultados práticos. Além disso, em alguns casos utilizamos tabelas para evidenciar algum fato relevante.

Nestas implementações utilizamos o LAM/MPI 6.5. Os testes foram executados para duas instâncias de cada tamanho, três vezes para cada instância. Em cada execução foi tomado o tempo máximo, somando-se computação e comunicação, gasto pelos processadores. Os tempos mostrados refletem a média destas seis execuções. Em todas as máquinas, as medidas de tempo se mostraram bastante robustas em cada instância, não sendo necessário, desta forma, executar um número maior de testes para cada instância. O algoritmo seqüencial de cada problema foi tomado como referência para o cálculo dos *speedups*.

Devido à falta de memória na máquina I-cluster/INRIA não foi possível executar o algoritmo seqüencial para instâncias de tamanho 10^7 dos algoritmos para emparelhamento, MIS e Caminho Hamiltoniano. Como conhecemos o tempo para instâncias de tamanho 10^6 e as funções assintóticas de tempo para os problemas, para cada problema fizemos uma estimativa de tempo usando a seguinte fórmula:

$$T(n) = T(10^6)f(n)/f(10^6)$$

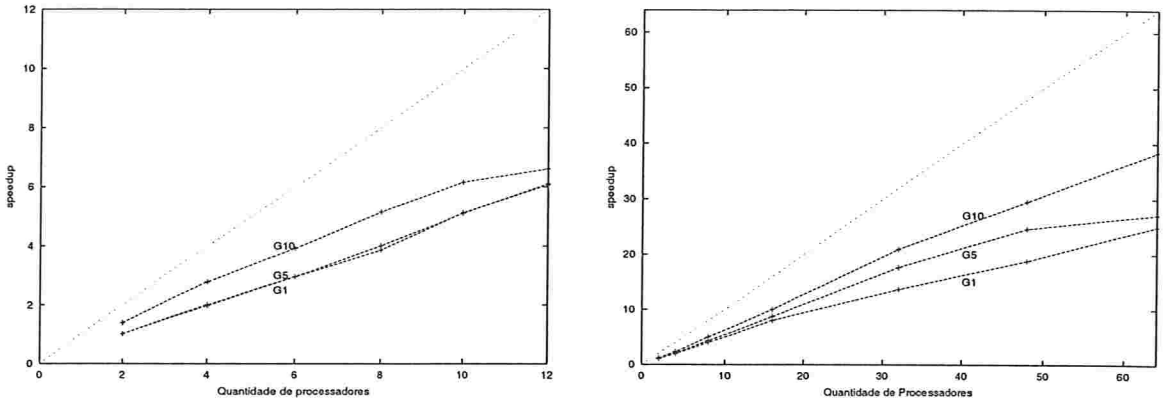
onde:
$T(n)$ é o tempo gasto pelo problema para a instância de tamanho n
$f(n)$ é função de tempo conhecida para o problema de tamanho n.

Para o problema do emparelhamento a função $f(n)$ é $n \log n$ e para o problema MIS $f(n)$ é n.

6.4.1 Ordenação

As Figuras 6.5(a) e 6.5(b) mostram os ganhos de tempo (*speedups*) do algoritmo para ordenação nas máquinas Biowulf/IME e I-cluster/INRIA quando variado o número de processadores.

Os *speedups* obtidos para a ordenação foram bons, principalmente se considerarmos a situação em que o custo de comunicação é muito elevado como a que trabalhamos.



(a) Biowulf/IME

(b) I-cluster/INRIA

Figura 6.5: *Speedup* do Algoritmo para Ordenação

Observamos que o *speedup* variou entre 1,02 e 5,18 com 8 processadores na Biowulf/IME, e entre 1,03 e 38,42 com 64 processadores I-cluster/INRIA.

6.4.2 Emparelhamento

As Figuras 6.6(a) e 6.6(b) mostram o desempenho do algoritmo para emparelhamento nas máquinas Biowulf/IME e I-cluster/INRIA quando variado o número de processadores.

