Um algoritmo de aproximação paralelo para transversal mínima com aplicação em análise da expressão gênica

Danielle Passos de Ruchkys

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

Orientador: Prof. Dr. Siang Wun Song

– Durante o desenvolvimento deste trabalho, a autora recebeu apoio financeiro do CNPQ –

— São Paulo, dezembro de 2002 —

Um algoritmo de aproximação paralelo para transversal mínima com aplicação em análise da expressão gênica

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Danielle Passos de Ruchkys e aprovada pela comissão julgadora.

São Paulo, 10 de dezembro de 2002.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) (IME-USP)
- Prof. Dra. Nina Sumiko Tomita Hirata (IME-USP)
- Prof. Dr. Edson Norberto Cáceres (UFMS)

Aos meus pais Tito e Edméa,
ao meu padrinho José (*in memoriam*) e
à tia Mita ("... dos nossos planos é que tenho
mais saudade, quando olhávamos juntas na mesma
direção. Aonde está você agora além de aqui dentro
de mim? Agimos certo sem querer, foi só o tempo
que errou, vai ser difícil sem você, porque você
está comigo o tempo todo, e quando vejo o mar,
existe algo que diz que a vida continua e se
entregar é uma bobagem. Já que você não está
aqui, o que posso fazer é cuidar de mim."
- Vento no Litoral, letra de Renato Russo)

Agradecimentos

Gostaria de agradecer a Deus por ter sido tão bondoso, por ter me conduzido por este caminho e por ter escolhido e recusado coisas para a minha vida que com o meu pouco conhecimento não saberia fazer sozinha.

Ao meu orientador Siang Wun Song por sua tamanha atenção, paciência, doçura e generosidade durante o desenvolvimento deste trabalho. Não poderia ter escolhido para mim um orientador melhor, não somente pela competência na área, que dispensa comentários, mas também pela humanidade com que ele conduz seus orientandos e pela segurança que ele transmite. Aos professores Edson Cáceres e Cristina Gomes Fernandes pelas valiosas contribuições a este trabalho. À Martha Delgado pela ajuda que me deu com a máquina paralela.

Aos meus pais. É muito difícil agradecer por tudo o que meus pais fizeram por mim nesses 26 anos, e mais difícil ainda é agradecer pelos mais recentes, que são os anos em que me mudei de Campo Grande para fazer o mestrado em São Paulo. Apenas ouvir suas vozes de confiança e carinho pelo telefone já seria o melhor dos alentos para as horas mais difíceis, mas além disso eles me deram apoio em todas as áreas necessárias para o desenvolvimento deste trabalho. Agradeço também pela confiança que depositam em mim.

Às minhas irmãs Valéria e Thaty, e à minha avó Losa por me darem apoio e por me fazerem sentir que havia pessoas torcendo por mim. Também por elas senti uma saudade boa da minha casa, aquela saudade que diz que se as coisas apertarem muito por aqui, tenho para os braços de quem correr.

Ao Said pelo apoio. Para o Said corri em um momento um pouco crítico e ele ficou ao meu lado por quase todos os dias deste ano, me ouvindo, falando, aconselhando, ajudando... Fez por mim tudo o que ele podia, e ainda mais do que posso esperar de um verdadeiro amigo. Com sua experiência, deu contribuições imensas a este trabalho. O Said é um anjo na vida de todos os seus amigos e especialmente na minha vida. Não tenho como agradecer, nada do que eu possa fazer pelo Said pagaria o que ele fez por mim. Passamos por momentos difíceis juntos e quase conseguimos convencer o pessoal do IME de que não somos nem casados e nem namorados. E para o Said, fica a pergunta de todos os dias, logo depois do almoço, muitas vezes sem resposta: "o que você vai jantar hoje?".

Agradeço à Ana Lúcia pelo aprendizado de companherismo e convivência que tivemos juntas, pelas risadas das horas felizes (e foram tantas) e pelo consolo nas horas tristes. Aprendemos juntas a morar aqui em São Paulo e ela cursou comigo as disciplinas, estudamos sempre juntas. Por ter participado da minha Crisma, o que foi muito importante para meu fortalecimento espiritual durante o mestrado, e além de tudo ainda ser minha madrinha. A Ana é uma pessoa de grande sensibilidade e de grande coração... espero sempre estar dentro dele. Ela sempre me deu a maior

atenção, exceto nas horas em que a gente assiste televisão, “mas Tatu, você é muito malvada!!!”. Said, Paula e Ana Lúcia formaram a atenta platéia para a qual eu apresentei minhas prévias. Com a Ana estive desde o início do mestrado e para ela dedico a amizade mais sincera.

À Paulinha, por ter trazido luz para a nossa casa. Como somos felizes, não é Paulinha? Depois que você chegou, dobramos o número de risadas por dia. Além disso, a Paula está quase aprendendo como fazer um algoritmo depois de tanto ter assistido às minhas prévias. À Paula e a Liz (Akemily) agradeço o ambiente de companheirismo em que vivemos, o que é muito importante durante o mestrado.

Ao Fábio Viduani pelos livros e pela força no início do mestrado. Ao Rodrigo Souza pela ajuda com o meu texto e pelo companheirismo. Aos amigos como um todo pela torcida. Agradecimentos especiais aos amigos que fiz aqui no IME, Lê, Mateus, Mimiça, Daniel, Bianka, Gui, Alessandro, Weslley, Rosianni, entre outros, que tiveram muita importância durante o mestrado, nas horas de estudo e nas horas de lazer e que enfrentaram análise de algoritmos junto comigo. Ao Pil, por estar sempre pronto a ajudar. Aos meus familiares, à minha madrinha e a todos os amigos que oraram por mim.

Aos funcionários da secretaria de pós-graduação, especialmente à Francisca, Pinho e Feijão pela atenção e boa vontade com que tratam os alunos.

Ao Luiz Vitor, por me trazer a paz e por me fazer acreditar que há momentos em que a vida é cor-de-rosa, como as suas bochechas.

Danielle Passos de Ruchkys

"Você não sabe o quanto eu caminhei pra chegar até aqui. Percorri milhas e milhas antes de dormir, eu não cochilei. Os mais belos montes escalei nas noites escuras de frio chorei. Meu caminho só meu Pai pode mudar, meu caminho só meu Pai. Meu caminho só meu Deus pode mudar".
A Estrada (Marisa Monte/Nando Reis)

Resumo

Com o recente progresso de tecnologias como dos DNA-*microarrays*, é possível medir os níveis de expressão de milhares de genes simultaneamente no mesmo experimento. Uma rede genética é um modelo que descreve como o nível de expressão de cada gene é afetado pelos níveis de expressão de outros genes na rede. Dados os resultados de um experimento com n genes e m medidas de tempo ($m << n$), consideramos o problema de encontrar um subconjunto de genes (k genes, onde $k << n$) que explica o nível de expressão de um dado gene alvo que desejamos estudar. Neste trabalho, nós apresentamos um algoritmo de aproximação seqüencial de complexidades de tempo $O(km^2n)$ e de espaço $O(m^2n)$. Temos que k=$O(m^2)$ e a complexidade de tempo apenas em termos de m e n é $O(m^4n)$. Para desenvolver o algoritmo paralelo, usamos o modelo CGM, com p processadores. O principal resultado é um algoritmo de aproximação paralelo que determina os k genes em tempo de computação local $O(\frac{km^2n}{p})$ (ou $O(\frac{m^4n}{p})$) mais $O(k)$ (ou $O(m^2)$) rodadas de comunicação, utilizando espaço local $O(\frac{m^2n}{p})$. Neste trabalho, também mostramos os tempos de execução obtidos pelos programas que implementam os algoritmos paralelo e seqüencial, utilizando uma máquina do tipo *Beowulf*, tendo como entrada matrizes de expressões geradas aleatoriamente. Também mostramos as acelerações obtidas pelo programa paralelo.

Abstract

With the recent technological advances in DNA-microarrays, it is possible to measure the expression levels of thousands of genes simultaneously in the same experiment. A genetic network is a model that describes how the expression level of each gene is affected by the expression levels of other genes in the network. Given the results of an experiment with n genes and m measures over time ($m << n$), we consider the problem of finding a subset of genes (k genes, where $k << n$) that explain the expression level of a given target gene under study. In this work we first present a sequential approximation algorithm of $O(km^2n)$ time and $O(m^2n)$ space complexities. We have k=$O(m^2)$ and the time complexity only in terms of m and n is $O(m^4n)$. To design the parallel algorithm, we use the coarse-grained multicomputer (CGM) model, with p processors. The main result is a new parallel approximation algorithm that determines the k genes in $O(\frac{km^2n}{p})$ (or $O(\frac{m^4n}{p})$) local computing time plus $O(k)$ (or $O(m^2)$) communication rounds, and with local space requirement of $O(\frac{m^2n}{p})$. In this work we also show the execution times obtained by the parallel and sequential programs that implement the algorithms, by using a Beowulf machine and input matrices of randomly generated expression values. We also show the speedups obtained by the parallel program.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A biologia computacional vem emergindo como uma das principais áreas de pesquisa, com aplicações de significados imensuráveis. Devido à complexidade dos problemas biológicos e principalmente à grande quantidade de dados envolvidos, os algoritmos para resolver tais problemas estão quase sempre associados a longos tempos de execução. Com isso, é natural e indiscutível a importância da utilização de computadores paralelos nas pesquisas em biologia computacional.

Um dos principais objetivos dos estudos em biologia molecular é adquirir um completo entendimento das funções dos genes, bem como das regras que governam suas interações. Um dos passos mais promissores na direção do complexo objetivo citado acima é a *análise da expressão gênica* [33]. Segundo Simmons e Snustad [31], expressão gênica é o processo no qual os genes produzem RNA e proteínas e exercem seus efeitos no fenótipo[1] de um organismo . Em geral, toda célula contém o genoma completo. Mas, para cada célula, a Natureza sabe exatamente quais genes precisam ser expressos, no tempo apropriado, durante o desenvolvimento do organismo. De uma forma simples, um gene é dito expresso se é transcrito em RNA mensageiro (RNAm) e então, tal RNAm é traduzido na proteína correspondente. Uma análise de expressão gênica determina os genes que se encontram expressos em uma dada célula.

A justificativa para a análise da expressão gênica segue diretamente da crença de que as atividades das células são determinadas pelos genes que estão expressos. Isso pode ser visto, a partir de uma perspectiva computacional, como se cada gene fosse responsável por realizar certas funções, e colaborar com os outros para realizar tarefas maiores. A coordenação (isto é, quando mandar os genes expressarem-se e realizarem suas tarefas) é realizada de forma "mágica" pelo sistema biológico do organismo. Talvez a importância da expressão gênica possa ser exemplificada através do fato de que muitas doenças são causadas pela expressão incorreta de

[1] Fenótipo é o conjunto de características observáveis de um organismo [31].

certos genes. Portanto, um entendimento de quando e como os genes são expressos é um dos problemas fundamentais da biologia molecular [33, 13].

A abordagem tradicional para pesquisa nesse campo baseia-se em experimentos de escala local, onde apenas alguns genes são examinados e seus relacionamentos determinados. Isso limita o escopo do resultado, uma vez que os genes podem interagir com vários outros genes e suas proteínas, que podem não estar no experimento. Recentemente, um grande número de técnicas experimentais têm sido desenvolvidas com a finalidade de observar a expressão de muitos genes simultaneamente. À frente dessas tecnologias, está o *DNA-microarray* [2]. A principal atração dessas tecnologias é que inúmeros genes podem ser monitorados no mesmo experimento, tornando possível uma análise global da expressão gênica da célula.

Com essas novas tecnologias, o volume de dados de expressão gênica está crescendo substancialmente. Perfis de expressão gênica são medidos, gerando séries de tempo ou tabelas de transição de estados. Muitos métodos computacionais têm sido propostos para analisar tais valores, permitindo a inferência da *rede genética*. Uma rede genética é um conjunto de genes e componentes moleculares que coletivamente realizam uma função celular. O objetivo de tais métodos é inferir a arquitetura de rede funcional, produzindo um modelo de rede que descreve como o nível de expressão de cada gene na rede depende da expressão de outros genes.

Vários modelos formais que capturam as interações e dependências funcionais em redes genéticas têm sido propostos na literatura, como por exemplo, redes booleanas, redes Bayesianas, equações diferenciais, redes de Petri e matrizes de pesos. Cada uma das diferentes abordagens tem suas próprias vantagens e limitações, em termos de considerações, tais como fidelidade ou acurácia do modelo, transparência do modelo (ou equivalentemente, seu valor explicativo), a viabilidade experimental (em termos de requerimentos de dados) da construção do modelo e a tratabilidade computacional da inferência do modelo a partir dos dados [30].

O mais simples dos modelos é a *rede booleana* [25], no qual cada gene pode estar em um estado ligado (expresso) ou desligado (não expresso). De acordo com Silvescu e Honavar [30], uma das maiores vantagens do modelo de rede booleana é que ele é de fácil compreensão, devido à transparência de sua representação. O processo de construção de redes modelo a partir de dados obtidos de redes regulatórias é chamado de *engenharia reversa* de redes genéticas. Apesar de sua extrema simplicidade, existem vários algoritmos para engenharia reversa de redes booleanas, produzindo esboços do comportamento de grandes redes genéticas. Para dois desses trabalhos damos especial atenção.

O trabalho de Liang *et al.* [25] foi um dos primeiros em engenharia reversa de redes booleanas. Os autores desenvolveram um algoritmo, chamado *REVEAL*, que

[2] Mais informações sobre os *DNA-microaarays* podem ser encontradas nos trabalhos de Lander [24], Duggan *et al.* [15], Cheung *et al.* [6], Bowtell [3], Brown [4], Debouck e Goodfellow [8] e Eisen e Brown [16].

faz a inferência de uma rede a partir de seus padrões de atividade. Tal algoritmo utiliza os princípios da *análise da informação mútua* para identificar um conjunto mínimo de entradas que definem unicamente a saída para cada gene no próximo passo.

No trabalho de Ideker *et al.* [20], os autores desenvolveram um algoritmo, chamado *Predictor*, no qual o conhecido problema *hitting set* [18], ou *problema da transversal mínima*, que é NP-difícil, é colocado para que se possa encontrar o conjunto mínimo de entradas que definem o nível de expressão de cada gene no próximo passo. Eles desenvolveram uma estratégia de *perturbações* que pode ser utilizada por cientistas para projeto experimental sistemático [14].

Estudando os dois trabalhos citados acima, concluímos que o *Predictor* é mais simples e elegante para a resolução do problema. Nossa escolha foi então paralelizá-lo. De forma mais objetiva, o problema que o *Predictor* quer resolver é: dados os resultados de um experimento com n genes e m medidas de tempo ($m << n$), consideramos o problema de encontrar um subconjunto de genes (k genes, onde $k << n$) que explica o nível de expressão de um dado gene alvo que desejamos estudar. O número de genes pode ser aproximadamente 10.000, enquanto o número de experimentos pode ser 20, por exemplo. O primeiro obstáculo foi como abordar o problema da transversal mínima, que é NP-difícil, contido no *Predictor*. A maneira que escolhemos para isso foi utilizar um algoritmo de aproximação para o problema. Pesquisando na literatura, encontramos dois algoritmos muito conhecidos. Um deles utiliza o método de aproximação primal-dual, enquanto o outro utiliza uma estratégia gulosa. Baseados no problema da inferência de relacionamentos entre genes que o *Predictor* resolve, entendemos ser o algoritmo guloso, com razão de aproximação $ln|\mathcal{S}|+1$, onde $\mathcal{S}$ é a coleção de conjuntos construída pelo *Predictor* em seu primeiro passo, o que obteria a melhor aproximação.

Desenvolvemos um algoritmo de aproximação seqüencial baseado no *Predictor*, que inclui o algoritmo de aproximação guloso para o problema da transversal mínima. Nosso algoritmo de aproximação possui complexidade de tempo $O(km^2n)$. k é $O(m^2)$, e a complexidade de tempo pode ser expressa apenas em termos de m e n como $O(m^4n)$. O algoritmo tem complexidade de espaço $O(m^2n)$. Devido ao algoritmo guloso para a transversal mínima, nosso algoritmo tem razão de aproximação $ln|\mathcal{S}|+1$, onde novamente $\mathcal{S}$ é a coleção de conjuntos construída pelo *Predictor* em seu primeiro passo. O tamanho de $\mathcal{S}$ pode ser expresso em função de m, o número de linhas da matriz de entrada, como $|\mathcal{S}| = O(m^2)$, lembrando sempre que $m << n$. Tendo desenvolvido esse algoritmo de aproximação seqüencial, o próximo passo foi paralelizá-lo, utilizando o modelo *CGM (Coarse Grained Multicomputer)* [9].

Nosso algoritmo de aproximação CGM para p processadores divide verticalmente a matriz de entrada, sendo que cada processador fica com uma parte de tamanho $\frac{m(n-1)}{p}$ da matriz. O algoritmo tem complexidade de espaço local $O(\frac{m^2n}{p})$, complexidade de tempo $O(\frac{km^2n}{p})$ e $O(k)$ rodadas de comunicação, onde novamente k

é o tamanho da transversal encontrada e $m \times n$ o tamanho da matriz fornecida como entrada. Novamente, como k é $O(m^2)$, podemos afirmar que o algoritmo paralelo tem complexidade de tempo $O(\frac{m^4 n}{p})$ e $O(m^2)$ rodadas de comunicação

Os algoritmos seqüencial e paralelo foram implementados utilizando a linguagem C e a biblioteca de troca de mensagens *MPI* (*Message Passing Interface*). Utilizamos a máquina paralela do tipo *Beowulf*, do projeto temático CAGE, existente no Departamento de Computação do IME/USP. Simulamos matrizes de expressão, através da geração aleatória de 0´s e 1´s. Geramos matrizes de 20 e 40 linhas, simulando o número de medidas de tempo feitas com os genes. Essas matrizes foram dadas como entrada para nossos programas. Os tempos de execução dos programas e as acelerações foram colocadas em gráficos que mostram o comportamento dos programas.

Resultados parciais desta dissertação foram publicados em [28]. Uma versão estendida aparecerá em uma edição especial do periódico internacional *International Journal of High Performance Computing Applications* [29].

Além desta Introdução, este trabalho está dividido em mais 5 capítulos. No Capítulo 2, reforçamos os conhecimentos sobre o problema da determinação de relacionamentos para os genes de uma rede, falando um pouco mais sobre a sua importância e sobre o modelo de rede booleana. Ainda no Capítulo 2, fornecemos alguns importantes conceitos para o desenvolvimento de algoritmos paralelos e também definimos o modelo CGM que utilizamos para o desenvolvimento de nosso algoritmo paralelo. No Capítulo 3, aparecem as descrições dos trabalhos de Liang *et al.* [25], detalhando o algoritmo REVEAL, e de Ideker *et al.* [20], mostrando o algoritmo *Predictor*. No Capítulo 4, expomos o problema da transversal mínima e comentamos as razões de aproximação dos dois algoritmos mais conhecidos para a sua solução. De posse dessas razões de aproximação, apresentamos a justificativa para a escolha do algoritmo guloso, aplicando a sua razão de aproximação ao problema de inferência de relacionamentos entre os genes. No Capítulo 4 também detalhamos o algoritmo de aproximação seqüencial que desenvolvemos, juntamente com suas estruturas de dados, e comentamos sobre sua complexidades de tempo e de espaço. Ainda no Capítulo 4, descrevemos o algoritmo CGM desenvolvido, juntamente com seus custos de tempo, espaço e comunicação. No Capítulo 5, mostramos os resultados experimentais obtidos para matrizes de entrada de 513, 1025, 2049, 4097 e 8193 colunas e 20 e 40 linhas, mostrando para cada uma das execuções, suas curvas de tempo de execução por número de processadores e também suas curvas de aceleração. No Capítulo 6, apresentamos a conclusão deste trabalho.

Capítulo 2

Preliminares

Desde que a estrutura do DNA foi revelada, em 1953, por Watson e Crick, a biologia molecular tem alcançado grandes avanços. Com o crescimento das tecnologias de manipulação de seqüências biomoleculares, uma enorme quantidade de dados foi e está sendo gerada. A necessidade de processar essas informações cria problemas novos e interdisciplinares. Devido ao tamanho e complexidade desses problemas, os biólogos necessitam da ajuda de outras disciplinas, particularmente daquelas das ciências matemáticas e computacionais. Essa necessidade criou um novo campo, que ganhou o nome de *biologia molecular computacional*, ou simplesmente *biologia computacional*. De forma geral, biologia computacional consiste no desenvolvimento e uso de técnicas matemáticas e computacionais para ajudar a resolver problemas da biologia molecular [26].

A biologia computacional está rapidamente emergindo como uma das principais áreas de pesquisa, com aplicações de incomparável significado. Os algoritmos para resolver problemas biológicos estão quase sempre associados a longos tempos de execução. Como bem observou Aluru [1], isso se deve a vários fatores:

1. Dados biológicos são obtidos por experimentos propensos a erros. A necessidade de tratar tais erros e incertezas resulta em algoritmos com grandes complexidades;
2. A quantidade de dados pode ser muito grande e resultar em longos tempos de execução;
3. Muitos dos problemas pertencem à classe NP.