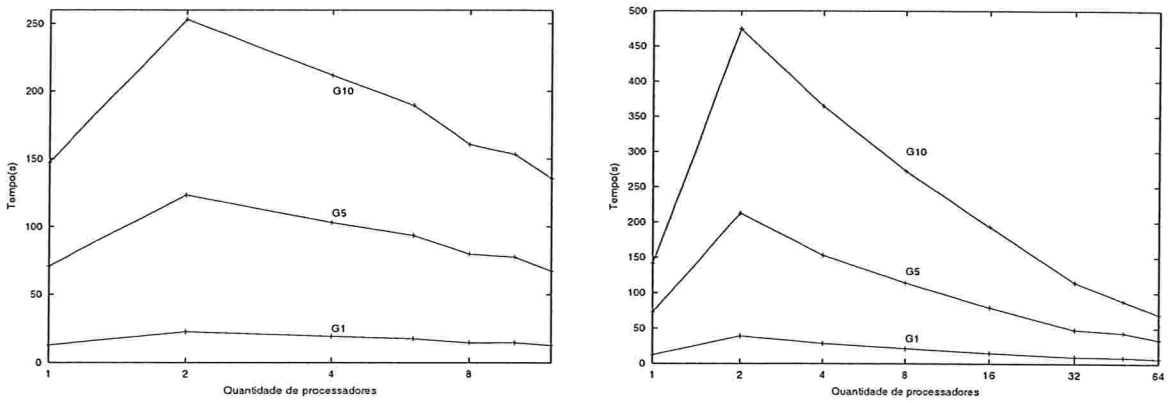


Figura 6.6: tempo × processadores do algoritmo para emparelhamento na Biowulf/IME e na I-cluster/INRIA

Os *speedups* obtidos para o emparelhamento não foram muito bons. Conforme Figura 6.7, podemos ver que o *speedup* variou entre 0,31 e 0,56 com 8 processadores na

máquina Biowulf/IME, e entre 0,31 e 2,10 com 64 processadores na máquina I-cluster/INRIA.

	Biowulf/IME			I-cluster/INRIA		
P	1×10^6	5×10^6	1×10^7	1×10^6	5×10^6	1×10^7
2	0,32	0,31	0,35	0,32	0,34	0,30
4	0,37	0,38	0,42	0,43	0,55	0,39
8	0,49	0,49	0,56	0,58	0,64	0,52
16				0,84	0,92	0,73
32				1,35	1,53	1,24
64				1,85	2,16	2,04

Figura 6.7: *Speedup* do algoritmo para emparelhamento em grafos com $|V| = 1 \times 10^6$, 5×10^6 e 1×10^7

O fato dos *speedups* obtidos para o emparelhamento não terem sido bons se deve a dois fatores principais: o custo de comunicação é muito elevado em relação à computação local nas máquinas que trabalhamos e, além disso, os algoritmos que implementamos são $O(n \lg n)$ para a ordenação e $O(n)$ para o emparelhamento. Isto faz com que o tempo de comunicação necessário seja relativamente grande comparado ao tempo de computação local realizado. Este fato pode ser constatado pela tabela da Figura 6.8.

	1×10^6		5×10^6		1×10^7	
	Comp	Com	Comp	Com	Comp	Com
2	16,99	5,62	94,32	28,95	194,28	58,71
4	13,09	6,16	71,81	31,17	146,72	64,98
6	8,89	8,65	49,10	44,30	101,94	87,49
8	9,94	5,11	48,97	30,79	103,69	52,92
10	7,16	7,44	39,27	38,24	82,41	73,85
12	6,07	6,06	33,98	33,32	70,74	64,85

Figura 6.8: Tempo de computação e tempo de comunicação do algoritmo para emparelhamento em grafos com $|V| = 1 \times 10^6$, 5×10^6 e 1×10^7

6.4.3 MIS

Para o algoritmo que encontra o MIS utilizamos dois tipos de entradas distintas: quando na entrada não temos o emparelhamento calculado, neste caso o tempo inclui o cálculo do emparelhamento no grafo; e quando na entrada já temos o emparelhamento, não sendo, portanto, necessário incluir o tempo do emparelhamento.