Dado o crescimento exponencial das descobertas científicas na área biológica, podemos esperar que a utilização efetiva de computadores paralelos torne-se cada vez mais importante para a biologia computacional.

Achamos por bem paralelizar algum algoritmo que atendesse a uma necessidade relevante da biologia. Deparamo-nos então com o problema de *inferência de redes genéticas*, ou seja, inferência de relacionamentos entre os genes a partir de dados experimentais. Com técnicas modernas, tais como os *microarrays*, grandes quantidades de genes, da ordem de 10.000, são analisados em um único experimento. Portanto, buscamos algoritmos que resolvam esse problema, e que possam necessitar de uma paralelização. Neste capítulo, daremos uma visão geral sobre o processamento paralelo e o problema biológico que escolhemos para estudar.

2.1 Processamento Paralelo

Um *computador paralelo* é uma coleção de processadores, tipicamente do mesmo tipo, interconectados de uma certa maneira que permita a coordenação de suas atividades e a troca de dados [21]. Esses processadores trabalham ao mesmo tempo e de forma coordenada para resolver um problema específico. O principal objetivo do processamento paralelo é utilizar vários processadores para resolver problemas de forma mais rápida do que seriam resolvidos por apenas um processador.

Nesta seção, discutimos mais detalhadamente os modelos de computação paralela e outros conceitos básicos do processamento paralelo.

2.1.1 Níveis de Paralelismo

Na computação paralela, uma tarefa é dividida em processos que são executados simultaneamente por diferentes processadores. O *nível de paralelismo*, também conhecido como granularidade, está relacionado com o tamanho dos processos executados pelos processadores. Podemos falar em três tipos de granularidade, a saber:

- Granularidade grossa: quando cada processo contém um grande número de instruções seqüenciais e gasta um tempo considerável para executá-las.
- Granularidade fina: quando o processo consiste em poucas instruções, ou até mesmo uma única instrução.
- Granularidade média: representa um grupo intermediário.

A granularidade não está envolvida diretamente com a otimização de algoritmos paralelos, porém as tendências atuais incentivam o uso de algoritmos com granularidade grossa, no intuito de minimizar a comunicação e aumentar a eficiência [27].

2.1.2 Avaliação de Algoritmos Paralelos: Custo, *Speedup* ou Aceleração e Eficiência

Um computador paralelo depende de um grande número de parâmetros, tais como o número de processadores, as capacidades das memórias locais de cada processador, o esquema de comunicação e os protocolos de sincronização. A preocupação com esse grande número de parâmetros faz com que o projeto, avaliação e análise de algoritmos paralelos sejam mais complexos do que no modelo seqüencial. Falaremos a princípio de medidas gerais, comumente utilizadas para avaliar o desempenho de um algoritmo paralelo.

Seja P uma dado problema computacional e seja n o tamanho da entrada. Seja $T^*(n)$ a complexidade seqüencial de P. Isso significa que existe um algoritmo seqüencial que resolve P com esse tempo, e ainda mais, podemos provar que nenhum algoritmo seqüencial pode resolver P de forma mais rápida. Seja A um algoritmo paralelo que resolve P em um tempo $T_p(n)$ em um computador paralelo com p processadores.

Definimos o *custo* do algoritmo A como:

$$C_p(n) = T_p(n) \times p$$

Definimos o *speedup* ou *aceleração* alcançado por A como:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

No decorrer do trabalho, será usado apenas o termo aceleração. O valor $S_p(n)$ mede a aceleração obtida por um algoritmo A quando p processadores estão disponíveis para utilização. Como temos $S_p(n) \leqslant p$, o objetivo quando se projeta um algoritmo paralelo é que ele alcance $S_p \backsimeq p$.

Note que $T_1(n)$, o tempo do algoritmo paralelo A quando o número de processadores é igual a 1, não é necessariamente igual a $T^*(n)$, já que a aceleração é obtida relativamente ao melhor algoritmo seqüencial possível. Uma outra medida de execução do algoritmo paralelo A é a *eficiência*, definida por:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

Essa medida indica a utilização efetiva de p processadores relativa ao algoritmo dado. Se E_p for aproximadamente 1, para algum p, então o algoritmo A executa aproximadamente p vezes mais rápido usando p processadores do que executaria utilizando apenas 1.

Vimos nesta subseção algumas formas de avaliar a execução de algoritmos paralelos. Necessitamos definir um modelo, que considere as diferentes características presentes em um sistema de computação paralela, para que possamos analisar a complexidade de algoritmos paralelos.

2.1.3 Modelos Realísticos de Computação Paralela

O modelo *RAM*, comumente aceito para o projeto e análise de algoritmos seqüenciais, consiste em uma unidade central de processamento, com uma memória de acesso aleatório atrelada a ela. O conjunto de intruções típico para esse modelo inclui leitura e escrita na memória e operações lógicas e aritméticas básicas. O sucesso desse modelo é devido à sua simplicidade e capacidade de capturar o desempenho de algoritmos seqüenciais em computadores do tipo Von Neumann. A função de um modelo é, então, descrever e analisar algoritmos paralelos.

Segundo Jájá [21], o modelo paralelo ideal deveria ter as seguintes características:

- Simplicidade: o modelo deveria ser simples o suficiente para descrever algoritmos paralelos facilmente e para analisar matematicamente medidas de desempenho importantes, tais como velocidade, comunicação e utilização da memória. O modelo também não deveria ser vinculado a nenhuma classe particular de arquiteturas e, portanto, deveria ser o mais independente de hardware possível;
- Facilidade de Implementação: os algoritmos paralelos deveriam ser facilmente implementáveis nos computadores paralelos. Além disso, a análise realizada deveria ser capaz de capturar o desempenho real desses algoritmos em computadores paralelos.

O processamento paralelo sofre a falta de um modelo que seja amplamente aceito. Ao final dos anos 80, vários problemas haviam sido estudados e vários limites inferiores e superiores foram demonstrados para esses problemas em diferentes modelos de computação, tais como memória compartilhada, hipercubos e *meshs*. Quando esses resultados teóricos eram implementados nas máquinas existentes, as acelerações obtidas eram, muitas vezes, desapontadoras. Esses resultados levaram à criação de modelos de computação paralela com granularidade grossa. Nesses modelos, o conceito de computação paralela é representado com uma série de *superpassos*, ao invés de passos com o envio individual de mensagens ou acessos individuais à memória compartilhada. O *BSP* e o *CGM* são exemplos desses modelos. Os algoritmos projetados para esses modelos, quando implementados nas máquinas existentes, têm obtido acelerações próximas das previstas em resultados teóricos. Isso faz com que sejam chamados de modelos realísticos [9].

O Modelo BSP (*Bulk-Synchronous Parallel*)

O modelo BSP foi proposto por Valiant [34] em 1990 e é um dos modelos realísticos mais importantes. Além disso, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros. O modelo seqüencial de Von Neumann, já citado anteriormente, serve de ponte entre as necessidades de hardware e software da computação seqüencial, sendo essa a razão do seu sucesso. O principal objetivo do modelo BSP é ser essa ponte para a computação paralela e, conseqüentemente, servir como um padrão.

O modelo BSP consiste na combinação de três atributos, quais sejam, p processadores com memória local, um roteador que entrega mensagens ponto-a-ponto entre pares de processadores e facilidade de sincronização de todos ou apenas um subconjunto de processadores.

Um algoritmo BSP consiste em uma seqüência de *superpassos* separados por *barreiras de sincronização*, como mostra a Figura 2.1.

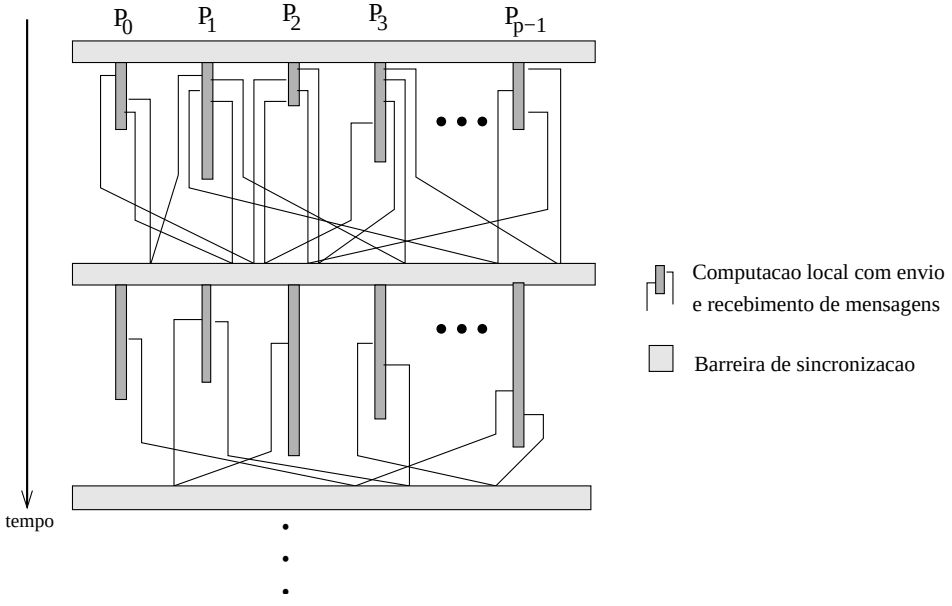


Figura 2.1: Algoritmo BSP.

.

Em cada superpasso, a cada processador é dada uma tarefa que consiste em uma combinação de passos de computação local e transmissões de mensagens. Implicitamente, consideramos também as mensagens que chegam de outros processadores. As tarefas de computação e comunicação podem ser separadas em superpassos de computação e superpassos de comunicação. Depois de cada período de L unidades de tempo, uma verificação global, através de uma barreira de sincronização, é feita para determinar se o superpasso já foi completado por todos os processadores. Em caso afirmativo, a máquina continua com o próximo superpasso. Caso contrário, outro período de L unidades é alocado para terminar o superpasso. Neste modelo,

uma h-relação em um superpasso corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador e a resposta a uma mensagem enviada em um superpasso somente será usada no próximo superpasso.

O modelo possui os seguintes parâmetros:

- n: tamanho do problema;
- p: número de processadores disponíveis, cada um com sua memória local;
- L: tempo mínimo de um superpasso;
- g: descreve a taxa de eficiência de computação e comunicação, que corresponde à razão entre o número total de operações de computação local de todos os processadores em uma unidade de tempo, e o número total de mensagens enviadas/recebidas em uma unidade de tempo.

Os parâmetros L e g são utilizados para computar o custo de comunicação de um algoritmo BSP. O parâmetro L é o custo de sincronização, de forma que cada operação de sincronização contribui com L unidades de tempo para o tempo total de execução. O parâmetro g está relacionado com a capacidade de comunicação de uma rede de computadores. Se o número máximo de mensagens enviadas por algum processador durante uma troca simples é h, então seriam necessárias até gh unidades de tempo para a conclusão da troca [5]. Cada superpasso de um algoritmo BSP possui tempo total de execução $w_i + gh_i + L$, onde $w_i = max\{L; t_1; ...; t_p\}$, sendo que t_j é o tempo das operações de computação executadas pelo processador j no superpasso i e $h_i = max\{L; c_1; ...; c_p\}$, onde c_j é tempo das mensagens recebidas e/ou enviadas pelo processador j no superpasso i. Seja T o número de superpassos, teremos:

- Custo total de computações locais: $W = \sum_{i=0}^{T} w_i$.
- Custo total de comunicação: $H = \sum_{i=0}^{T} h_i$.

De tudo isso, o custo total de um algoritmo BSP é dado por $W + gH + LT$.

O Modelo CGM (*Coarse Grained Multicomputer*)

O modelo CGM foi proposto por Dehne [15] e consiste em um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{n}{p})$, onde n é o tamanho do problema. Um algoritmo CGM consiste em uma seqüência alternada de *rodadas de computação* e *rodadas de comunicação*, separadas por uma barreira de sincronização, como mostra a Figura 2.2. Normalmente, durante uma rodada de computação é utilizado o melhor algoritmo seqüencial para o processamento dos dados disponibilizados localmente. Um algoritmo CGM é um caso especial de algoritmo BSP, onde

uma rodada de computação é equivalente a um superpasso de computação no BSP, e o custo total de computação é definido analogamente. Uma rodada de comunicação consiste em uma única h-relação, com $h \leqslant \frac{n}{p}$. Cada processador envia $O(\frac{n}{p})$ dados e recebe $O(\frac{n}{p})$ dados. Toda informação mandada de um processador para outro em uma rodada de comunicação é empacotada em uma grande mensagem, visando com isso reduzir a sobrecarga (*overhead*) para envio de mensagem. O custo das rodadas de comunicação é $H_{n,p} = O(\frac{n}{p})$. Portanto, o custo total de comunicação de um algoritmo CGM com x rodadas de comunicação é $x \times H_{n,p}$.

A principal diferença entre os modelos BSP e CGM é que o CGM permite apenas um único tipo de operação de comunicação, a h-relação, e apenas conta o número de h-relações como sua principal medida de custo de comunicação. Uma rodada CGM de computação/comunicação corresponde a um superpasso do BSP com custo de comunicação $g(\frac{n}{p})$ (mais o empacotamento) [10].

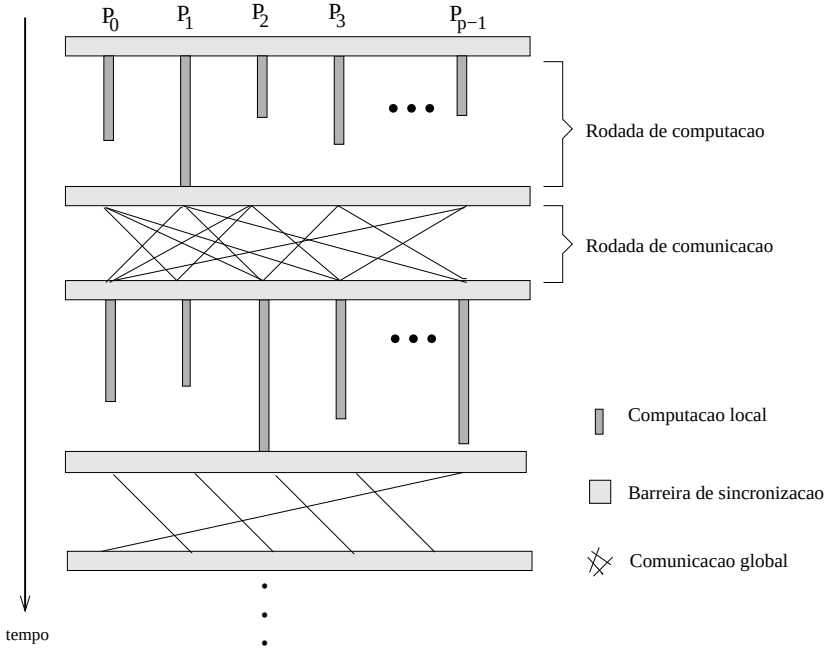


Figura 2.2: Algoritmo CGM.

O custo de um algoritmo CGM é a soma dos tempos obtidos em termos do número total de rodadas de computação local e o número total de rodadas de comunicação.

Os algoritmos CGM, quando implementados em multiprocessadores atualmente disponíveis, comportam-se bem e exibem acelerações similares àquelas previstas em suas análises. Para esses algoritmos, o objetivo é minimizar o número de superpassos e a quantidade de computação local.

2.2 Inferência de Redes Genéticas

A biologia entrou em uma era onde os genomas de vários organismos já foram ou estão prestes a ser completamente seqüenciados. No entanto, existe ainda um longo caminho a ser percorrido da seqüência de DNA até o entendimento funcional do organismo correspondente. Segundo Thieffry e Thomas [32], mesmo no caso da *E. coli*, um organismo bem caracterizado, o seqüenciamento completo do seu DNA nos deixa com várias perguntas em aberto, relativas às funções dos genes, mecanismos regulatórios, ou integração global.

Após o seqüenciamento do genoma, muitas análises de larga escala iniciaram-se, com o objetivo de descobrir a organização funcional das células. Nesse campo, um grande desafio é determinar *redes genéticas regulatórias*. Todo gene tem um ou mais ativadores, sinais bioquímicos que são necessários para iniciar a transcrição desse gene. Os genes também possuem inibidores, que são sinais bioquímicos que previnem a expressão de um gene particular, mesmo na presença de um ativador apropriado. Apenas um pequeno número de genes funcionam como ativadores ou inibidores, mas identificá-los é um problema importante e difícil. Uma rede genética regulatória define a complicada estrutura de genes que ativam/desativam outros genes. Identificar redes genéticas regulatórias a partir de dados experimentais é atualmente uma área de pesquisa extremamente ativa.

Nesta seção reforçamos o entendimento da inferência de redes genéticas a partir de dados experimentais e apresentamos o modelo de rede booleana.

2.2.1 O Problema da Identificação de Redes Genéticas a partir de Dados Experimentais

Segundo a definição de Ideker *et al.* [20], uma rede genética pode ser descrita como uma coleção de componentes moleculares, como genes, e interações entre eles, que coletivamente realizam uma função celular. Para entendermos as funções dos genes nos processos biológicos é preciso, então, que entendamos os princípios do comportamento de tais redes. Como uma rede genética pode ser muito complexa, uma maneira de obter progressos no entendimento de seu comportamento é simplificar as interações moleculares individuais e concentrar-se no efeito coletivo.

Padrões de expressão gênica refletem o estado da rede em um instante de tempo particular. Devido ao recente progresso de tecnologias, como os *DNA-microarrays*, é possível medir os níveis de expressão de um grande número de genes de um organismo simultaneamente.

Com o grande volume de dados de expressão gênica disponíveis, desejam-se métodos para analisar as medidas de expressão gênica de séries de tempo ou tabelas

de transição de estados, para inferir a rede genética que está por trás desses dados. Na inferência de rede, o objetivo é construir um modelo da rede de interações regulatórias entre os genes. Isso requer a *engenharia reversa* de relacionamentos causais entre os genes a partir de seus padrões de atividades. Conforme afirma D'haeseleer *et al.* [14], os resultados desse tipo de engenharia reversa apenas dizem respeito à estrutura da rede como um todo. Enquanto isso irá esclarecer pouco sobre os mecanismos moleculares envolvidos, muitas informações úteis serão adquiridas sobre os genes críticos para um processo biológico. Tais informações serão suficientes, por exemplo, para a identificação de drogas que combatam o processo, caso esse desencadeie doenças.

Devido à complexidade das redes genéticas, utilizamos os modelos para fazer uma estimativa dessa rede no organismo de interesse. Vários tipos de modelos têm sido propostos e o escolhido é sempre determinado pela questão que o pesquisador está tentando responder. Existem muitas particularidades a serem consideradas na modelagem embora, muitas vezes, simplificações tenham que ser feitas. Por exemplo, embora modelos estocásticos mostrem dinâmicas mais realistas, os modelos de redes genéticas são, geralmente, determinísticos. A razão para essa simplificação é a dificuldade de se inferir uma rede básica, se os padrões de expressão são resultados de um processo estocástico [15].

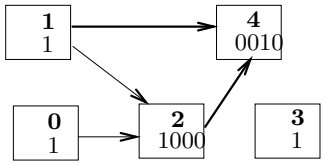
Os métodos de inferência estudados no presente trabalho dizem respeito à engenharia reversa de redes genéticas utilizando o modelo de rede booleana, que é o mais simples dentre os modelos conhecidos.

2.2.2 O Modelo de Rede Booleana

Baseados em Ideker *et al.* [20], podemos dizer que uma rede booleana é representada como um grafo consistindo de N nós, que representam genes, rotulados a_n $(0 \leqslant n < N)$, um conjunto de arestas orientadas (flechas) entre os nós e uma função f_n para cada nó.

Um nó tem um nível de expressão associado x_n, que representa a quantidade de produto de tal gene presente na célula. Este nível é aproximado para "alto" ou "baixo" e é representado pelos valores binários 1 ou 0, respectivamente. O nível de expressão é 1 se o gene está expresso e 0, caso contrário. Uma aresta orientada de um nó para outro representa a influência do primeiro gene sobre o segundo, de tal forma que o nível de expressão de um nó a_n é uma função booleana f_n dos níveis dos nós na rede que se conectam (têm uma aresta orientada) com a_n. Essa função possibilita que cada nó determine seu próximo estado a partir dos estados correntes de todos os seus nós de entrada. Nós de entrada de a_n são os nós que têm uma aresta orientada para a_n.

Um exemplo de uma pequena rede booleana é mostrado na Figura 2.3.



(a)

x_1	1	0	1	0
x_2	1	1	0	0
x_4	**0**	**0**	**1**	**0**

(b)

$x_0 := 1$
$x_1 := 1$
$x_2 := x_0$ and x_1
$x_3 := 1$
$x_4 := x_1$ and not x_2

(c)

Figura 2.3: Representação de uma rede booleana.

Na Figura 2.3(a), cada nó a_n é representado por uma caixa, com o número do nó, n, colocado na metade superior da caixa. Na metade inferior da caixa está um código numérico representando a função f_n associada com o nó. Esse código é extraído da linha de saída de uma tabela-verdade. Uma tabela-verdade é exemplificada na Figura 2.3(b), para o nó a_4. Uma tabela-verdade nos dá uma descrição única da função booleana para um nó, e o código (mostrado em negrito) representa o nível de saída x_n, correspondente a cada combinação possível de níveis de entrada. No exemplo, o nível do gene a_4 é determinado pelos níveis dos genes a_1 e a_2, de acordo com a função associada, representada pelo código binário 0010. O nível de expressão de estado fixo x_4 será alto apenas quando x_1 for alto e x_2 for baixo. As equações da Figura 2.3(c) mostram uma representação alternativa da topologia da rede e suas funções.

No modelo booleano, uma suposição adicional é que a rede é síncrona, ou seja, todos os genes mudam seu estado ao mesmo tempo. Esse modelo é também claramente determinístico, já que o mesmo estado final é encontrado, dados estados iniciais idênticos.

Segundo Dutilh [15], comparadas com redes genéticas, redes booleanas são simplificações grosseiras. A expressão gênica nunca é um caso de tudo ou nada (1 ou 0) mas, como bem pondera D'haeseleer [12], apesar de sua extrema simplicidade, redes booleanas têm provado-se útil para o desenvolvimento de esboços do comportamento de grandes redes genéticas.

Capítulo 3

Trabalhos sobre Inferência de Redes Booleanas

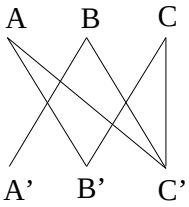
Muitos algoritmos foram e têm sido propostos para inferência de redes booleanas. Nossa pesquisa a esse respeito iniciou-se por um dos primeiros e talvez mais citado trabalho a esse respeito denominado REVEAL, de Liang *et al.* [25]. Nesse trabalho, os autores desenvolveram um algoritmo utilizando os *princípios da informação mútua* para identificar um conjunto mínimo de entradas que unicamente definem a saída para cada gene no próximo passo. Segundo Tsang [33], o REVEAL é viável apenas para séries de tempo pequenas, pois o contrário faz com que o algoritmo leve muito tempo para realizar os cálculos da *entropia* e da informação mútua, necessários para a resolução do problema. Além disso, o algoritmo supõe que o número de genes que influenciam um outro é pequeno (de 2 a 3), caso contrário a estratégia exaustiva do algoritmo seria inviável. Buscando um algoritmo mais simples e sem essas restrições, estudamos o *Predictor* proposto por Ideker *et al.* [20]. Eles desenvolveram um algoritmo no qual o problema *hitting set* [18], ou *problema da transversal mínima*, precisa ser resolvido para encontrar as dependências de um gene em estudo. Eles também desenvolveram uma estratégia de *perturbações* para obtenção dos dados. Neste capítulo, apresentamos os dois métodos estudados.

3.1 Algoritmo REVEAL

Esta seção está totalmente baseada no trabalho de Liang *et al.* [25].

REVEAL (*A General Reverse Engineering Algorithm for Inference of Genetic Network Architectures*) é um algoritmo que analisa tabelas de transição de estados de perfis de expressão gênica para inferir a rede genética que está por trás desses perfis. A Figura 3.1 mostra uma outra forma de representação da rede booleana, com sua tabela-verdade completa. Os exemplos desta seção utilizarão essa pequena rede.

Podemos ver na Figura 3.1(c) uma tabela completa de transições de estado definindo a rede. As colunas de entrada correspondem ao estado no tempo t, enquanto as colunas de saída (elementos marcados com apóstrofo) correspondem ao estado no tempo $t + 1$.



(a) Diagrama de ligações.

A' = B

B' = A or C

C' = (A and B) or (B and C) or (A and C)

(b) Regras lógicas(booleanas).

entrada			saida		
A	**B**	**C**	**A'**	**B'**	**C'**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(c) Tabela de transições de estado definindo a rede.

Figura 3.1: Outra representação de uma rede booleana.

3.1.1 A Informação Quantificada: Entropia de Shannon (H)

Na teoria da informação existe uma medida quantitativa de informação, a *entropia de Shannon* (H). A entropia de Shannon é definida em termos da probabilidade de se observar um símbolo particular ou evento p_i dentro de uma dada seqüência:

$$H = -\sum p_i \log p_i.$$

Em um sistema binário, um elemento X pode estar em $s = 2$ estados, *ligado* ou *desligado*. As Figuras 3.2 e 3.3 nos dão uma idéia do comportamento de H. Sobre uma seqüência particular de eventos (Figura 3.2(a)), a soma das probabilidades de X estar *ligado*, $p(1)$, ou *desligado*, $p(0)$, tem que ser igual a 1.

A entropia de Shannon H alcança seu máximo quando seus estados *ligado* e *desligado* são equiprováveis, ou seja, possuem a mesma probabilidade de acontecer (Figura 3.3). À medida que um estado torna-se mais provável do que o outro, H decresce. No caso limite, quando uma probabilidade é 1 (certeza) e a outra é zero (impossibilidade), H é zero (nenhuma incerteza, nenhuma liberdade de escolha, nenhuma informação). A entropia máxima, H_{max}, ocorre quando todos os estados são equiprováveis, isto é $p(0) = p(1) = 1/2$. Então, $H_{max} = \log(2)$. Entropias são comumente medidas em bits, quando usamos o logaritmo na base 2; por exemplo, $H_{max} = 1$, para um sistema de 2 estados.

X	0	1	1	1	1	1	1	0	0	0
Y	0	0	0	1	1	0	0	1	1	1

$H(X) = -0.4\log(0.4) - 0.6\log(0.6) = 0.97$
(40% $0s$ and 60% $1s$)

$H(Y) = -0.5\log(0.5) - 0.5\log(0.5) = 1.00$
(50% $0s$ and 50% $1s$)

(a) Elementos isolados.

Y \ X	0	1
1	3	2
0	1	4

$H(X,Y) = -0.1\log(0.1) - 0.4\log(0.4)$
$-0.3\log(0.3) - 0.2\log(0.2) = 1.85$

(b) Distribuição de pares de valores.

Figura 3.2: Determinação de H.

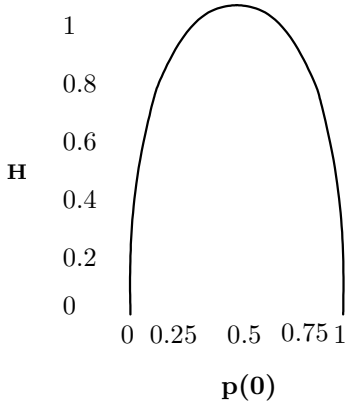


Figura 3.3: Entropias de Shannon para uma origem de informação de 2 estados. Como a soma das probabilidades dos estados tem que ser 1, $p(1) = 1 - p(0)$ para 2 estados.

Nosso objetivo é comparar seqüências diferentes usando medidas de informação para estabelecer relacionamentos funcionais entre elementos de uma rede. Em um sistema de 2 elementos binários, digamos X (índice i) e Y (índice j), as entropias de Shannon individuais e combinada são definidas essencialmente como abaixo (Figura 3.2(b))):

$$H(X) = -\sum p_i \log p_i$$

$$H(Y) = -\sum p_j \log p_j$$

$$H(X,Y) = -\sum p_{i,j} \log p_{i,j}$$

Existem duas entropias condicionais que capturam o relacionamento entre as seqüências de X e Y. São elas $H(X|Y)$ e $H(Y|X)$ e relacionam-se como segue:

$$H(X,Y) = H(Y|X) + H(X) = H(X|Y) + H(Y).$$

Em palavras, a incerteza de X e a incerteza restante de Y tendo conhecimento de X, $H(Y|X)$ (a informação contida em Y que não é compartilhada com X), somam a entropia da combinação de X e Y.

3.1.2 Informação Mútua (M): a Informação (Entropia de Shannon) Compartilhada por Elementos Não-Independentes

Após a introdução dos conceitos acima, podemos encontrar uma expressão para a *informação mútua* compartilhada, também chamada de *taxa de transmissão* entre um par de canais entrada/saída:

$$M(X,Y) = H(Y) - H(Y|X) = H(X) - H(X|Y).$$

A informação compartilhada entre X e Y corresponde à informação restante de X, se removermos a informação de X que não é compartilhada com Y. Usando as equações acima, a informação mútua pode ser definida diretamente em termos das entropias originais:

$$M(X,Y) = H(X) + H(Y) - H(X,Y).$$

Os diagramas de *Venn* da Figura 3.4 ilustram os relacionamentos entre tais medidas. Esses princípios de informação serão usados para extrair as conexões críticas entre elementos de rede, a partir de dados de transição de estado de redes binárias.

3.1.3 O Núcleo do REVEAL: M-Análise Sistemática de Tabelas de Transição de Estado

A estratégia do algoritmo é utilizar as medidas de informação mútua para extrair os relacionamentos entre os genes a partir de tabelas de transição de estados.

Nesta seção, mostraremos o algoritmo, juntamente com sua execução passo a passo, na análise da rede-exemplo da Figura 3.1.

Podemos ver na Figura 3.5 um exemplo da execução do algoritmo para a rede exemplo da Figura 3.1.

Algoritmo 1 Reveal

(1) Identificação de k (=1) ligações.
Começamos por determinar as entropias e informação mútua de todos os pares entrada-saída únicos. No caso do exemplo, determinamos $M(A', A)$, $M(A', B)$, $M(A', C)$, $M(B', A)$, $M(B', B)$, $M(B', C)$, $M(C', A)$, $M(C', B)$ e $M(C', C)$. O símbolo " ' " denota o estado de saída de um elemento, por exemplo, A'. Se $M(A', X) = H(A')$, isto é, $M(A', X)/H(A') = 1$, então X determina A'. Esse é o caso de B e A' na Figura 3.5.

(2) Determinação da regra k (=1).
A tabela inicial (Figura 3.1) para os pares entrada-saída constitui a regra.

(3) Identificação de k (=2) ligações.
Se nem todas as transições de estado podem ser explicadas em termos de $k = 1$, determinaremos entropias de combinações de pares de entradas com o restante dos elementos de saída não resolvidos. Se $M(B', [X, Y]) = H(B')$ ou, mais concisamente, $H(B', X, Y) = H(X, Y)$, então, o par $[X, Y]$ determina completamente B'. Na Figura 3.5, nenhuma entrada única pode determinar B', mas o par $[A, C]$ responde por toda a entropia de B'.

(4) Determinação da regra k (=2). (como acima para $k = 1$)

(5) Identificação de k (=3) ligações.
Se nem todos os elementos podem ser resolvidos em termos de $k = 1$ e $k = 2$, o próximo passo é determinar as entropias de combinações de triplas de entradas, com os elementos de saída restantes. Em nosso exemplo, como $M(C', [A, B, C]) = H(C')$, ou, mais concisamente, $H(C', A, B, C) = H(A, B, C)$, então $[A, B, C]$ determina completamente C'.

(6) Determinação da regra $k = 3$. (como acima para $k = 1$)

(i) Identificação de k ($= i$, $i < n$, onde n é o número de nós da rede) ligações.
Isso se aplica às redes de qualquer tamanho. Se nem todas os elementos podem ser explicados em termos de $k = i - 1, i - 2, ..., 1$ entradas, a busca procura pelas entropias de combinações de i-uplas entradas com o restante dos elementos de saída não resolvidos. Se $M(Y, [X_1, X_2, X_3, ...X_i]) = H(Y)$, ou $H(Y, X_1, X_2, X_3, ...X_i) = H(X_1, X_2, X_3, ...X_i)$, então a i-upla $[X_1, X_2, X_3, ...X_i]$ determina completamente Y (Y=elemento de saída). Naturalmente, as combinações dos valores de entrada das tabelas de transição de estado cobrindo Y e $[X_1, X_2, X_3, ...X_i]$ definem a tabela de busca da regra.

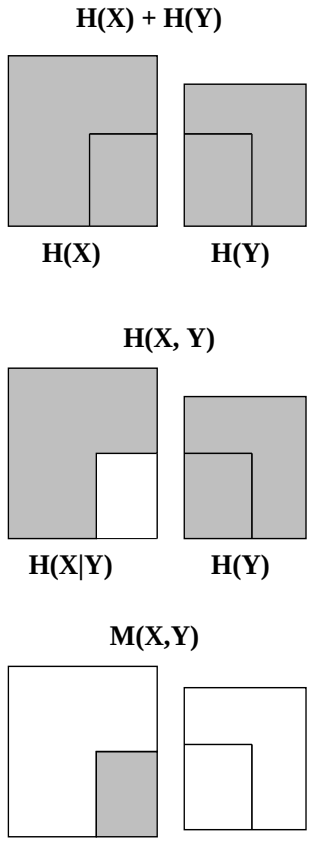


Figura 3.4: Diagramas de Venn. Os retângulos pequenos nos cantos representam a informação que X e Y têm em comum.

3.1.4 Avaliação do REVEAL

Observando os passos do REVEAL, podemos afirmar que sua complexidade é exponencial.

3.2 *Algoritmo Predictor*

Esta seção está totalmente baseada no trabalho de Ideker *et al.* [20].

O *Predictor* é um algoritmo que analisa séries de tempo de perfis de expressão gênica para inferir a rede genética que está por trás desses perfis. Para medir essas séries de tempo é utilizada uma estratégia de *perturbações*.

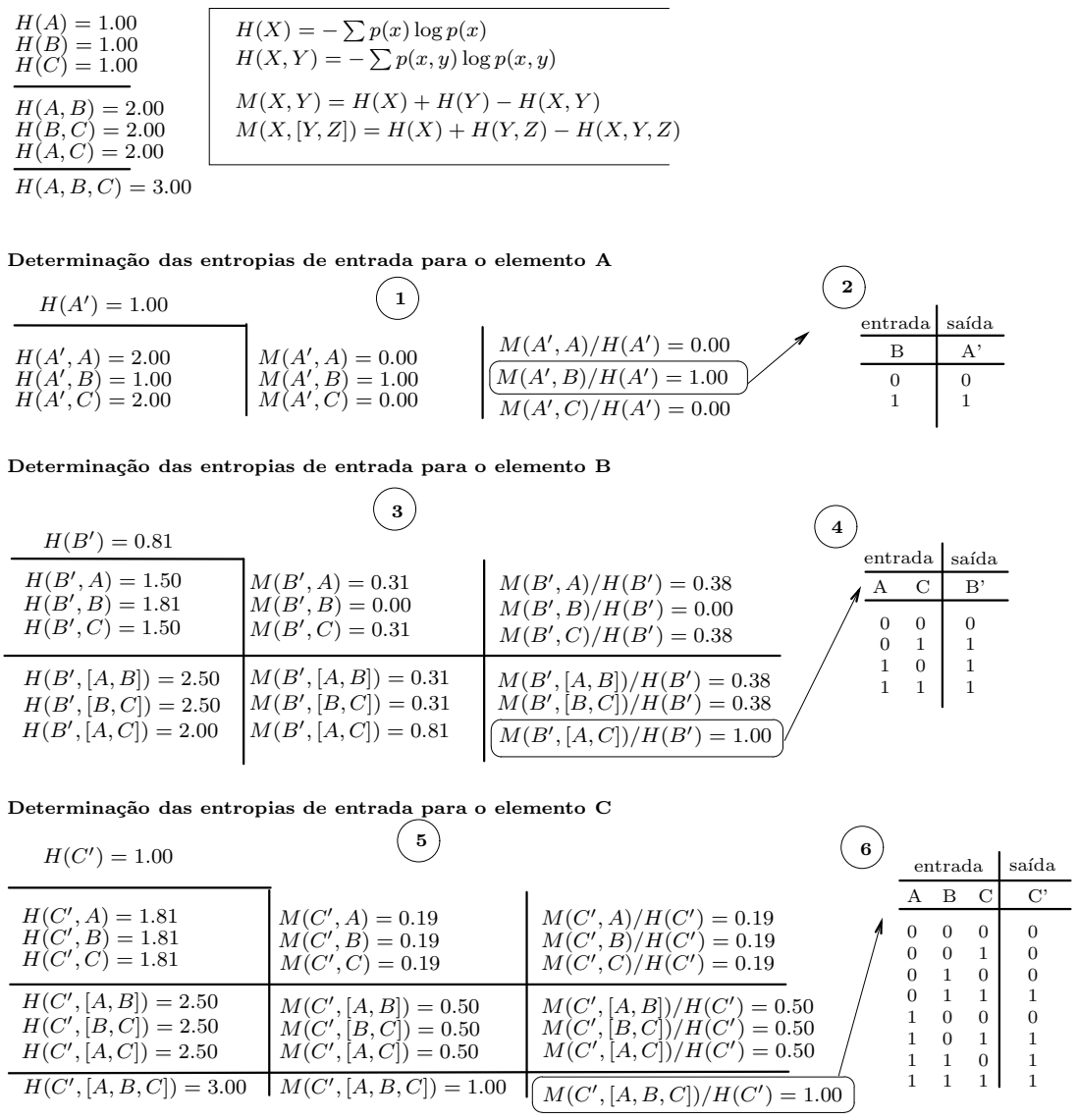


Figura 3.5: Visão geral da M-análise progressiva usando o REVEAL para a rede exemplo da Figura 3.1.

3.2.1 Estratégia de Perturbações

Informações adicionais sobre a rede genética podem ser obtidas experimentalmente aplicando-se uma *perturbação* à rede e observando os níveis de expressão de estado fixo de todos os genes na rede na presença da perturbação. Perturbações podem ser *genéticas*, nas quais os níveis de expressão de um ou mais genes são fixados por *delection* ou *overexpression*, ou *biológicas*, nas quais um ou mais fatores não genéticos são alterados, tal como uma mudança na temperatura.

De acordo com a estratégia de Ideker *et al.* [20], a rede genética de interesse é exposta a uma série de perturbações genéticas e/ou biológicas e um perfil de expressão gênica de estado fixo é obtido para cada uma das perturbações. O próximo passo é utilizar esses tais perfis como entrada para um algoritmo, denominado *Predictor*, que irá inferir uma ou mais redes booleanas hipotéticas, consistentes com esses perfis.

O método apresentado utiliza o modelo de rede booleana para apresentar um esboço da rede genética real estudada. Para inferir uma rede genética do tipo booleana, uma população de células contendo uma rede genética alvo T é monitorada no estado fixo sobre uma série de P perturbações experimentais. Em cada perturbação p_m $(0 \leqslant m < P)$, qualquer número de nós pode ser forçado a um nível *alto*(1) ou *baixo*(0). Genes podem ser perturbados através de métodos laboratoriais para *deletion* e *overexpression* de genes. Os níveis de expressão de estado fixo observados para todos os genes sobre todas as perturbações são representados pela matriz de expressão M. A Figura 3.6 mostra uma matriz de expressão gerada por perturbações ilustrativas aplicadas à rede genética da Figura 2.3. Linhas de M representam condições de perturbação, enquanto colunas representam os valores dos nós em cada condição de estado fixo, de tal forma que a entrada da matriz M_{mn} é o nível de expressão do nó a_n na presença da perturbação p_m. Os símbolos + e − são usados para mostrar que um nó foi forçado a um valor alto ou baixo, respectivamente.

$$M = \left|\begin{array}{ccccc:c} x_0 & x_1 & x_2 & x_3 & x_4 & \\ 1 & 1 & 1 & 0 & 0 & p_0 \\ - & 1 & 0 & 0 & 1 & p_1 \\ 1 & - & 0 & 0 & 0 & p_2 \\ 1 & 1 & - & 0 & 1 & p_3 \\ 1 & 1 & 1 & 0 & + & p_4 \end{array}\right|$$

Figura 3.6: Matriz exemplo gerada a partir da rede genética da Figura 2.3.

3.2.2 O Algoritmo

Vamos agora descrever o algoritmo *Predictor*, um algoritmo para inferência de redes booleanas utilizando os dados de expressão contidos na matriz M. Para chegar a uma função booleana f_n independente para cada nó a_n, nós determinamos um conjunto mínimo de nós, cujos níveis precisam ser incluídos como variáveis de entrada de f_n, para explicar os dados observados em M, e então construir uma tabela-verdade usando esses nós como entradas. O passo de determinar a entrada

para cada nó é equivalente a determinar de que outros nós cada um deles depende. As entradas para cada nó a_n e a tabela-verdade são determinadas como segue:

Algoritmo 2 *Predictor*

(1) Considere todos os pares de linhas (i,j) de M nos quais o nível de expressão de a_n difere, excluindo as linhas onde a_n foi forçado a um valor *alto* ou *baixo*. Para cada par, encontre o conjunto S_{ij} de todos os outros nós cujo nível de expressão também difere entre as duas linhas (i,j). Devido ao fato de a rede ser auto-contida, uma mudança em pelo menos um desses genes precisa ter causado a diferença correspondente em a_n. Portanto, no mínimo um nó nesse conjunto precisa ser incluído como uma variável de f_n.