As Figuras 6.9(a) e 6.9(b) mostram o desempenho do algoritmo para encontrar o MIS nas máquinas Biowulf/IME e I-cluster/INRIA quando variado o número de processadores. Neste caso, o tempo inclui o cálculo do emparelhamento no grafo, o qual acarreta

uma degradação na performance do algoritmo como um todo, uma vez que o cálculo do emparelhamento domina o tempo do algoritmo para encontrar o MIS.

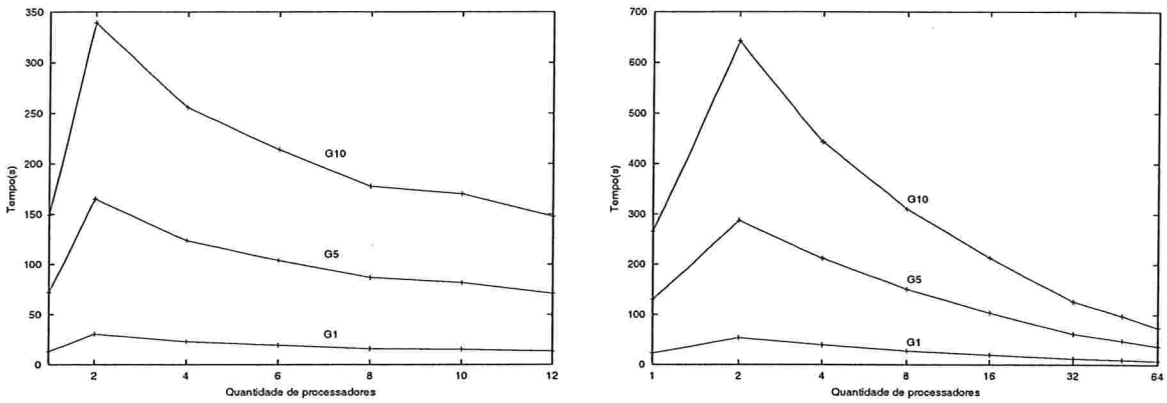


Figura 6.9: tempo × processadores do algoritmo MIS na Biowulf/IME e na I-cluster/INRIA com tempo do emparelhamento incluso.

Nas Figuras 6.10(a) e 6.10(b), ilustramos o comportamento do algoritmo MIS, através da curva de *speedup*, quando o grafo de entrada já possui o emparelhamento calculado. Nesta implementação houve um redução significativa do tempo de comunicação e uma conseqüente melhora do algoritmo como um todo.

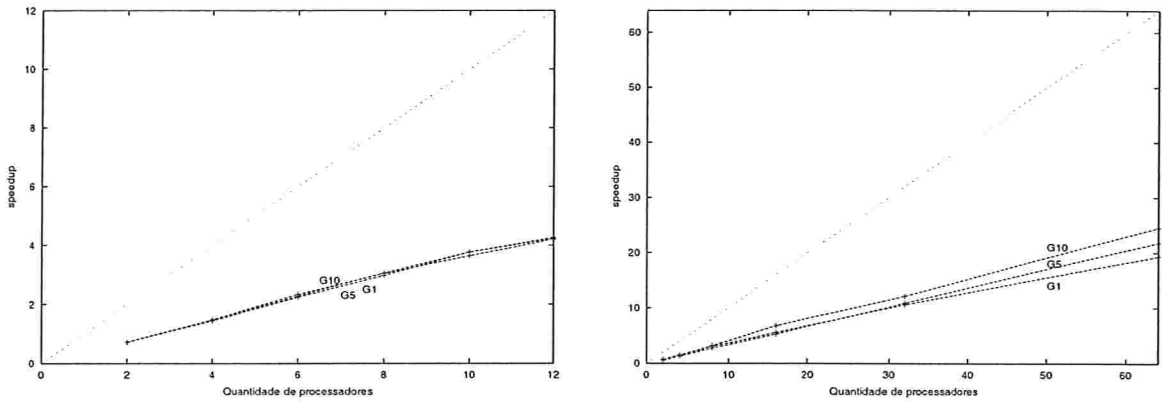


Figura 6.10: *Speedup* do MIS na Biowulf/IME e na I-cluster/INRIA sem o tempo para emparelhamento.

6.4.4 Circuito Hamiltoniano

Nos gráficos a seguir vemos, para dois tipos de entradas distintas, o tempo do algoritmo para o circuito hamiltoniano quando variado o número de processadores.

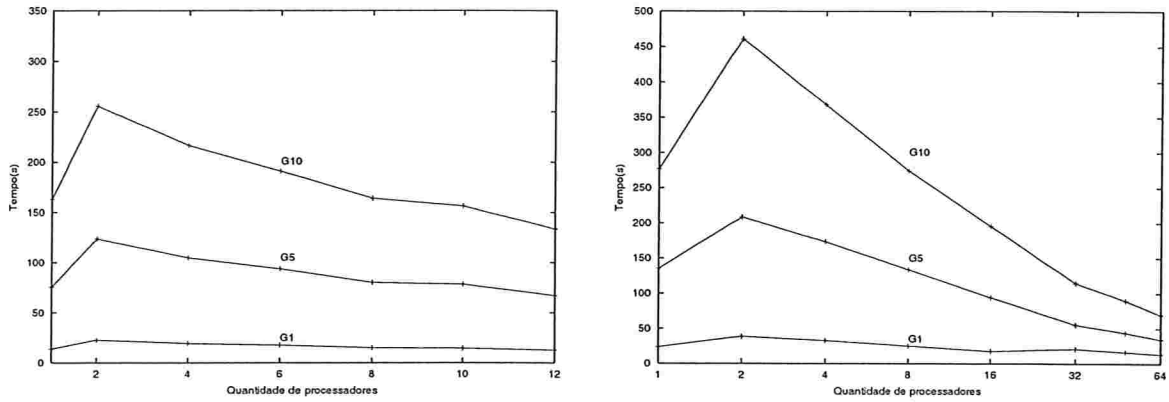


Figura 6.11: Diagrama tempo × processadores do caminho hamiltoniano na Biowulf/IME e na I-cluster/INRIA em grafos com densidade de 8%.

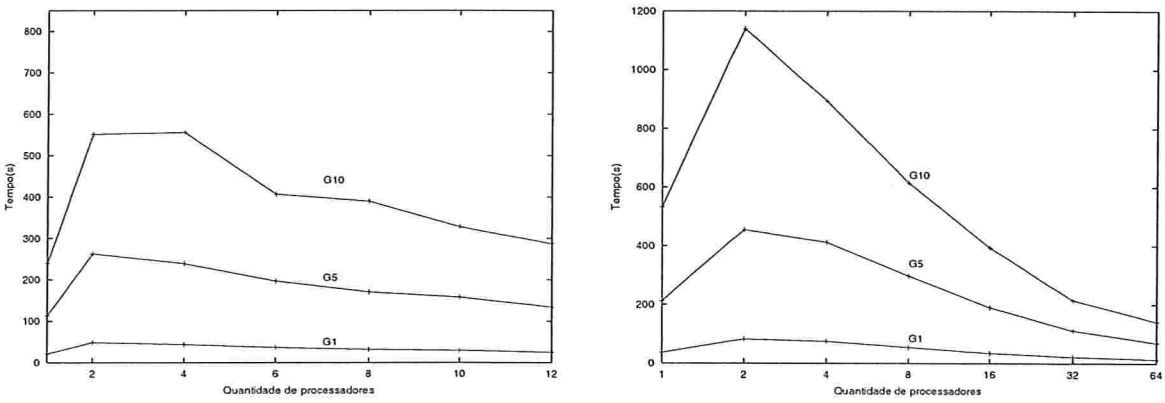


Figura 6.12: Diagrama tempo × processadores do caminho hamiltoniano na Biowulf/IME e na I-cluster/INRIA em grafos com densidade de 45%.