(2) Identifique o menor conjunto de nós S_{min} necessário para explicar as diferenças observadas sobre todos os pares de linhas (i,j), de tal forma que pelo menos um nó em S_{min} esteja presente em cada conjunto S_{ij}. Essa tarefa constitui o *problema da transversal mínima*. Note que mais de um conjunto S_{min} pode ser encontrado.

(3) Uma vez que S_{min} tenha sido determinado para o nó a_n, uma tabela-verdade é determinada para f_n, em termos dos níveis dos genes em S_{min}, tomando níveis relevantes diretamente de M. Se nem todas as combinações dos níveis de entrada estiverem presentes em M, o nível de saída correspondente para o gene a_n não pode ser determinado e é representado pelo símbolo "*" na tabela-verdade.

Para exemplificar o algoritmo acima, vamos ver como ele é utilizado para inferir a função para a_4 da matriz de expressão M mostrada na Figura 3.6. O nível de expressão de a_4 difere entre os pares de linhas (0,1), (0,3), (1,2) e (2,3). Note que na linha p_4, o nível de expressão de a_4 foi forçado a um valor *alto* e, portanto, p_4 não é considerada no Passo (1) do algoritmo. Ainda no Passo (1), para o par (0,1), encontramos o conjunto $S_{01} = \{a_0, a_2\}$ contendo todos os outros nós cujos níveis de expressão também diferem entre as linhas p_0 e p_1. A diferença no nível de a_4 entre as linhas p_0 e p_1 poderia ter sido causada por uma mudança no nível de a_0, a_2, ou ambos. Da mesma forma, temos $S_{03} = \{a_2\}$, $S_{12} = \{a_0, a_1\}$ e $S_{23} = \{a_1\}$. No Passo (2), encontramos $S_{min} = \{a_1, a_2\}$, que é o menor conjunto construído de tal forma que pelo menos um nó em S_{min} esteja presente em cada um dos conjuntos S_{ij}.

No Passo (3), determinamos a tabela-verdade para x_4, em termos de x_1 e x_2. Começamos por buscar linhas de M para as quatro combinações de valores binários para x_1 e x_2. Nas linhas p_0 e p_1 encontramos para o par x_1, x_2, os valores 1,1 e 1,0, respectivamente. Na linha p_2, $x_1 = -$ e $x_2 = 0$ mas, como o símbolo "$-$" significa que x_1 foi forçado a um valor *baixo*, temos $x_1 = 0$. Da mesma forma, na linha p_3, temos $x_1 = 1$ e $x_2 = - = 0$. Como anteriormente, a linha p_4 é excluída da análise. Como o estado onde $x_1 = 0$ e $x_2 = 1$ não é observado em M, não sabemos seu efeito sobre x_4, e representamos essa falta de conhecimento através da inserção do

símbolo "*" na tabela-verdade. Na Figura 3.7 mostramos a tabela-verdade para x_4 encontrada pelo algoritmo. Exceto pela falta de conhecimento observada acima, a tabela-verdade encontrada reproduz a original, da Figura 2.3.

x_1	x_2	x_4
0	0	0
0	1	*
1	0	1
1	1	0

Figura 3.7: Tabela-verdade construída pelo *Predictor*, tendo como entrada a matriz de expressão da Figura 3.6.

3.2.3 Avaliação do *Predictor*

O *Predictor* é um algoritmo que se baseia em uma solução para o problema da transversal mínima, que é NP-difícil.

Capítulo 4

Os Algoritmos Desenvolvidos

Após os estudos a respeito dos trabalhos mostrados no capítulo anterior, decidimos paralelizar o *Predictor* que, sob nossa análise, aborda o problema de uma forma mais elegante e simples. Baseados no *Predictor*, desenvolvemos um algoritmo seqüencial para encontrar as dependências entre os genes de uma rede e o paralelizamos.

Vimos que o *Predictor* é um algoritmo baseado em uma solução para o *problema da transversal mínima*, que é *NP-difícil*. Por essa razão, escolhemos buscar *algoritmos de aproximação* conhecidos para esse problema, observando a *razão de aproximação* de cada um deles, de forma que pudéssemos escolher o que melhor se adequasse ao nosso problema.

Neste capítulo, apresentamos uma introdução sobre NP-completude e o problema da transversal mínima, juntamente com os algoritmos de aproximação mais conhecidos para a sua solução e as nossas razões para a escolha de um deles. Depois disso, apresentamos o algoritmo seqüencial desenvolvido e sua versão paralelizada.

4.1 NP-completude e o Problema da Transversal Mínima

Antes da exposição e discussão acerca dos algoritmos de aproximação para o problema da transversal mínima pesquisados, vamos mostrar uma formulação para esse problema:

Problema da Transversal Mínima [19]: dados um conjunto finito E, uma coleção finita $\mathcal{S} = \{S_1, ..., S_w\}$ de subconjuntos de E, encontrar um subconjunto $A \subseteq E$ de cardinalidade mínima, tal que $A \cap S_i \neq \emptyset$ para todo $i = 1, ..., w$.

Muitas vezes, o problema da transversal mínima é formulado com custos nos elementos:

Problema da Transversal Mínima (com custos nos elementos) [19]: dados um conjunto finito E, uma coleção finita $\mathcal{S} = \{S_1, ..., S_w\}$ de subconjuntos de E e um custo não-negativo c_e para todo elemento $e \in E$, encontrar um subconjunto $A \subseteq E$ de custo mínimo, tal que $A \cap S_i \neq \emptyset$ para todo $i = 1, ..., w$.

Os problemas computacionais podem ser classificados em duas classes: aqueles para os quais existem algoritmos eficientes (tais problemas pertencem à classe P), e aqueles para os quais tais algoritmos não existem. Infelizmente, existe uma terceira classe importante de problemas que ainda não puderam ser devidamente classificados. Para esses problemas, ninguém conseguiu encontrar algoritmos eficientes para solucioná-los, mas por outro lado ninguém conseguiu mostrar que tais algoritmos não existem. Esses são chamados *problemas NP-completos*. Esses problemas pertencem à classe NP, que é a classe de problemas cuja solução, uma vez encontrada, pode ser verificada em tempo polinomial. Essa classe também inclui a classe P como subconjunto. Os problemas NP-completos são os mais difíceis da classe NP, ou seja, qualquer instância de qualquer problema na classe NP pode ser transformada, em tempo polinomial, em uma instância de um problema NP-completo. Um caso particular dessa afirmação é que todos os problemas NP-completos são equivalentes sob transformações polinomiais. Por sua vez, isso significa que se um algoritmo polinomial for encontrado para um problema NP-completo, então todos os problemas NP-completos poderão ser resolvidos em tempo polinomial [26].

Outro termo relacionado que utilizamos é *problema NP-difícil*. Para explicar a diferença entre os problemas NP-completos e NP-difíceis, necessitamos de algumas afirmações adicionais a respeito das classes de problemas mencionadas acima. As classes P e NP incluem apenas problemas de decisão: aqueles cuja resposta deve ser sim ou não. Um exemplo típico é o problema de decidir se um grafo é hamiltoniano. No entanto, a maioria dos problemas que encontramos são problemas de otimização, o que significa que existe alguma função associada com o problema e queremos encontrar a solução que minimize ou maximize o valor dessa função. Esse é o caso, por exemplo, do problema da transversal mínima. Um problema de otimização pode ser transformado em um problema de decisão incluindo um parâmetro K como parte da entrada e perguntando se existe uma solução cujo valor é $\leq K$ (no caso de problemas de minimização) ou $\geq K$ (no caso de problemas de maximização). Esse truque nos permite ter o problema de decisão associado a qualquer problema de otimização e portanto podemos tentar associá-lo à classe P ou NP. Segundo Garey e Johnson[18], uma instância para o problema da transversal mínima é uma coleção C de subconjuntos de um conjunto S e um inteiro positivo K. O problema de decisão associado ao problema da transversal mínima é: S contém uma transversal para C de tamanho menor ou igual a K, que é um subconjunto S' de S com $|S'| \leq K$ e tal que S' contém no mínimo um elemento de cada subconjunto em C? Pode-se

demonstrar que esse problema de decisão é NP-completo. Mas o que podemos dizer sobre o problema da transversal mínima como enunciado primeiramente? Ele não pertence à classe NP, já que ele não é um problema de decisão; mas ele é claramente no mínimo tão difícil quanto seu correspondente problema de decisão. Problemas que são no mínimo tão difíceis quanto um problema NP-completo mas que não pertencem à classe NP são chamados NP-difíceis. Todo problema de otimização que tem um problema de decisão NP-completo associado a ele é um problema NP-difícil. Portanto, o problema da transversal mínima é NP-difícil e segundo Hochbaum [19] é equivalente ao conhecido problema da cobertura de conjuntos (*set cover*). Devido a essa equivalência, resultados para o problema da cobertura de conjuntos podem ser transformados em resultados para o problema da transversal mínima.

Para atacar um problema NP-completo ou NP-difícil, podemos desenvolver um algoritmo de aproximação polinomial. Tal algoritmo encontra uma solução que tem uma garantia de ser próxima da ótima, mas que não é necessariamente a ótima. Para uma definição de algoritmo de aproximação, considere um problema de otimização em que $val(\mathcal{S}) \geqslant 0$ para toda solução viável $\mathcal{S}$ de qualquer instância do problema, onde $val(\mathcal{S})$ é o valor de $\mathcal{S}$. Seja A um algoritmo que, para toda instância viável I do problema, devolve uma solução viável $A(I)$ de I. Considere $opt(I)$ o valor da solução ótima para um problema de minimização dada a instância I. Se para esse problema temos que $val(A(I)) \leqslant \alpha opt(I)$ para toda instância I, dizemos que A é uma $\alpha-$aproximação para o problema. Além disso, chamamos α de razão de aproximação do algoritmo [17].

4.1.1 O Algoritmo Guloso

O primeiro algoritmo estudado utiliza uma estratégia gulosa e é devido a Johnson [23]. Os passos deste algoritmo estão detalhados no Algoritmo 3.

Seja $\mathcal{H}_d$ o d-ésimo número harmônico $\sum_{i=1}^{d} \frac{1}{i}$. Seja $|S(e)|$ o número de conjuntos na coleção $\mathcal{S}$ que são cobertos pelo elemento e.

O algoritmo guloso é uma ρ-aproximação polinomial para o problema da transversal mínima, onde $\rho = \mathcal{H}_d$, com $d = (\max_{e \in E}\{|S(e)|\})$. Segundo Jha *et al.* [22], a prova da afirmação anterior segue da equivalência entre o problema da transversal mínima e o problema da cobertura de conjuntos e da prova da razão de aproximação do algoritmo guloso para a cobertura de conjuntos (que pode ser encontrada no trabalho de Cormen *et al.* [7]). Da afirmação anterior, segue ainda que a razão de aproximação do algoritmo guloso para o problema da transversal mínima é limitada por $\ln |\mathcal{S}| + 1$ [7, 19].

Algoritmo 3 Transversal Mínima Guloso

Entradas: o conjunto de elementos E e a coleção $\mathcal{S}$ de subconjuntos de E.
Saída: os elementos que constituem uma transversal mínima.

Sejam E' e $\mathcal{S}'$ conjuntos vazios inicialmente. Repita os seguintes passos até que $\mathcal{S}' = \mathcal{S}$.

(1) Escolha um elemento e do conjunto $E \backslash E'$ que cobre o maior número de conjuntos na coleção $\mathcal{S} \backslash \mathcal{S}'$.

(2) Seja e o elemento escolhido no passo anterior e seja $S(e)$ a coleção de conjuntos em $\mathcal{S}$ cobertos por e:

$$E' \leftarrow E' \cup \{e\}$$
$$\mathcal{S}' \leftarrow \mathcal{S}' \cup S(e)$$

4.1.2 O Algoritmo Primal-Dual

O segundo algoritmo estudado é resultante da aplicação do método de aproximação primal-dual ao problema da transversal mínima. Esse algoritmo é devido a Bar-Yehuda e Even [2], e foi originalmente concebido para o problema da cobertura mínima por vértices. O algoritmo é uma α-aproximação polinomial para o problema da transversal mínima com custos nos elementos, onde $\alpha = \max_{i=1}^{w} |S_i|$, com $w = |S|$ [19, 17]. Claramente, esse algoritmo poderia ser usado para o problema da transversal mínima sem custos nos elementos, bastando apenas considerar que todos os elementos de E possuem o mesmo custo.

4.1.3 A Escolha do Algoritmo de Aproximação

Dados os dois algoritmos mencionados acima, a tarefa agora é escolher um deles de acordo com o problema inicial de encontrar dependências entre genes que o *Predictor* resolve. Vamos supor que E é o conjunto de genes analisados, com $|E| = n$, e seja m o número de experimentos feitos com tais n genes. Sabemos que $n >> m$. Em experimentos reais, podemos dizer, por exemplo, que n é aproximadamente 10.000, enquanto m é aproximadamente 20. A entrada para o algoritmo de aproximação será uma matriz M, com m linhas e n colunas e um gene a_n que se deseja estudar. Antes de detalhar o algoritmo, é importante saber que o primeiro passo do *Predictor*, e portanto de nosso algoritmo, é construir uma coleção $\mathcal{S}$ de subconjuntos S_{ij} de E. Cada um desses subconjuntos representa um par de linhas (i, j) no qual o nível de expressão do gene a_n sofreu modificação e contém todos os outros genes que também sofreram modificação no par de linhas (i, j). O que se deseja depois desse passo é aplicar uma das aproximações para o problema da transversal mínima

para descobrir o menor conjunto de genes que, juntos, cobrem todos os subconjuntos S_{ij} de $\mathcal{S}$. Para decidir entre os dois algoritmos de aproximação comentados vamos considerar suas razões de aproximação aplicadas ao problema.

Vimos que a razão de aproximação do algoritmo primal-dual para o problema da transversal mínima é $\alpha = \max_{i=1}^{w} |S_i|$ o que, aplicando às nossas necessidades, equivale a dizer que α é o tamanho do maior subconjunto S_{ij} encontrado pelo *Predictor* em seu primeiro passo. Para um determinado par de linhas (i, j) onde a_n sofreu modificação temos que o conjunto S_{ij} contém todos os outros genes que também sofreram modificação no mesmo par de linhas. O tamanho de S_{ij} é portanto $O(n)$. Como sabemos que o número de genes n da entrada pode ser muito grande, e geralmente será, podemos concluir que o resultado do algoritmo primal-dual poderia não ser uma boa aproximação.

Vamos considerar agora a razão de aproximação, $\ln w + 1$ (onde w é igual a $|\mathcal{S}|$), do algoritmo guloso. Trazendo para a realidade de nosso problema, w é o número de subconjuntos construídos pelo *Predictor* em seu primeiro passo. Temos que cada $S_{ij} \in \mathcal{S}$ refere-se a um par de linhas da matriz de entrada M onde o nível de expressão do gene a_n sofreu modificação. Sabemos que M possui m linhas, o que nos leva a concluir que o *Predictor* pode encontrar, no máximo, $\frac{m(m-1)}{2}$ pares de linhas distintas. Portanto, o *Predictor* terá $O(m^2)$ subconjuntos pertencentes a $\mathcal{S}$. Daí, $|\mathcal{S}| = w = O(m^2)$.

Como $n >> m$, podemos concluir que o algoritmo guloso para o problema da transversal mínima, aplicado ao algoritmo para resolver o problema inicial, produzirá uma melhor aproximação. Feita essa escolha, aplicamos o algoritmo de aproximação guloso para o problema da transversal mínima dentro do *Predictor*, obtendo um novo algoritmo, que apresentamos na próxima seção.

4.2 O Algoritmo Seqüencial

Vamos chamar nosso algoritmo de **DepAproxG** (Dependência Aproximada Gulosa). Sejam dados como entrada uma matriz de expressão M, com $|M| = m \times n$ e $n >> m$, e um gene para o qual se deseja descobrir as dependências, como no *Predictor*. Convencionamos que o gene em estudo será sempre o último gene a_{n-1} e, portanto, a última coluna da matriz corresponde aos níveis de expressão do gene a_{n-1} em estudo.

4.2.1 Detalhes do Algoritmo e das Estruturas de Dados

Desejamos encontrar os genes dos quais o gene a_{n-1} depende. Para isso, o algoritmo deve encontrar primeiramente os pares de linhas de M onde o nível de ex-

pressão de a_{n-1} difere. Cada par de linhas (i, j) encontrado representa um conjunto S_{ij}. Para cada um desses pares de linhas, o algoritmo deve procurar os outros genes de M que também têm seus níveis de expressão modificados da linha i para a linha j. Tais genes são os elementos do conjunto S_{ij}.

O algoritmo deve construir um vetor, que chamaremos de *vetor genes*, onde cada elemento representa um gene e contém informações relevantes sobre ele. Cada elemento a_x do *vetor genes* conterá os seguintes campos:

- *ocorrência*: armazena o número de conjuntos S_{ij} aos quais o gene a_x pertence;
- *lista*: um ponteiro para uma lista ligada cujos elementos representam conjuntos S_{ij} dos quais o gene a_x faz parte.

O algoritmo deve construir também um outro vetor, que chamaremos de *vetor coleção*, onde cada elemento representa um conjunto S_{ij} e contém informações relevantes sobre esses conjuntos. Cada elemento do *vetor coleção* deve conter os seguintes campos:

- *i1 e j1*: armazenam respectivamente a linha i e a linha j do par (i, j) que o elemento representa;
- *coberto*: contém o valor "verdadeiro" se o conjunto representado já se encontra coberto, ou "falso", caso contrário. A função deste campo poderá ser melhor compreendida mais adiante;
- *lista*: um ponteiro para uma lista ligada onde cada elemento corresponde a um gene do *vetor genes* que está contido neste conjunto.

O *vetor genes* terá tamanho $n - 1$ (pois não conterá informações sobre o gene alvo) e o *vetor coleção* terá tamanho, no máximo, m^2. Isso se deve ao fato de que, como já vimos anteriormente, cada elemento do *vetor coleção* corresponde a um conjunto S_{ij}, que representa um par de linhas (i, j) da matriz onde ocorreu mudança no nível de expressão de a_{n-1}. Com m linhas, o número máximo de pares de linhas que podem diferir uma da outra é $O(m^2)$. Podemos ver essas estruturas de dados representadas na Figura 4.1 e na Figura 4.2, que passaremos a explicar.

Expostas as estruturas de dados principais, podemos voltar ao primeiro passo do algoritmo aproximado, que é percorrer a última coluna da matriz M, referente aos níveis de expressão do gene a_{n-1} em análise. Cada vez que encontrar um par de linhas (i, j) onde o nível de expressão de a_{n-1} difere, o algoritmo deverá atribuir a tal par de linhas um índice d no *vetor coleção*, onde o campo *i1* será i e o campo *j1* será j. Além disso, o algoritmo deverá verificar para quais outros genes (colunas de M) houve mudança em seu nível de expressão entre o par de linhas (i, j). Para cada gene a_x que também teve seu nível de expressão modificado no par de linhas (i, j), o

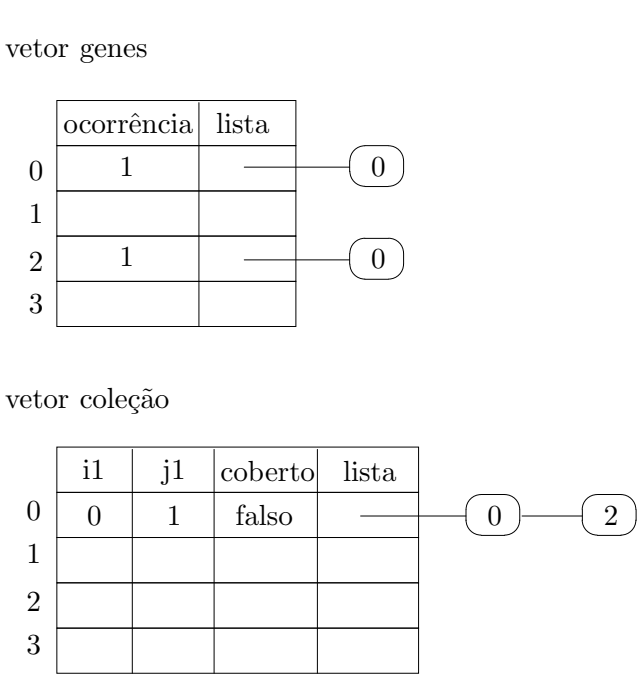


Figura 4.1: Estruturas de dados.

algoritmo deverá tomar o elemento de índice x do *vetor genes* e somar 1 a seu campo *quantidade*, sinalizando que o elemento a_x está presente em mais um conjunto do *vetor coleção*. O passo seguinte é colocar o índice d no campo *lista* do gene a_x, sinalizando que a_x pertence ao conjunto representado pelo elemento de índice d do *vetor coleção*. Dessa forma, o algoritmo terá todas as informações necessárias para o início do passo 2, onde ele aplicará o algoritmo aproximado para o problema da transversal mínima.

Vamos tomar como exemplo a matriz M da Figura 3.6. Seja M a matriz de entrada para o algoritmo, onde a última coluna corresponde aos níveis de expressão associados ao gene a_4, que se deseja estudar. Tomemos os pares de linhas de M onde o nível de expressão de a_4 difere: (0,1), (0,3), (1,2) e (2,3). Ao encontrar o primeiro par (0,1), o algoritmo utiliza o índice 0 do *vetor coleção* para representá-lo, coloca 0 no campo *i1* e 1 no campo *j1* e verifica, para todos os outros genes da matriz M, quais deles tiveram seu nível de expressão modificado da linha 0 para a linha 1. Para o exemplo, os genes a_0 e a_2 são encontrados. Podemos ver na Figura 4.1 detalhes das mudanças das estruturas após essas primeiras operações.

Notemos que o par de linhas (0,1) ganhou o elemento de índice 0 do *vetor coleção*. No seu campo *lista* foram colocados os índices dos elementos do *vetor genes* que também sofreram modificação no par (0,1). Nesse caso, os índices do *vetor genes* são 0 e 2, representando os elementos a_0 e a_2. No *vetor genes* os genes de índices 0 e 2 tiveram seus campos *ocorrência* acrescidos de 1 e o índice 0 (que se refere ao conjunto S_{01}) foi inserido na lista de cada um deles. Da mesma forma procedemos

com os outros pares de linhas e, ao final desse passo, teremos nossas estruturas de dados como mostra a Figura 4.2.

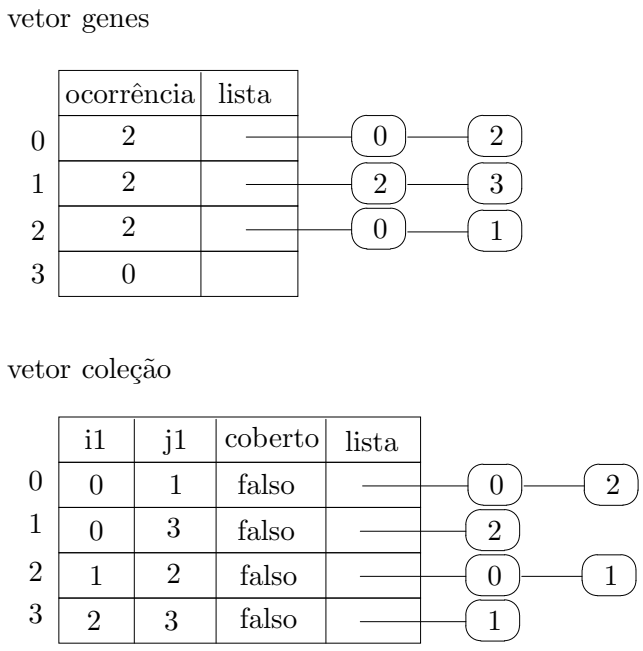


Figura 4.2: Estruturas de dados após processamento de toda a matriz.

O passo seguinte é aplicar o algoritmo de aproximação guloso para o problema da transversal mínima. Vamos imaginar o conjunto de genes (*vetor genes*) como um conjunto universo E e o *vetor coleção* como uma coleção de subconjuntos de E, chamada $\mathcal{S}$. O problema da transversal mínima busca um subconjunto TM de E contendo o menor número de genes que, juntos, cobrem todos os subconjuntos da coleção $\mathcal{S}$. Em outras palavras, todos os subconjuntos da coleção $\mathcal{S}$ devem conter pelo menos um dos genes de TM. Esse conjunto TM conterá o menor número possível de genes (ou o menor número possível de genes encontrados pela aproximação) responsáveis pelo comportamento de a_{n-1}.

A estratégia gulosa consiste em considerar o *vetor genes* e encontrar um elemento que está presente no maior número de conjuntos do *vetor coleção*. Para encontrar esse gene, basta que o algoritmo percorra o *vetor genes*, na busca pelo elemento de maior valor no campo *ocorrência*. Ao encontrar um gene a_k de maior valor no campo *ocorrência*, o algoritmo deverá colocá-lo no vetor TM e percorrer seu campo *lista*, que indica de quais conjuntos do *vetor coleção* o gene a_k faz parte. Chamemos de L tal lista. Para cada elemento l presente em L, que indexa um elemento do *vetor coleção*, o algoritmo verifica seu campo *coberto*. Se esse campo estiver marcado com "verdadeiro", nada fazemos. Se, ao contrário, o campo *coberto* estiver "falso", o algoritmo marca o campo *coberto* com "verdadeiro" e visita todos os genes pertencentes a esse conjunto, referenciados em seu campo *lista*. Chamemos tal lista de F. Para cada elemento f presente na lista F, o algoritmo toma o

elemento indexado por f do *vetor genes* e subtrai 1 de seu campo *ocorrência*.

Ao terminar esses passos, o algoritmo volta ao passo de encontrar um gene de maior campo *ocorrência* e repete os passos acima até que todos os conjuntos do *vetor coleção* estejam com campo *coberto* marcado com "verdadeiro". Podemos ver na figura 4.3 as estruturas de dados após o primeiro gene de maior valor no campo *ocorrência* ser encontrado (índice 0), tendo como base as estruturas de dados da Figura 4.2.

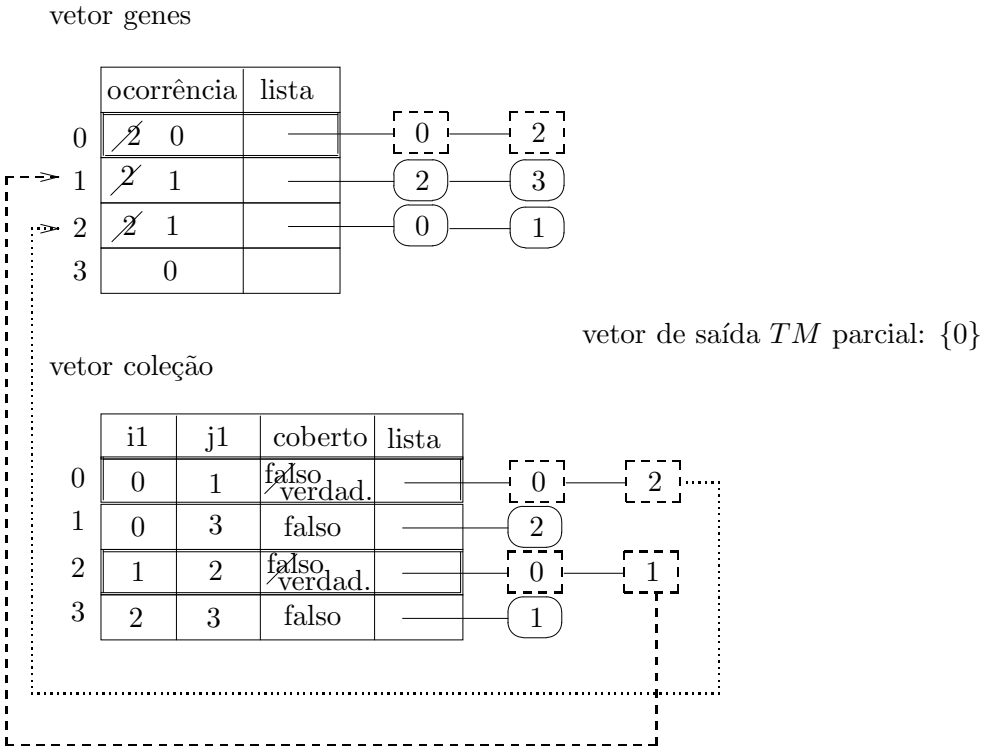


Figura 4.3: Situação após encontrar um gene de maior campo *ocorrência* (índice 0).

Ao encontrar o gene de índice 0 (a_0), o algoritmo começa a percorrer sua lista de índices, que contém os índices 0 e 2. Isso significa que o gene a_0 está presente nos conjuntos de índices 0 e 2 do *vetor coleção*. Primeiramente, para o elemento de índice 0, o algoritmo verifica que seu campo *coberto* está marcado com "falso". O algoritmo marca o campo *coberto* com "verdadeiro" e percorre sua lista, encontrando os índices 0 e 2. Daí, o algoritmo vai até o *vetor genes*, nos elementos de índices 0 e 2 e subtrai seu campo *ocorrência* de 1. O algoritmo realiza as mesmas operações para o conjunto de índice 2 do *vetor coleção*. Depois, ele volta a procurar o elemento de maior campo *ocorrência* e refaz todos os passos acima, até que todos os elementos do *vetor coleção* estejam com seu campo *coberto* marcado com "verdadeiro".

Apresentamos a seguir o algoritmo **DepAproxG**.

Algoritmo 4 DepAproxG

Entradas: uma matriz de expressão M de tamanho $m \times n$. Vamos convencionar que o último gene da matriz é o gene que se deseja estudar.
Saída: um vetor TM com os genes dos quais o último gene da matriz depende.
Observações: o algotimo constrói as estruturas de dados e chama o algoritmo **TransvGuloso** (Algoritmo 5) para obter a saída. Chamaremos o *vetor genes* de E e o *vetor coleção* de S.

```
1: tamanho ← 0
2: for (i = 0 to m − 1) do
3:    for (j = i + 1 to m − 1) do
4:       if (M[i][n − 1] e M[j][n − 1] são diferentes) then
5:          S[tamanho].i1 ← i
6:          S[tamanho].j1 ← j
7:          S[tamanho].coberto ← "falso"
8:          for (k = 0 to n − 2) do
9:             if (M[i][k] e M[j][k] são diferentes) then
10:               E[k].ocorrência ← E[k].ocorrência +1
11:               Insere k em S[tamanho].lista
12:               Insere tamanho em E[k].lista
13:            end if
14:         end for
15:         tamanho ← tamanho + 1
16:      end if
17:   end for
18: end for
19: TM ← TransvGuloso(E, n, S, tamanho);
20: retorne TM;
```

4.2.2 Complexidades de Tempo e de Espaço do Algoritmo DepAproxG (Algoritmo 4)

Para analisar a complexidade do algoritmo DepAproxG (Algoritmo 4), precisamos observar seus passos. Vamos determinar primeiramente sua complexidade de tempo. O Algoritmo 4 possui 3 laços aninhados que, juntos, consomem tempo $O(m^2n)$. Esses laços são responsáveis por construir as estruturas de dados. Após a construção das estruturas de dados, o Algoritmo 4 chama o TransvGuloso (Algoritmo 5). Podemos observar que o Algoritmo 5 possui um laço principal que é executado k vezes, onde k é o tamanho da transversal mínima encontrada. Essa afirmação deve-se ao fato de *tamanho* representar a quantidade de conjuntos (elementos do *vetor coleção*) ainda não cobertos nas iterações anteriores do algoritmo. Se a transversal mínima encontrada tiver tamanho 1, o laço da linha 1 será executado apenas 1 vez, pois todos os elementos do *vetor coleção* já estarão cobertos. Dentro desse laço principal, a linha 2 obtém o gene de maior valor no campo *ocorrência* em

Algoritmo 5 TransvGuloso

Entradas: o vetor genes E e seu tamanho n e o vetor coleção S e seu tamanho *tamanho*.
Saída: um vetor TM resposta.
Observações: os elementos das listas possuem dois campos: um campo *info* e um campo *próximo*, apontando para o próximo elemento ou para NIL.

```
1: while (tamanho > 0) do
2:    Selecione um gene a_x com valor máximo no campo ocorrência em E.
3:    Insere x em TM
4:    Y ← E[x].lista
5:    while (Y ≠ NIL) do
6:       z ← info(Y)
7:       if (S[z].coberto = "falso") then
8:          S[z].coberto ← "verdadeiro"
9:          tamanho ← tamanho − 1
10:         K ← S[z].lista
11:         while (K ≠ NIL) do
12:            s ← info(K)
13:            E[s].ocorrência ← E[s].ocorrência−1
14:            K ←próximo(K)
15:         end while
16:      end if
17:      Y ←próximo(Y)
18:   end while
19: end while
20: retorne TM
```

tempo $O(n)$. O laço da linha 5 é executado $O(m^2)$ vezes, pois ele percorre a lista de conjuntos de cada gene escolhido para a transversal mínima, e podemos ter $O(m^2)$ conjuntos. O laço da linha 11 leva tempo $O(n)$, pois ele percorre a lista dos genes pertencentes a um conjunto. Portanto, o laço da linha 5 leva tempo total $O(m^2n)$. De tudo isso, o laço da linha 1 consome tempo $O(km^2n)$ que é a complexidade de tempo do Algoritmo 5. No Algoritmo 4, unindo a complexidade de tempo para a construção das estruturas e a chamada ao Algoritmo 5, temos $O(m^2n) + O(km^2n)$. Daí, a complexidade de tempo do algoritmo DepAproxG é $O(km^2n)$. Essa complexidade também pode ser expressa apenas em termos do número de linhas e colunas da matriz de entrada. Como o tamanho do *vetor coleção* é $O(m^2)$, podem ser necessários $O(m^2)$ elementos na transversal para cobrir todos os conjuntos do *vetor coleção*. Dessa forma, o tamanho k da transversal encontrada é também $O(m^2)$ e a complexidade de tempo do algoritmo DepAproxG é $O(m^4n)$.

Em termos de complexidade de espaço, duas estruturas de dados de tamanho $O(m^2n)$ são utilizadas.

4.3 O Algoritmo Paralelo

Para a versão paralela do algoritmo DepAproxG, a matriz de entrada M é particionada verticalmente para ser armazenada em cada processador. Chamaremos o algoritmo paralelo de **DepAproxGParalela**. Aqui, como no algoritmo seqüencial, convencionamos que a última coluna da matriz de entrada refere-se aos níveis de expressão do gene para o qual se deseja descobrir as dependências. Podemos ver na Figura 4.4 um exemplo de particionamento de uma matriz de entrada M de tamanho $m \times 10$ entre $p = 3$ processadores. Podemos notar que a última coluna (referente ao gene a_9) não entra na divisão. Supomos que os níveis de expressão do gene que se deseja estudar estão em um vetor v, presente em todos os processadores.

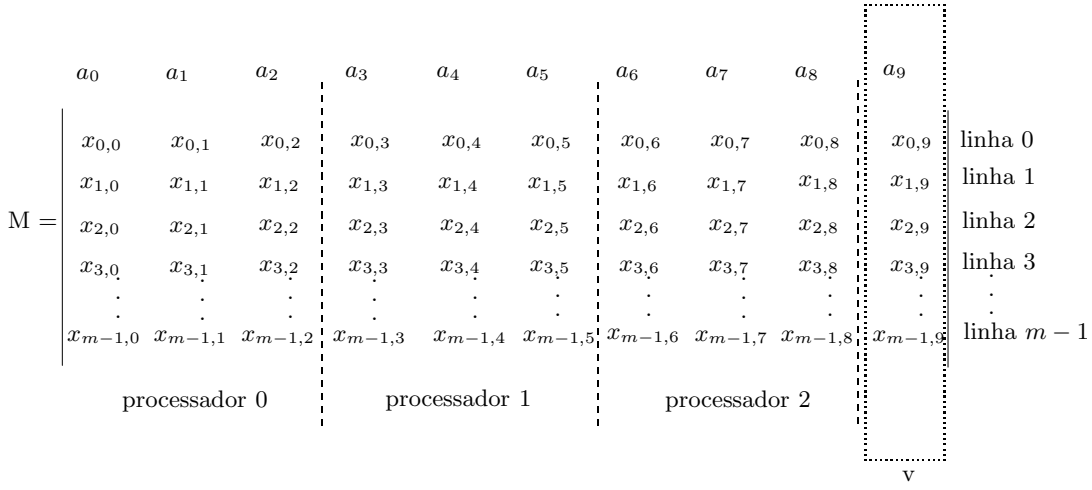


Figura 4.4: Divisão de uma matriz $m \times 10$ entre 3 processadores.

O algoritmo DepAproxGParalela correspondente ao DepAproxG é obtido dividindo as responsabilidades de cada processador de tal forma que cada processador lida com os níveis de expressão apenas dos genes que ele armazena. Além do algoritmo ser naturalmente paralelizado, as estruturas de dados podem ser divididas basicamente em p partes, onde p é o número de processadores. Portanto, o tempo de processamento e os requisitos de espaço são igualmente divididos entre os processadores. Utilizando o modelo *CGM*, o objetivo é apresentar o algoritmo paralelo que obtivemos, juntamente com detalhes sobre a divisão de tarefas entre os p processadores, sabendo que a entrada para o algoritmo é uma matriz de dimensões $m \times n$, com $n >> m$.

4.3.1 Detalhes do Algoritmo para o Modelo CGM e da Divisão das Estruturas de Dados entre os Processadores

Cada processador será responsável por ler um pedaço da matriz de entrada, de tamanho $m \times \frac{n-1}{p}$. Todos os processadores devem ter um vetor v correspondente à última coluna da matriz M. Podemos notar que cada processador ficará responsável por parte dos genes e os níveis de expressão associados a eles. Cada processador deverá criar um *vetor genes*, como no algoritmo seqüencial, porém o vetor conterá apenas informações a respeito dos genes que estão sob sua responsabilidade. Portanto, o *vetor genes* de cada processador terá tamanho $\frac{n-1}{p}$.

Cada processador deverá também criar um *vetor coleção*, da mesma forma que no algoritmo seqüencial. Primeiramente, cada processador deverá tomar o vetor v que contém as informações do gene a_{n-1} a serem estudadas. Cada processador deverá verificar em quais pares de linhas foram observadas mudanças no nível de expressão de a_{n-1}. Da mesma forma, cada processador deverá atribuir um índice a cada conjunto encontrado e preencher o *vetor coleção* com *i1*, *j1* e *coberto* = "falso". Para cada par de linhas (i, j) encontrado, o processador deverá observar, dentre os genes que estão sob sua responsabilidade (no seu *vetor genes*), quais também sofreram modificação no par de linhas (i, j). Então, o processador deverá inserir os índices de tais elementos no campo *lista* do conjunto representante de tal par de linhas.

Para exemplificar, vamos considerar a matriz de expressão M da Figura 3.6. Supomos que temos 2 processadores disponíveis. A divisão da matriz M entre os 2 processadores pode ser vista na Figura 4.5. Para aplicar o algoritmo paralelo, supomos que a matriz de entrada, excluindo a coluna de níveis de expressão do gene a ser estudado, possui um número de colunas que seja potência de dois, para que todos os processadores contenham o mesmo número de genes.

$$M = \begin{array}{|cc:cc:c|c|} \multicolumn{2}{c:}{x_0 \quad x_1} & \multicolumn{2}{c:}{x_2 \quad x_3} & x_4 & \\ \hline 1 & 1 & 1 & 0 & 0 & p_0 \\ - & 1 & 0 & 0 & 1 & p_1 \\ 1 & - & 0 & 0 & 0 & p_2 \\ 1 & 1 & - & 0 & 1 & p_3 \\ 1 & 1 & 1 & 0 & + & p_4 \\ \hline \multicolumn{2}{c:}{P_0} & \multicolumn{2}{c:}{P_1} & \text{v} & \end{array}$$

Figura 4.5: Divisão da matriz M entre 2 processadores.

Observemos na Figura 4.6 um exemplo de paralelização da construção das es-

truturas de dados, onde é utilizada a matriz da Figura 4.5 e 2 processadores. Para um melhor entendimento, podemos comparar essa construção com a do algoritmo seqüencial, exemplificada na Figura 4.2.

Após construídas as estruturas, a próxima tarefa é encontrar uma solução aproximada para o problema da transversal mínima. Os seguintes passos dever-se-ão repetir até que todos os conjuntos do *vetor coleção* de todos os processadores estejam cobertos. Cada processador deve encontrar no *vetor genes* o índice de um gene com maior valor no campo *ocorrência*. Feito isso, cada processador envia seu elemento para o processador 0, que escolhe dentre os elementos recebidos um de maior valor no campo *ocorrência*. Feito isso, o processador 0 coloca o índice de tal elemento no vetor de saída TM e avisa o dono de tal elemento, para que ele envie para todos os outros processadores o campo *lista* do gene escolhido, contendo todos os conjuntos dos quais ele faz parte. Quando cada processador recebe a *lista* do dono, ele deverá percorrer essa lista, indo no *vetor coleção*, em cada índice do campo *lista*, para verificar se o conjunto já está coberto. Se não estiver, o campo *coberto* é marcado com "verdadeiro" e sua lista é percorrida. Para cada índice de gene encontrado na lista, o processador deve ir até ele e decrementar seu campo quantidade.