Como podemos ver pelos gráficos a seguir, o tempo do algoritmo para o circuito hamiltoniano é essencialmente o tempo para realizar o emparelhamento. No primeiro caso, Figuras 6.11(a) e 6.11(b), o grafo não é hamiltoniano e o algoritmo realiza apenas um emparelhamento para fazer a verificação, e no segundo caso, Figuras 6.12(a) e 6.12(b), os grafos são hamiltonianos e o algoritmo chama duas vezes o algoritmo para emparelhamento para fazer a verificação.

Capítulo 7

Considerações Finais

Apresentamos neste trabalho uma visão geral dos modelos de computação paralela atualmente em uso, especialmente aqueles chamados realísticos. Relacionamos alguns critérios que consideramos relevantes para um modelo de computação paralela satisfazer. Longe de ser um consenso, a discussão que realizamos é apenas parcial e bastante incipiente devido à complexidade do próprio tema. A despeito disto, o surgimento de novas áreas de aplicações como Inteligência Artificial e Biologia Computacional, além das tradicionais, tem intensificado as pesquisas na área.

Não obstante, novas arquiteturas paralelas têm surgido nos últimos anos, e um grande esforço de se encontrar um paradigma adequado para o desenvolvimento de algoritmos tem sido feito. O primeiro se deve como conseqüência ao barateamento do *hardware* e o desenvolvimento de máquinas do tipo Beowulf, e o segundo tem se refletido, principalmente nos modelos realísticos, BSP, CGM e LogP. Em geral, os algoritmos desenvolvidos para estes modelos correspondem a implementações satisfatórias nas máquinas atuais.

Posterior à discussão sobre os modelos e apresentação de alguns algoritmos de comunicação dentro destes modelos, descrevemos dois algoritmos de ordenação nos modelos realísticos que incorporam as discussões feitas inicialmente.

O tema de nosso estudo foi encontrar algoritmos eficientes em modelos realísticos para problemas em grafos cuja paralelização, a partir de algoritmos desenvolvidos para o modelo PRAM, não são triviais. Neste estudo enfatizamos tipos especiais de grafos, a saber, grafos cujo grau máximo é limitado e grafos bipartidos convexos. Em grafos cujo grau máximo é limitado, descrevemos um algoritmo para coloração de vértices.

Estudamos, como principal foco de nosso trabalho, grafos bipartidos convexos, para os quais foram apresentados alguns novos algoritmos paralelos realísticos. O primeiro problema tratado foi encontrar um emparelhamento máximo. Este algoritmo levou ao desenvolvimento de um algoritmo para encontrar um conjunto independente máximo e a

resolver o problema do circuito hamiltoniano neste tipo de grafo. Além destes problemas, apresentamos um algoritmo para encontrar todos os caminhos mínimos em uma subclasse deste tipo de grafo, os grafos bipartidos biconvexos.

A implementação dos algoritmos para grafos bipartidos convexos comentados anteriormente foram relevantes para uma avaliação cuidadosa dos modelos. Estas implementações foram realizadas em duas máquinas diferentes e o comportamento dos algoritmos se mostraram semelhantes, o que afirma a portabilidade do modelo. Quanto a complexidade de tempo a comunicação entre processadores se apresenta como um gargalo a ser superado. Ademais, as ferramentas de programação paralela, apesar de avanços recentes, estão em amadurecimento.