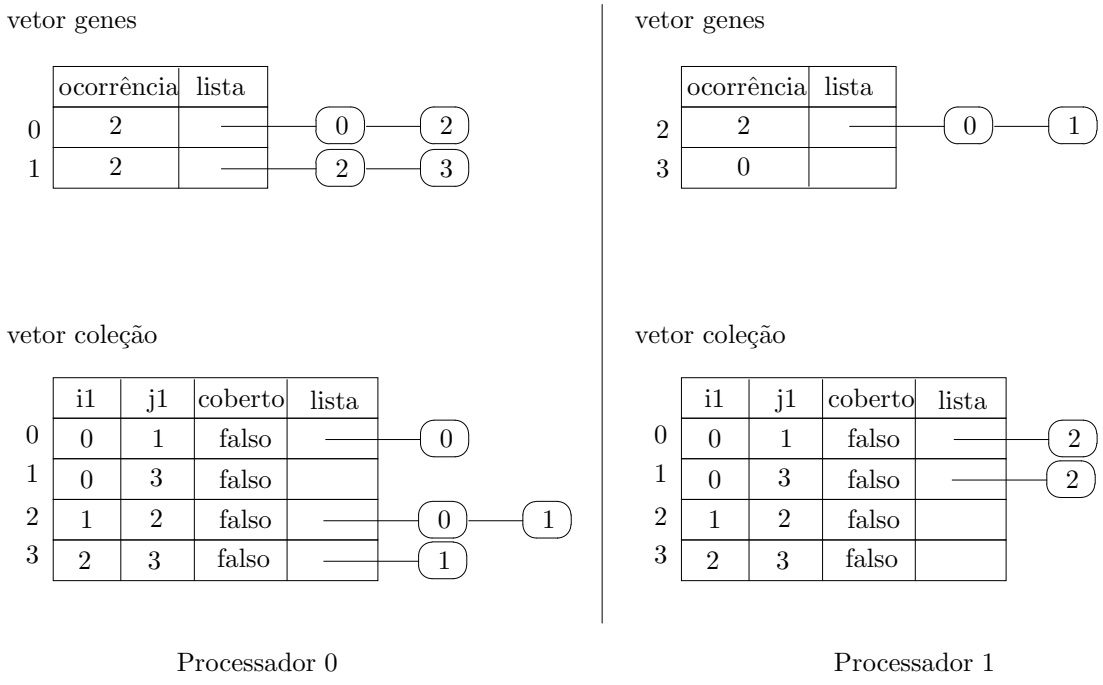


Figura 4.6: Construção das estruturas de dados pelos $p = 2$ processadores.

Apresentamos a seguir o algoritmo DepAproxGParalela (Algoritmo 6).

Algoritmo 6 DepAproxGParalela - para o processador i

***E*ntradas**: um pedaço de tamanho $m \times \frac{n-1}{p}$ da matriz M correspondente a $\frac{n-1}{p}$ genes do experimento e um vetor v de tamanho m correspondente aos estados do gene a ser estudado.
***S*aída**: um vetor TM contendo os genes dos quais o gene alvo depende. O processador P_0 armazenará o vetor TM.
***O*bservações**: chamaremos de E_i o *vetor genes* e de S_i o *vetor coleção* do processador i. Assumimos, por simplicidade, que o *vetor genes* do processador i tem índices que iniciam em $(i\frac{n-1}{p})$. Pela mesma razão, a matriz M_i de cada processador terá suas linhas indexadas de $(i\frac{n-1}{p})$ a $(((i+1)\frac{n-1}{p}) - 1)$. Os elementos das listas têm 2 campos: um campo *info* e um campo *próximo*, apontando para o último elemento ou para NIL.

1: $tamanho \leftarrow 0$;
2: **for** ($x = 0$ to $m - 1$) **do**
3: **for** ($y = x + 1$ to $m - 1$) **do**
4: **if** ($v_i[x] \neq v_i[y]$) **then**
5: $S[tamanho].i1 \leftarrow x$; $S[tamanho].j1 \leftarrow y$; $S[tamanho].coberto \leftarrow$ "$falso$";
6: **for** ($z = (i \times \frac{n-1}{p})$ to $(((i+1) \times \frac{n-1}{p}) - 1)$) **do**
7: **if** ($M_i[x][z]$ e $M_i[y][z]$ diferem) **then**
8: $E[z]$.ocorrência $\leftarrow E[z]$.ocorrência +1;
9: Insere z em $S_i[tamanho].lista$; Insere $tamanho$ em $E_i[z].lista$;
10: **end if**
11: **end for**
12: $tamanho \leftarrow tamanho + 1$;
13: **end if**
14: **end for**
15: **end for**
16: **while** ($tamanho > 0$) **do**
17: $x \leftarrow$ um elemento de maior valor no campo *ocorrência* em E_i;
18: **if** ($i \neq 0$) **then**
19: Envia x, $E_i[x]$.ocorrência e i para P_0 e receber de P_0 o número do processador que contém o elemento escolhido como máximo entre os máximos e colocá-lo na variável *dono*;
20: **else**
21: Recebe x,$E_i[x]$.ocorrência e *remetente* dos outros processadores;
22: $x \leftarrow$ elemento de maior campo *ocorrência* dentre todos os recebidos dos outros processadores;
23: $dono \leftarrow$ número do processador que contém o elemento x;
24: Insere x em TM;
25: Envia aviso para todos os processadores de que *dono* foi escolhido;
26: **end if**
27: **if** ($i = dono$) **then**
28: Envia a lista de $E_i[x]$ p/ todos os processadores e colocá-la em Y;
29: **else**
30: Recebe a lista de $E_i[x]$ do processador *dono* e colocá-la em Y;
31: **end if**
32: **while** ($Y \neq NIL$) **do**
33: $z \leftarrow info(Y)$;
34: **if** ($S[z].coberto =$ "$falso$") **then**
35: $S[z].coberto \leftarrow$ "$verdadeiro$"; $tamanho \leftarrow tamanho - 1$; $K \leftarrow S_i[z].lista$;
36: **while** ($K \neq NIL$) **do**
37: $s \leftarrow info(K)$; $E_i[s]$.ocorrência $\leftarrow E_i[s]$.ocorrência-1; $K \leftarrow$próximo(K);
38: **end while**
39: **end if**
40: $Y \leftarrow$ próximo(Y);
41: **end while**
42: **end while**

4.3.2 Complexidades de Tempo e de Espaço do Algoritmo Paralelo

Podemos notar que o algoritmo DepAproxGParalela particiona o tempo total $O(km^2n)$ de processamento seqüëncial de um modo muito natural entre os p processadores. O algoritmo paralelo tem complexidade de tempo de $O(\frac{km^2n}{p})$, ou $O(\frac{m^4n}{p})$, onde novamente k é o tamanho da transversal encontrada e $m \times n$ o tamanho da matriz M de entrada.

Adicionalmente ao tempo de processamento, o algoritmo paralelo requer $O(k)$ rodadas de comunicação. Podemos dizer também que o algoritmo paralelo requer $O(m^2)$ rodadas de comunicação.

O algoritmo seqüencial requer espaço $O(m^2n)$, o qual é convenientemente particionado entre os p processadores, resultando em requerimento de espaço de $O(\frac{m^2n}{p})$ em cada processador.

Capítulo 5

Resultados Experimentais

Observar como resultados teóricos se comportam na prática é uma tarefa bastante importante dentro da área de algoritmos paralelos. Buscamos através de experimentos comprovar a eficiência de nossos programas, frente à difícil tarefa de lidar com uma quantidade considerável de dados. O ideal era que conseguíssemos testá-los com dados reais, o que não foi possível. Porém, simulando de forma simples a realidade, conseguimos obter resultados animadores com relação ao desempenho de nossos programas. No presente capítulo, falamos a respeito das implementações, das características da máquina utilizada e dos resultados obtidos.

5.1 Implementações

Implementamos os algoritmos seqüencial e paralelo, descritos no Capítulo 4, utilizando a linguagem C e a biblioteca *MPI (Message Passing Interface)*.

O programa paralelo desenvolvido no presente trabalho segue o modelo SPMD (*Single Program Multiple Data*), no qual cada processo executa o mesmo programa. Apesar disso, os processos podem executar instruções diferentes pois podem seguir ramificações diferentes do programa, dependendo de seu número. Nossos programas seqüencial e paralelo recebem como entrada um arquivo contendo a matriz de expressões que desejamos investigar. No caso do programa paralelo, um dos processos lê toda a matriz, faz a divisão vertical e distribui cada uma das partes aos processadores, conforme descrito no Algoritmo 6. O tempo de leitura e distribuição dos dados da entrada não é incluído na contagem do tempo do programa. A contagem de tempo do programa paralelo é feita pela função *MPI_Wtime* do MPI. Para que pudesse haver coerência entre as medidas de tempo seqüencial e paralelo, utilizamos a biblioteca MPI também no programa seqüencial, de forma que pudéssemos utilizar a função MPI_Wtime para computar o tempo do programa. Dessa forma, o programa seqüencial utiliza a biblioteca MPI, mas não utiliza primitivas de co-

municação, e é executado por apenas um processo, sendo que a única função da biblioteca MPI nesse programa é computar o tempo. Também no programa seqüencial não consideramos o tempo de leitura da matriz de entrada. Os códigos dos programas seqüencial e paralelo podem ser encontrados no Apêndice A.

Procuramos obter dados de expressão reais, mas não conseguimos encontrar dados da maneira como o programa necessita. As matrizes de expressão dos testes foram geradas aleatoriamente. Assim, 0s e 1s aleatórios foram gerados, de forma a simular os níveis de expressão obtidos em experimentos. Dessa forma, pode-se escolher o número de genes e o número de experimentos desejados. Essas matrizes de 0s e 1s aleatórios foram dadas como entrada para nossos programas. Em geral, como o número de experimentos (linhas da matriz) é muito menor do que o número de genes (colunas), obtivemos transversais de tamanho pequeno (de 1 a 3, e algumas vezes 4). À medida que aumentamos o número de linhas da matriz, a probabilidade de encontrar uma transversal de tamanho maior aumenta.

Utilizamos um sistema de processamento paralelo do tipo *Beowulf* do Instituto de Matemática e Estatística da USP. O objetivo do *Beowulf* do IME é fornecer uma plataforma de computação de alto desempenho para o desenvolvimento de ferramentas computacionais utilizadas em aplicações da bioinformática. Por isso, o sistema ganhou o nome de *Biowulf/IME*.

O Biowulf/IME é formado por 16 AMD PCs, conectados por uma chave (*switch*) Fast Ethernet 100 Mbit/seg. O *Node1* é o servidor e seu nome é tiramisu.ime.usp.br para acesso externo. O sistema está conectado à rede do IME via Ethernet 100 Mbit/seg. Cada AMD PC é basicamente configurado como segue [11]:

- Processador: 1.2 GHz AMD Thunderbird Athlon, 256 KB L2 cache.
- Memória: 768 MB PC133 SDRAM
- Disco: 30.73 GB ATA100 7200 RPM HD Deskstar 756GXP DLT-307030 IBM.
- Sistema Operacional: Debian Linux 2.2.19.

Utilizamos o compilador GNU gcc 2.95.2-13 e o software MPICH 1.2.1.

5.2 Matrizes Geradas e Resultados Obtidos

Nesta seção, mostramos os tempos, em segundos, obtidos pelos programas paralelo e seqüencial tendo como entrada algumas matrizes geradas aleatoriamente. Cada subseção mostra os tempos dos programas para um certo número de genes (colunas), para que possamos observar como o programa comporta-se em cada situação. Para

cada quantidade de genes, temos dois diferentes números de experimentos (linhas), 20 e 40. Cada um dos tempos mostrados nas tabelas e nos gráficos representa a médias entre 5 tempos obtidos. O resultado mais importante a ser observado é o que diz respeito à diferença entre os programas seqüencial e paralelo. Podemos notar que a partir de 1025 genes, o programa paralelo tende a ser executado em muito menos tempo do que o seqüencial. Para verificar a correção da resposta do programa paralelo, sempre tivemos o cuidado de comparar sua resposta à do programa seqüencial. Esperamos que o resultado de nosso trabalho possa ser apreciado com clareza nas subseções seguintes.

5.2.1 Matrizes de 513 colunas

Nesta subseção, mostramos os tempos de execução, em segundos, obtidos para duas matrizes de entrada geradas aleatoriamente, as duas com 513 colunas. Uma delas com 40 linhas e outra com 20 linhas. Como as matrizes foram obtidas aleatoriamente, não podíamos controlar o tamanho da transversal encontrada pelos programas. Para a matriz de 40×513, o tamanho da transversal encontrada é 3 e para a matriz de 20×513, a transversal encontrada tem tamanho 2.

Podemos ver na Tabela 5.1 os tempos de execução obtidos para os programas paralelo e seqüencial. Nessa tabela e na Figura 5.1 observa-se que o programa paralelo, quando utiliza 2 processadores, já consegue um tempo de execução menor do que o tempo de execução do seqüencial. Com 4 processadores, tendo como entrada a matriz de 40 linhas, o programa paralelo executa em menos tempo do que o programa seqüencial e do que o programa paralelo com 2 processadores. Porém, tendo como entrada a matriz de 20 linhas, o programa paralelo com 4 processadores leva mais tempo do que utilizando apenas 2 processadores. Isso se deve ao fato da quantidade de dados da entrada (20×513) não ser grande o suficiente para compensar a comunicação entre 4 processadores. Novamente, a quantidade de dados insuficiente para compensar a comunicação entre vários processadores, mas agora para todas as matrizes, prejudica a execução do programa paralelo utilizando 8 processadores. Note que nesse caso o tempo de execução do programa paralelo rodando em 8 processadores é pior do que os tempos usando 2 e 4 processadores e também é pior do que os tempos de execução do programa seqüencial.

Na Tabela 5.2 e na Figura 5.2, podemos ver as acelerações para o programa tendo como entrada as mesmas matrizes. Essas acelerações foram obtidas a partir dos tempos da Tabela 5.1. Verificamos que a aceleração obtida diminui em relação à aceleração esperada à medida que o número de processadores e conseqüentemente a comunicação aumentam. Note que a curva da aceleração do programa paralelo para a matriz de 40 linhas cai com a utilização de 8 processadores. Tendo como entrada a matriz de 20 linhas, a curva da aceleração do programa paralelo aparece sempre na descendente, indicando um péssimo rendimento do programa paralelo para essa matriz. Esse comportamento também é reflexo da quantidade de dados da entrada

não conseguir compensar adequadamente o custo de comunicação. Acreditamos que tendo como entrada matrizes maiores, essa aceleração possa crescer.

Processadores	1	2	4	8
40×513	0.022535	0.011722	0.008050	0.023135
20×513	0.005558	0.003352	0.005375	0.009462

Tabela 5.1: Tabela de tempos para uma matriz de entrada de 513 colunas.

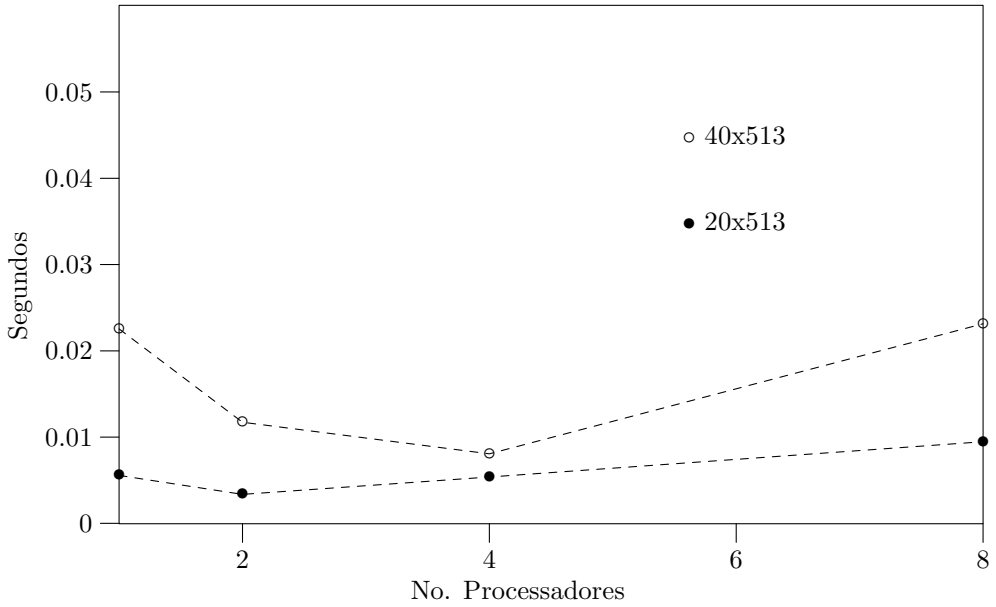


Figura 5.1: Curvas dos tempos observados para matriz de entrada de 513 colunas.

Processadores	2	4	8
40×513	1.92	2.80	0.97
20×513	1.66	1.03	0.59

Tabela 5.2: Tabela da aceleração (*speedup*) obtida dos tempos da Tabela 5.1.

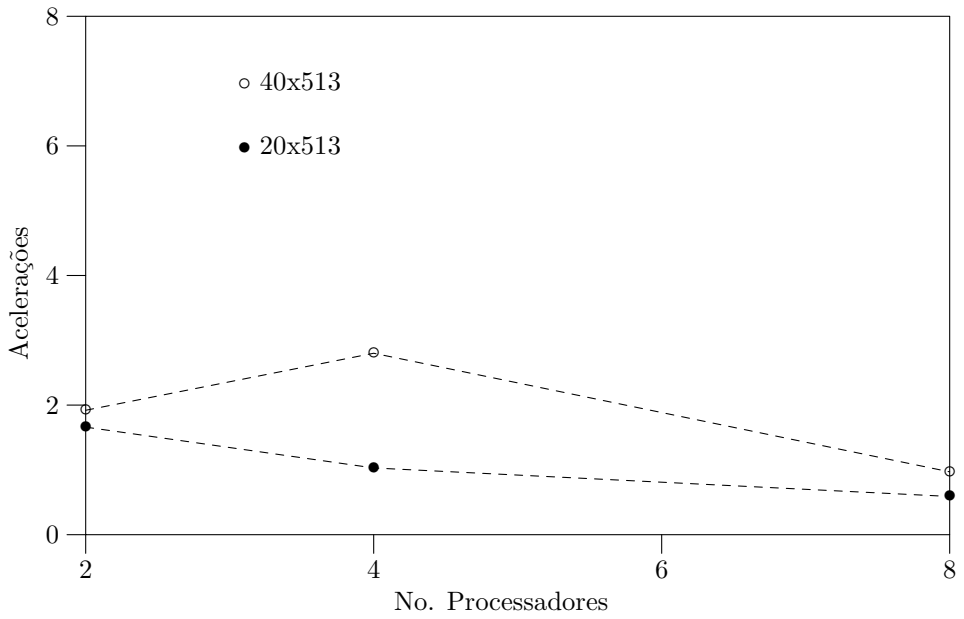


Figura 5.2: Acelerações (*speedups*) obtidas dos tempos da Tabela 5.1.

5.2.2 Matrizes de 1025 colunas

Nesta subseção, mostramos os tempos de execução, em segundos, obtidos pelos programas paralelo e seqüencial, tendo como entrada duas matrizes, geradas aleatoriamente. Cada uma delas contém 1025 colunas, sendo que uma contém 40 linhas e a outra 20 linhas. Para ambas as matrizes, os programas encontraram transversais de tamanho 2.

Mostramos na Tabela 5.3 os tempos de execução dos programas seqüencial e paralelo, tendo como entrada essas matrizes. Observe na tabela e na Figura 5.3 que para a matriz de entrada de 40×1025, quando o programa paralelo utiliza 8 processadores, seu tempo de execução é menor do que os tempos de execução do seqüencial e do paralelo com 4 e com 2 processadores, o que não acontecia com as matrizes de 513 colunas. Isso acontece devido ao fato de que a quantidade de dados aumentou, compensando os custos de comunicação entre os 8 processadores. Note que o mesmo não ocorreu para o programa paralelo com 8 processadores tendo como entrada a matriz de 20×1025, para o qual se observa um tempo de execução pior até do que o tempo de execução do seqüencial.

Na Tabela 5.4 e na Figura 5.4 podemos observar que, em sua maioria, as acelerações do programa paralelo melhoraram sensivelmente, se comparadas com as acelerações do mesmo programa tendo como entrada as matrizes de 513 colunas, mostradas na subseção anterior. Isso também se deve ao aumento da quantidade de dados da entrada. Note que a aceleração para o programa paralelo aumentou tendo como entrada a matriz de 20 linhas, mas sua curva continua na descendente para

mais do que 2 processadores. Isso significa que a quantidade de dados ainda não é grande o suficiente para compensar o uso de muitos processadores. Também por causa do aumento da quantidade de dados de entrada, note que as acelerações para o programa paralelo com 4 e com 8 processadores aumentaram, mas ainda estão longe das acelerações ideais.

Processadores	1	2	4	8
40×1025	0.044669	0.023399	0.016953	0.009030
20×1025	0.009425	0.005203	0.007320	0.010311

Tabela 5.3: Tabela de tempos para uma matriz de entrada de 1025 colunas.

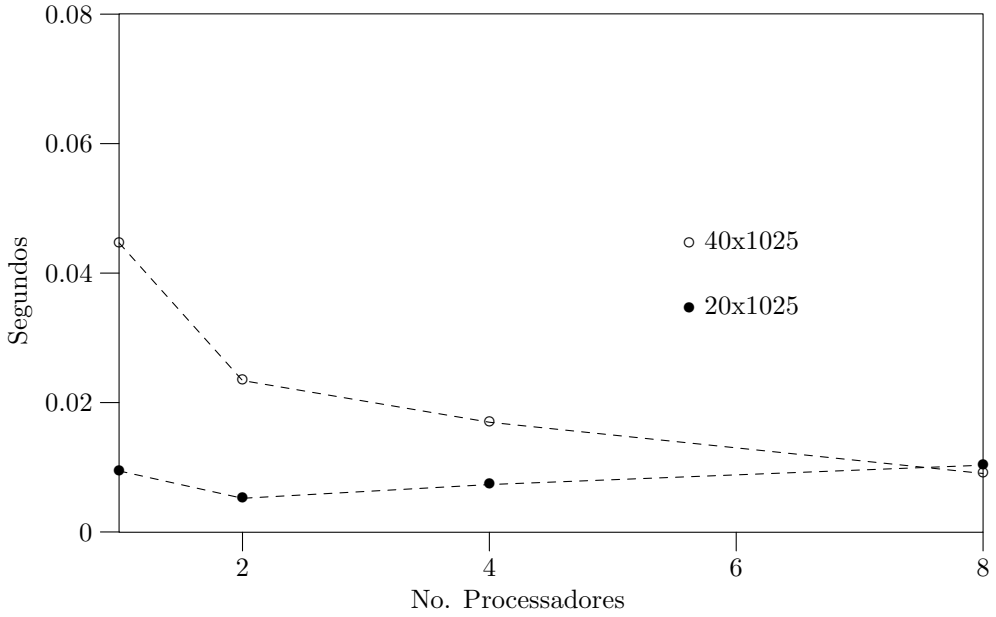


Figura 5.3: Curvas dos tempos observados para matriz de entrada de 1025 colunas.

Processadores	2	4	8
40×1025	1.91	2.63	4.95
20×1025	1.81	1.29	0.91

Tabela 5.4: Tabela da aceleração (*speedup*) obtida dos tempos da Tabela 5.3.

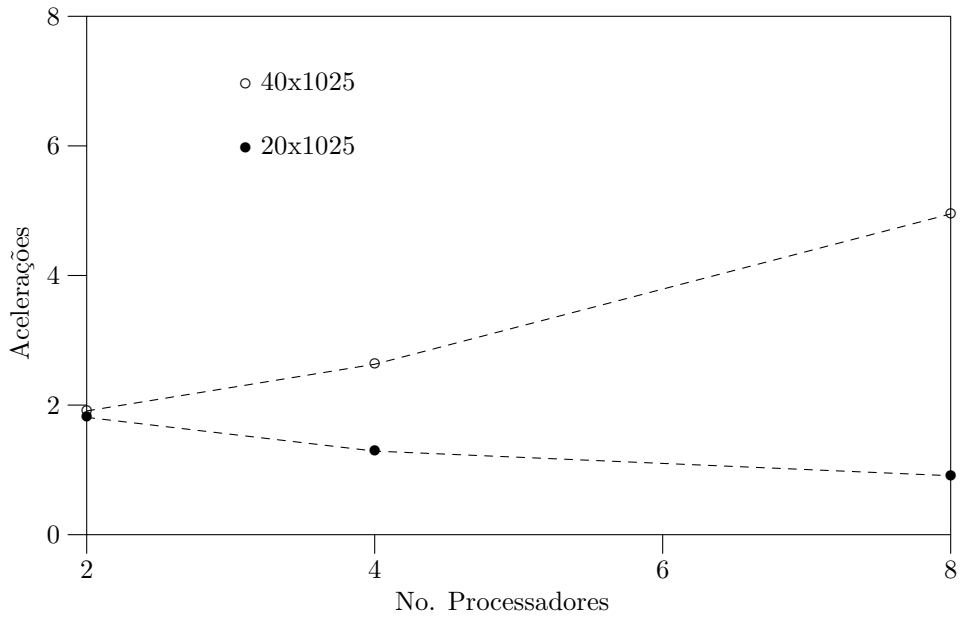


Figura 5.4: Acelerações (*speedups*) obtidas dos tempos da Tabela 5.3

5.2.3 Matrizes de 2049 colunas

Nesta subseção, mostramos os tempos de execução dos programas paralelo e seqüencial, tendo como entrada duas matrizes de 2049 colunas, geradas aleatoriamente. Novamente, uma delas possui 40 linhas e a outra possui 20 linhas. Os tamanhos das transversais encontradas pelos programas para as matrizes de 40×2049 e 20×2049 são, respectivamente, 3 e 2.

Na Tabela 5.5 podemos ver os tempos de execução para os programas, tendo como entrada as 2 matrizes. Note pela tabela e pela Figura 5.5 que aumentando o número de processadores, os tempos de execução do programa paralelo diminuem, exceto no caso da matriz de 20×2049 ser dada como entrada para o programa paralelo executado em 8 processadores. Nesse caso, o tempo de execução paralelo com 8 processadores é pior do que os tempos de execução com 4 e 2 processadores. Isso acontece novamente porque a quantidade de dados da entrada (20×2049) ainda é insuficiente para compensar a comunicação entre 8 processadores, mas podemos ver pela Tabela 5.6 que a aceleração nesse caso de 20 linhas vem aumentando progressivamente com o aumento do número de colunas da matriz, como acontece com a outra matriz, a diferença é que esta possui menos linhas, fazendo com que a quantidade de dados seja ainda insuficiente para compensar a comunicação. Observe como a curva da aceleração para a matriz de 20 linhas chega a "subir" com 4 processadores, para depois voltar a "cair" com 8. Podemos esperar então que o aumento no número de colunas faça a aceleração aproximar-se ainda mais da ideal.

Ainda na Tabela 5.6 e na Figura 5.6, podemos ver que as acelerações em geral

aumentaram sensivelmente, comparadas às acelerações do programa para as matrizes de 1025 colunas. Note que as acelerações para o programa paralelo utilizando 2 processadores aproximam-se bastante das ideais.

Processadores	1	2	4	8
40×2049	0.090763	0.045753	0.035213	0.027080
20×2049	0.023637	0.012206	0.009539	0.012896

Tabela 5.5: Tabela de tempos para uma matriz de entrada de 2049 colunas.

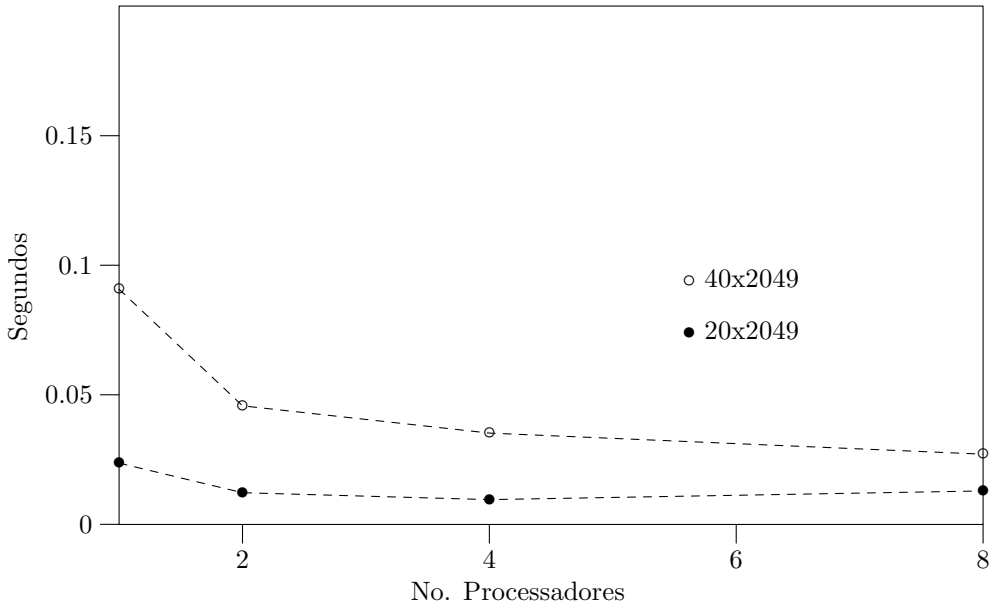


Figura 5.5: Curvas dos tempos observados para matriz de entrada de 2049 colunas.

Processadores	2	4	8
40×2049	1.98	2.58	3.35
20×2049	1.94	2.48	1.83

Tabela 5.6: Tabela da aceleração (*speedup*) obtida dos tempos da Tabela 5.5

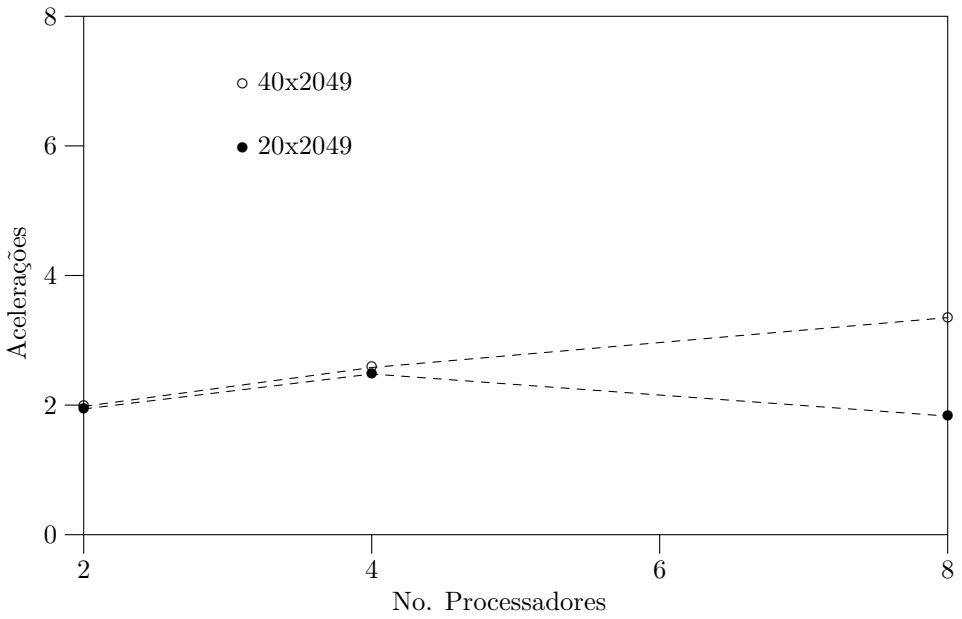


Figura 5.6: Acelerações (*speedups*) obtidas dos tempos da Tabela 5.5

5.2.4 Matrizes de 4097 colunas

Nesta subseção, apresentamos os tempos de execução dos programas paralelo e seqüencial, tendo como entrada duas matrizes de 4097 colunas. Uma delas possui 40 linhas e a outra possui 20 linhas. Os tamanhos das transversais encontradas para as matrizes de 40×4097 e 20×4097 são, respectivamente, 3 e 1. Observe na Tabela 5.7 os tempos de execução para os programas tendo como entrada essas matrizes. Podemos observar agora na tabela e na Figura 5.7 que o programa paralelo, quando utiliza 8 processadores, tem um tempo de execução melhor do que quando utiliza apenas 2 e 4 processadores, inclusive para a matriz de 20 linhas.

Na Tabela 5.8 podem ser vistas as acelerações obtidas pelo programa paralelo. Essa tabela e a Figura 5.8 mostram acelerações muito boas para todas as matrizes. Observe que o programa paralelo quando utiliza 2 processadores apresenta acelerações aparentemente super-lineares tendo qualquer uma das matrizes como entrada. Isso também acontece no caso do programa paralelo utilizando 4 processadores tendo como entrada a matriz de 20×4097. Para que pudéssemos entender esse resultado, consideramos quais são suas possíveis causas. A primeira hipótese é de que ele seria causado por "swap" de memória feito pelo programa seqüencial. Nesse caso, o tempo do programa seqüencial sofreria prejuízo, o que não aconteceria com o programa paralelo, que manipula uma quantidade menor de dados, diminuindo assim a necessidade de "swap" de memória. Como a memória de cada processador é de 768 MB e temos aproximadamente 20×2049 dados armazenados em forma de caractere, concluímos que a hipótese de "swap" de memória poderia ser descartada,

já que a quantidade de dados manipulada pelo programa não é grande o suficiente para provocá-lo. A segunda hipótese é o uso mais eficiente do *cache* por parte do programa paralelo. Como no programa paralelo os processadores possuem uma quantidade menor de dados, a probabilidade de que esses processadores utilizem os dados armazenados no *cache* é maior do que a probabilidade de que o único processador rodando o programa seqüencial o faça. Ao contrário, o programa seqüencial trabalha com uma quantidade maior de dados, encontrando menos dados necessários no *cache*. Portanto, devido à utilização mais intensa do *cache*, o programa paralelo torna-se mais eficiente do que o esperado. Acreditamos que essa segunda hipótese seja a causa mais provável das acelerações super-lineares.

Essas acelerações super-lineares também acontecem nas próximas matrizes, de 8193 colunas. Para essas matrizes, acreditamos que a causa esteja igualmente relacionada ao uso do *cache*.

Processadores	1	2	4	8
40×4097	0.207115	0.094005	0.060246	0.042911
20×4097	0.051221	0.024299	0.012244	0.006741

Tabela 5.7: Tabela de tempos para uma matriz de entrada de 4097 colunas.

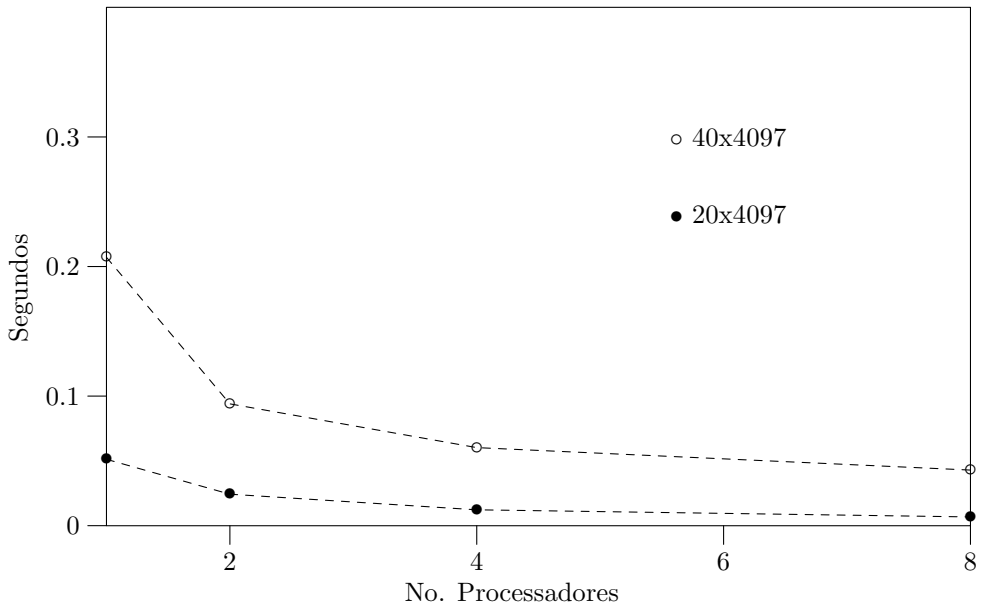


Figura 5.7: Curvas dos tempos observados para matriz de entrada de 4097 colunas.

Processadores	2	4	8
40×4097	2.20	3.44	4.83
20×4097	2.11	4.18	7.60

Tabela 5.8: Tabela da aceleração (*speedup*) obtida dos tempos da Tabela 5.7.

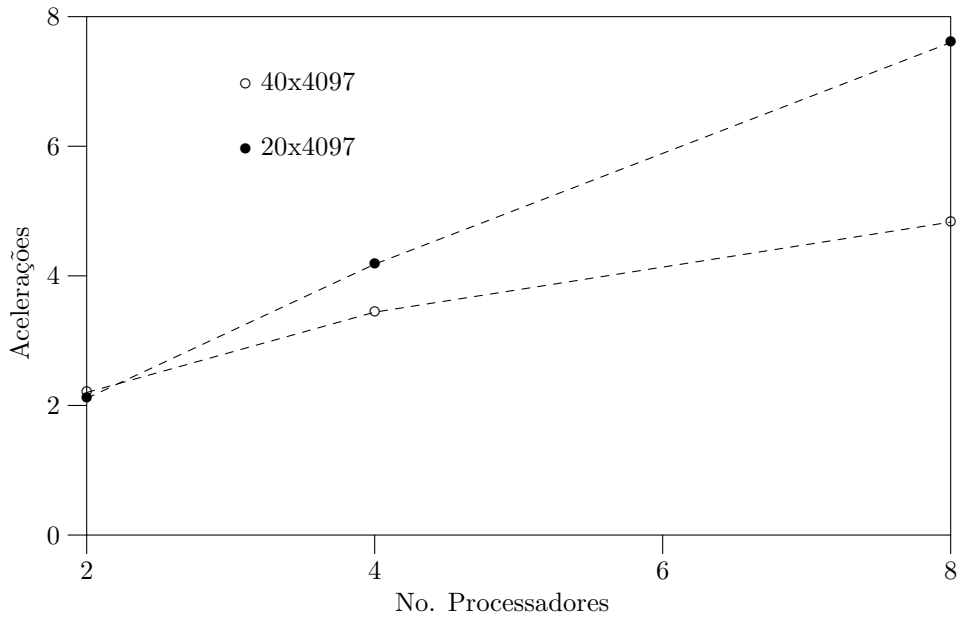


Figura 5.8: Acelerações (*speedups*) obtidas dos tempos da Tabela 5.7.

5.2.5 Matrizes de 8193 colunas

Nesta subseção, apresentamos os tempos de execução dos programas paralelo e seqüencial, tendo como entrada duas matrizes de 8193 colunas. Uma delas possui 40 linhas e a outra possui 20 linhas, como nas subseções anteriores. Os tamanhos das transversais encontradas para as matrizes de 40×8193 e 20×8193 são, respectivamente, 3 e 1. Observe na Tabela 5.9 os tempos de execução para os programas tendo como entrada essas matrizes. Na tabela e na Figura 5.9 podemos observar que quanto maior é o número de processadores, menor é o tempo de execução do programa.

Na Tabela 5.10 podem ser vistas as acelerações obtidas pelo programa paralelo. Esta tabela e a Figura 5.10 mostram acelerações muito boas para todas as matrizes. Note que as acelerações do programa paralelo utilizando 8 processadores aproximam-se das ideais. Isso se deve ao fato da quantidade de dados de entrada compensar a comunicação entre 8 processadores. Observe que o programa paralelo quando utiliza 2 e 4 processadores apresenta acelerações aparentemente super-lineares tendo

qualquer uma das matrizes como entrada. Novamente, acreditamos que a causa dessas acelerações super-lineares está relacionada ao uso do *cache*, como explicamos nas subseções anteriores.

Processadores	1	2	4	8
40×8193	0.505610	0.217824	0.101570	0.064036
20×8193	0.119981	0.052720	0.024691	0.022667

Tabela 5.9: Tabela de tempos para uma matriz de entrada de 8193 colunas.

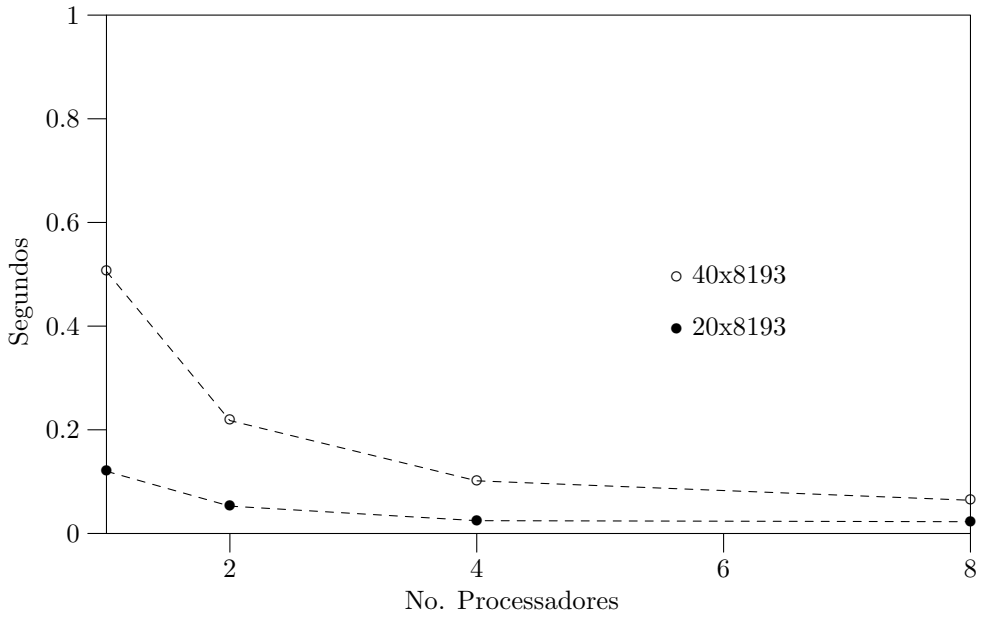


Figura 5.9: Curvas dos tempos observados para matriz de entrada de 8193 colunas.