Como sugestão para estudos futuros, temos o problema de encontrar um conjunto dominante mínimo em grafos bipartidos convexos. Os problemas que estudamos para este tipo de grafos, merecem um estudo em classes mais gerais de grafos, por exemplo, em grafos de intervalo. Apesar de não ter sido tratado em nossa tese, a área de biologia computacional com os modelos realísticos oferece problemas interessantes. Nos algoritmos que apresentamos em grafos bipartidos convexos o número de rodadas foi $O(\lg p)$, a possibilidade de encontrar algoritmos com número de rodadas constante pode ser considerada.

Referências Bibliográficas

[AISS97] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model for parallel computation. *J. of Parallel and Distributed computing*, 44:17–79, 1997.

[BCDL99] P. Bose, A. Chan, F. Dehne, and M. Latzel. Coarse grained parallel maximum matching in convex bipartite graphs. In *13th International Parallel Processing Symposium (IPPS'99)*, pages 125–129, 1999.

[BDadH98] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, 203:175–203, 1998.

[BL76] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13:335–379, 1976.

[CCDP00] E. Caceres, A. Chan, F. Dehne, and G. Prencipe. Coarse grained parallel algorithms for detecting convex bipartite graphs. In *26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000)*, Konstanz, Germany, 2000.

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

[CDP96] A. Czumaj, K. Diks, and T. M. Przytcka. Parallel maximum independent set in convex bipartite graphs. *Information Processing Letters*, 59:289–294, 1996.

[Che99] L. Chen. Optimal computation of shortest paths on doubly convex bipartite graphs. *Computer and Mathematics with Applications*, 38:1–12, 1999.

[CKS+93] D. Culler, R. Karp, D. Patterson. A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 4, pages 1–12, 1993.

[CLR90] T. H. Cormen, C. E. Lieserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[DCSM96] Dusseau, D. Culler, K. Schauser, and Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transitions on Parallel and Distributed Systems*, 7, 1996.

[DFRC93] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *9th Annual ACM Symposium on Computational Geometry*, 1993.

[DS84] E. Dekel and S. Sahni. A parallel matching for convex bipartite graphs and applications to scheduling. *Journal of Parallel and Distributed Computing*, 1:185–205, 1984.

[Gal84] G. Gallo. An $O(n \log n)$ algorithm for the convex bipartite matching problem. *Operations Research Letters*, 3:31–34, 1984.

[GJ79] M. Garey and D. Johnson. *Computers and Intractabilits, A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[Glo67] F. Glover. Maximum matching in convex bipartite graphs. *Naval Research Logistic Quartery*, 14:313–316, 1967.

[Gol80] M. Golumbic. *algorithmic graph theory and perfect graphs.* Academic Press, 1980.

[Goo96] M. T. Goodrich. Communication efficient parallel sorting. In *28th Annual ACM Symposium on Theory of Computing (STOC'96)*, 1996.

[GPS88] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Mathematics*, 1:434–446, 1988.

[GT85] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjunt set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[Jáj92] Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[KGGK94] V. Kumar, A. Grama, A. Grupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin-Cummings, 1994.

[Kri75] M. Krishnamoorthy. An NP-hard problem in bipartite graphs. *SIGACT News*, 7(1):26, 1975.

[Kuh55] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Resarch Logistics Quarterly*, 2:83–97, 1955.

[LB95] Y. D. Liang and N. Blum. Circular convex bipartite graphs: maximum matching and hamiltonian circuits. *Information Processing Letters*, 56:215–219, 1995.

[Lei85] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE transactions on Computers*, 34, 1985.

[LP81] W. Lipski and F. P. Preparata. Efficient algorithms for finding maximum matching in convex bipartite graphs and related problems. *Acta informatica*, 15:329–346, 1981.

[Mul96] H. Muller. Hamiltonian circuits in chordal bipartite graphs. *Discrete Mathematics*, 156:291–298, 1996.

[SS01] J. Soares and M. Stefanes. Coarse grained parallel maximum independent set in convex bipartite graphs. In *PDPTA*, 2001.

[SY96] G. Steiner and J.S. Yeoman. A linear time algorithm for maximum matchings in convex, bipartite graphs. *Computers and Mathematics with Applications*, 31(12):91–96, 1996.

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