Processadores	2	4	8
40×8193	2.32	4.98	7.90
20×8193	2.28	4.86	5.29

Tabela 5.10: Tabela da aceleração (*speedup*) obtida dos tempos da Tabela 5.8.

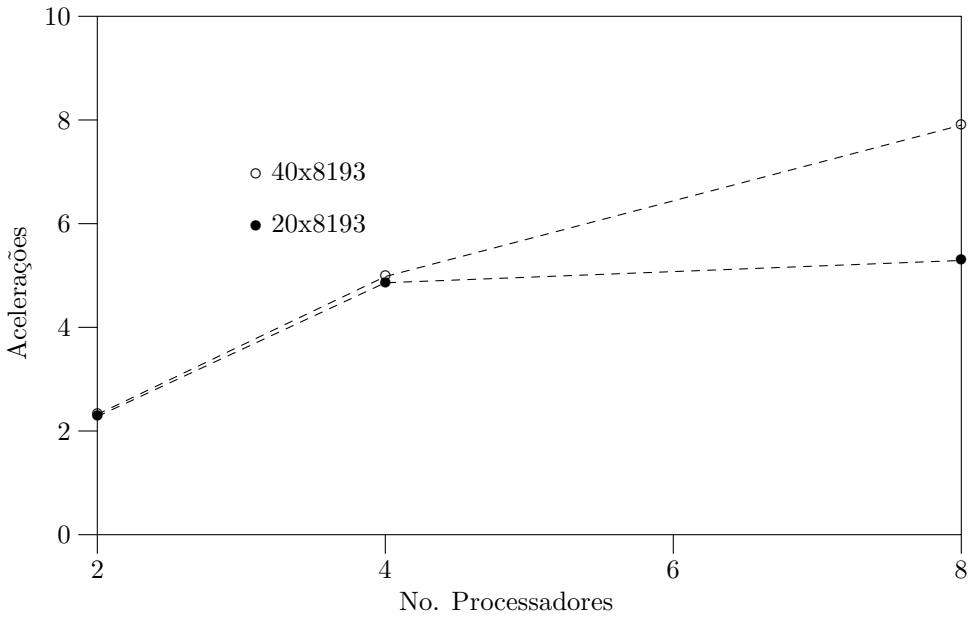


Figura 5.10: Acelerações (*speedups*) obtidas dos tempos da Tabela 5.8.

Capítulo 6

Conclusão

Atualmente são grandes os esforços dos cientistas da área de biologia molecular na tentativa de desvendar a forma como os genes atuam nos organismos. As perspectivas futuras são as mais otimistas possíveis tamanha é a quantidade e qualidade das pesquisas nessa área. Os *DNA-microarrays* trouxeram a possibilidade de observar o comportamento de milhares de genes simultaneamente e as interações entre eles. É natural que quando se fale em milhares de genes, logo se pense no desenvolvimento de técnicas computacionais, como as estudadas neste trabalho, que buscam solucionar esses problemas.

No início deste trabalho, tivemos a oportunidade de entender o problema da inferência de relacionamentos entre os genes. Através desses estudos, compreendemos que o problema da inferência de redes genéticas é extremamente importante para o início do entendimento sobre as funções dos genes. Entendemos também que essa inferência é feita através do desenvolvimento de modelos, como o modelo de rede booleana que escolhemos pesquisar. Os estudos a respeito do paralelismo foram importantes para conhecer melhor os conceitos necessários para o desenvolvimento de um algoritmo paralelo, como por exemplo o modelo CGM que nos propusemos a utilizar. O modelo PRAM é um dos modelos paralelos mais conhecidos e utilizados, mas as acelerações teóricas obtidas para os algoritmos desenvolvidos nesse modelo não são necessariamente obtidas também na prática. Por esse motivo, surgiram os modelos realísticos, como o BSP e o CGM, com a proposta de modelar um algoritmo para que ele obtivesse resultados próximos tanto na teoria quanto na prática. O modelo CGM é mais simples do que o modelo BSP porque simplifica o custo de comunicação.

Estudamos dois algoritmos de inferência de relacionamentos causais entre os genes de uma rede: o REVEAL e o *Predictor*. Com esse estudo, procuramos alguma base para compará-los, de forma que pudéssemos escolher um deles para então paralelizá-lo. O REVEAL, por ser um dos primeiros algoritmos desenvolvidos com essa finalidade, teve uma importância bastante grande em nossos estudos, especial-

mente no sentido de entender a dificuldade do problema. Conhecemos a teoria da informação como base para o desenvolvimento desse algoritmo. O *Predictor* é um algoritmo mais simples e elegante, e nos deu a oportunidade de ver uma aplicação do problema da transversal mínima para a solução de um problema biológico. Após esses estudos, optamos por paralelizar o *Predictor* pois, além de ser mais claro e objetivo do que o REVEAL, ele nos daria a oportunidade de atacar um problema importante, que é o problema da transversal mínima.

Como o *Predictor* é um algoritmo que necessita de uma solução para o problema da transversal mínima, nos baseamos em um algoritmo de aproximação para a sua solução, no intuito de desenvolver um algoritmo seqüencial. Podíamos escolher um entre dois algoritmos de aproximação muito conhecidos para o problema da transversal mínima. Um deles utiliza o método de aproximação primal-dual e o outro utiliza o método guloso. Baseados no problema da inferência de relacionamentos entre genes, escolhemos o algoritmo guloso para fazer parte de nosso novo algoritmo. Nosso algoritmo seqüencial desenvolvido tem complexidade de tempo $O(km^2n)$ (ou $O(m^4n)$), onde k é o tamanho da transversal encontrada pelo programa, $m \times n$ é o tamanho da matriz de expressão dada como entrada. Além disso, nosso algoritmo seqüencial tem razão de aproximação $ln|S| + 1$, onde S é a coleção de subconjuntos construída pelo *Predictor* em seu primeiro passo.

Baseados no algoritmo de aproximação seqüencial, desenvolvemos um algoritmo de aproximação no modelo CGM para encontrar as dependências de um gene, utilizando p processadores. O algoritmo CGM desenvolvido tem complexidade de computação $O(\frac{km^2n}{p})$ (ou $O(\frac{m^4n}{p})$), de espaço local $O(m^2n)$ e $O(k)$ (ou $O(m^2n)$) rodadas de comunicação. Não encontramos na literatura outros algoritmos paralelos para resolver o problema, por isso não houve como fazer comparações. Implementamos os algoritmos seqüencial e paralelo em uma máquina do tipo *Beowulf*, com 8 processadores disponíveis.

Simulamos matrizes de expressão através da geração aleatória de 0s e 1s. Tendo como entrada essas matrizes, o programa paralelo apresentou tempos muito bons se comparados aos tempos do programa seqüencial, especialmente para matrizes com mais de 1025 colunas. Acelerações muito boas para todas as quantidades de processadores, 2, 4 e 8, foram obtidas para matrizes com 4097 colunas ou mais.

A contribuição de nosso trabalho é mostrar que nosso algoritmo comporta-se muita bem na prática, como esperávamos pelos resultados teóricos, validando o modelo CGM. Acreditamos que o nosso algoritmo também é um passo inicial, cujos resultados experimentais encorajam a pesquisa em algoritmos paralelos para resolver de forma mais eficiente problemas de inferência de relacionamentos entre genes de um organismo.

Apêndice A

Código Fonte

Neste apêndice apresentamos os códigos dos programas seqüencial e paralelo. O programa seqüencial implementa o algoritmo DepAproxG, presente no Capítulo 4, enquanto que o programa paralelo implementa o algoritmo DepAproxGParalela, também presente no Capítulo 4. Ambos os programas foram escritos utilizando a linguagem C e a biblioteca MPI.

A.1 O Programa Seqüencial

```
//-------------------------------------------------------------/
// Programa: DepAproxG.c
// Programadora: Danielle Passos de Ruchkys e Siang Wun Song
// Data: 29/07/2002
// Este programa le uma matriz de um arquivo, que representa
// os niveis de expressao de varios genes em varios experimentos
// e determina, para um dado gene, quais os genes dos quais ele
// depende, possibilitando assim, que sejam descobertas as
// ligacoes de uma rede genetica. Para resolver esse problema
// nos baseamos em um algoritmo sequencial desenvolvido por [15],
// que tem como base resolver o problema da transversal minima.
//-------------------------------------------------------------/

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// Estrutura para a representacao dos genes.
typedef struct{
```

```
  int ocorrencia;
  int qCjts;
  int *listaCjts;
}gene;

// Estrutura para a representacao dos elementos de uma lista.
typedef struct elemento{
  int indice;
  struct elemento *prx;
}elemCjt;

enum booleano{falso,verdadeiro};

//Estrutura para a representacao dos conjuntos.
typedef struct{
  int i1, j1;
  enum booleano coberto;
  struct elemento *listaGenes;
}cjt;

// Funcoes:
char **Leitura(char *nomArq, int *qLin, int *qCol, char **coluna);

void *MontaEstruturas(cjt *X, int *tamX, gene *K, char **Mat,
                      char *v,int tamK, int qLin);

enum booleano Difere1(char a, char b);

enum booleano Difere2(char a, char b);

void Insere(elemCjt **lista, int indice);

elemCjt *HittingSet(cjt *X, gene *K, int tamX, int tamK);

int EncontraMaior1(gene *vetor, int tamanho);

// Inicio da funcao principal
int main(int argc, char *argv[])
{
  int meuId;          // numero de identificacao do processo.
  int nProcs;         // numero de processadores participantes.
  int nLinhas;        // numero de linhas da matriz de entrada.
  int nColunas;       // numero de colunas da matriz de entrada.
  int i, qS;
  int colDivisao;
```

```
double tInicial, tFinal, tTotal;
char **M, *v;
gene *E;            // ponteiro para um vetor de genes.
cjt *S;             // ponteiro para uma colecao(vetor) de
                    // conjuntos.
elemCjt *HS=NULL, *HSAux;

struct elemento *list;

// Pedir ao sistema que faca o necessario para iniciar o MPI.
MPI_Init(&argc, &argv);
// Saber quantos processadores estao trabalhando.
MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
// Pegar a identificacao do processador
MPI_Comm_rank(MPI_COMM_WORLD, &meuId);

if(nProcs != 1){
  printf("Rodar com apenas 1 processador");
  MPI_Finalize();
  exit(1);
}

if(argc != 2){
  printf("Uso incorreto: DepAproxG nome_do_arquivo");
  exit(1);
}

M = Leitura(argv[1], &nLinhas, &nColunas, &v);

// Inicio da contagem do tempo
MPI_Barrier(MPI_COMM_WORLD);
tInicial = MPI_Wtime();

// O vetor E de informacoes sobre os genes tera tamanho
// nColunas-1
E = (gene *)calloc(nColunas-1, sizeof(gene));
// O vetor S terah tamanho (nLinhas*nLinhas).
S = (cjt *)calloc((nLinhas*nLinhas),sizeof(cjt));

MontaEstruturas(S, &qS, E, M, v, nColunas-1, nLinhas);

HS = HittingSet(S, E, qS, nColunas-1);

HSAux = HS;
printf("RESPOSTA:");
```

```
  while(HS != NULL){
    printf(" -- %d --", HS->indice);
    HS = HS->prx;
  }

  for(i=0;i<nLinhas;i++)
    free(M[i]);
  free(M);
  free(v);
  free(E);
  free(S);
  free(HSAux);

  // Fim da contagem do tempo
  tFinal = MPI_Wtime();

  tTotal = (tFinal - tInicial);
  printf("TEMPO TOTAL: %lf", tTotal);

  MPI_Finalize();

  return 0;
}

// Funcao Leitura: esta funcao le a matriz de entrada do
// arquivo fornecido.
//-----------------------------------------------------------------
char **Leitura(char *nomArq, int *qLin, int *qCol, char **coluna)
{
  FILE *fp;
  char **mat, aux;
  int i, j;

  // Tenta abrir o arquivo para leitura
  if((fp = fopen(nomArq,"r"))==NULL){
    printf("O arquivo nao pode ser aberto.");
    exit(1);
  }

  // Leitura do numero de linhas
  fscanf(fp,"%d",qLin);
  // Leitura do numero de colunas
  fscanf(fp,"%d",qCol);

  // Aloca as linhas da matriz
```

```
  mat = (char **)calloc((*qLin),sizeof(char *));
  // Aloca o vetor que armazena os niveis de expressao do gene
  // estudado
  (*coluna) = (char *)calloc((*qLin),sizeof(char));

  // Aloca as colunas da matriz
  for(i=0;i<(*qLin);i++){
    mat[i] = (char *)calloc((*qCol),sizeof(char));
    if(!mat[i]){
      printf("Erro na alocacao de memoria");
      exit(1);
    }
  }

  // Leitura da matriz
  aux = getc(fp);
  for(i=0;i<(*qLin);i++){
    for(j=0;j<(*qCol);j++){
      aux = getc(fp);
      if(j == (*qCol)-1)
        *((*coluna)+i) = aux;
      else
        mat[i][j] = aux;
      aux = getc(fp);
    }
  }

  // Tenta fechar o arquivo
  if(fclose(fp))
    printf("Erro ao fechar o arquivo");

  // Retorna a matriz lida do arquivo
  return(mat);
}

// Funcao MontaEstruturas: esta funcao constroi o vetor de
// genes e o vetor de conjuntos, de acordo com os niveis de
// expressao da matriz de entrada.
//-----------------------------------------------------------
void *MontaEstruturas(cjt *X, int *tamX, gene *K, char **Mat,
                      char *v, int tamK, int qLin)
{
  int i, j, k, auxiliar;

  // Para cada gene pelo qual e responsavel:
```

```
  for(i=0;i<tamK;i++){
    // inicialize com 0 suas ocorrencias.
    K[i].ocorrencia = 0;
    // inicialize com 0 o tamanho de sua lista de conjuntos
    // onde esta presente.
    K[i].qCjts = 0;
    // aloque espaco para armazenar a lista de conjuntos.
    K[i].listaCjts = (int *)calloc((qLin*qLin), sizeof(int));
  }

  // A principio o vetor de conjuntos do processador esta vazio.
  *tamX = 0;
  for(i=0;i<qLin;i++){
    for(j=i+1;j<qLin;j++){
      // se o nivel de expressao sofreu alteracao de uma linha
      // para a outra
      if(Difere1(v[i], v[j])){
        // surge um novo conjunto i,j
        X[*tamX].i1 = i;
        X[*tamX].j1 = j;
        X[*tamX].coberto = falso;
        // ainda nao tem nenhum gene que tem seu nivel de
        // expressao alterado entre i e j.
        X[*tamX].listaGenes = NULL;
        // verifica se existem outros genes que tambem tem seu
        // nivel de expressao alterado ou tem seu nivel de
        // expressao forcado a um valor alto ou baixo entre
        // i e j. Os genes encontrados serao elementos de
        // S[*tamS].listaGenes.
        for(k=0;k<tamK;k++){
          if(Difere2(Mat[i][k], Mat[j][k])){
            auxiliar = K[k].ocorrencia;
            K[k].ocorrencia++;
            K[k].qCjts++;
            K[k].listaCjts[auxiliar] = *tamX;
            Insere(&(X[*tamX].listaGenes), k);
          }
        }
        *tamX = *tamX + 1;
      }
    }
  }
}

// Funcao Difere1: verifica se o nivel de expressao difere,
```

```
// mas sem ser forcado a um valor alto ou baixo.
//-------------------------------------------------------------
enum booleano Difere1(char a, char b)
{
  if((a == '0' && b == '1') || (a == '1' && b == '0'))
    return verdadeiro;
  else
    return falso;
}

// Funcao Difere2: verifica se o nivel de expressao difere,
// ou foi forcado a um valor alto ou baixo.
//-------------------------------------------------------------
enum booleano Difere2(char a, char b)
{
  if(Difere1(a,b) == verdadeiro)
    return verdadeiro;
  else if((a == '0' && b == '+') || (a == '1' && b == '-'))
    return verdadeiro;
  else if((a == '+' && b == '0') || (a == '+' && b == '-'))
    return verdadeiro;
  else if((a == '-' && b == '1') || (a == '-' && b == '+'))
    return verdadeiro;
  else
    return falso;
}

// Funcao Insere: insere um elemento em uma lista ligada.
//-------------------------------------------------------------
void Insere(elemCjt **lista, int indice)
{
  elemCjt *auxiliar;

  if((*lista) == NULL){
    (*lista) = (elemCjt *)calloc(1, sizeof(elemCjt));
    (*lista)->indice = indice;
    (*lista)->prx = NULL;
  }
  else{
    auxiliar = (elemCjt *)calloc(1, sizeof(elemCjt));
    auxiliar->prx = (*lista);
    auxiliar->indice = indice;
    (*lista) = auxiliar;
  }
}
```

```
elemCjt *HittingSet(cjt *X, gene *K, int tamX, int tamK)
{
  int i, j, indMaior, indice, tamLista, *listaDono,
      controle = 0, z;
  elemCjt *resposta = NULL, *lista;


  while(tamX > 0){
    // Encontra o elemento(gene) de maior ocorrencia
    indMaior = EncontraMaior1(K, tamK);

    Insere(&resposta, indMaior);

    // Guarda a lista de conjuntos em que ele esta presente.
    listaDono = K[indMaior].listaCjts;

    // Guarda o tamanho dessa lista.
    tamLista = K[indMaior].qCjts;

    // Percorre a lista.
    while(tamLista > 0){
      // Toma um indice de X por vez.
      z = listaDono[controle];
      // Verifica se ainda nao esta coberto.
      if(X[z].coberto == falso){
        X[z].coberto = verdadeiro;
        tamX--;
        // Toma a lista de elementos do conjunto em questao.
        lista = X[z].listaGenes;
        while(lista){
          j = lista->indice;
          K[j].ocorrencia--;
          lista = lista->prx;
        }
      }
      tamLista--;
      controle++;
    }
    controle = 0;
  }
  return resposta;
}
```

```
// Funcao EncontraMaior1: esta funcao encontra o elemento
// de maior ocorrencia de um vetor de genes
int EncontraMaior1(gene *vetor, int tamanho)
{
  int i=1, indMaior = 0, maior = vetor[0].ocorrencia;

  while(i < tamanho){
    if(vetor[i].ocorrencia > maior){
      maior = vetor[i].ocorrencia;
      indMaior = i;
    }
    i++;
  }
  return indMaior;
}
```

A.2 O Programa Paralelo

```
//-------------------------------------------------------------/
// Programa: DepAproxGParalela.c
// Programadora: Danielle Passos de Ruchkys e Siang Song
// Data: 29/07/2002
// Este programa paralelo le uma matriz de um arquivo, que
// representa os niveis de expressao de varios genes em varios
// experimentos e determina, para um dado gene desejado, quais
// os genes dos quais ele depende, possibilitando assim, que
// sejam descobertas as ligacoes de uma rede genetica. Para
// resolver esse problema nos baseamos em um algoritmo
// sequencial desenvolvido por [15], que tem como base resolver
// aproximadamente o problema da transversal minima ou problema
// hitting set.
//-------------------------------------------------------------/

#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>

#define root 0

// Estrutura para a representacao dos genes.
typedef struct{
```

```
  int ocorrencia;
  int rotulo;
  int qCjts;
  int *listaCjts;
}gene;

// Estrutura para a representacao dos elementos de uma lista.
typedef struct elemento{
  int indice;
  int rotulo;
  struct elemento *prx;
}elemCjt;

enum booleano{falso,verdadeiro};

//Estrutura para a representacao dos conjuntos.
typedef struct{
  int i1, j1;
  enum booleano coberto;
  struct elemento *listaGenes;
}cjt;

// Estrutura para auxiliar na troca de mensagens
// entre os processadores.
typedef struct{
  int ocorrencia;
  int rotulo;
  int dono;
}auxiliar;

// Funcoes:
char **Leitura(char *nomArq, int *qLin, int *qCol, char **coluna);
void *MontaEstruturas(cjt *X, int *tamX, gene *K, char **Mat,
                      char *v, int tamK, int ident, int qLin);
enum booleano Difere1(char a, char b);
enum booleano Difere2(char a, char b);
void Insere(elemCjt **lista, int indice, int rotulo);
elemCjt *HittingSet(cjt *X, gene *K, int tamX, int tamK,
                    int ident, int qProcs);
void ConstroiTipoDado(auxiliar *indata,
                      MPI_Datatype *message_type_ptr);
int EncontraMaior1(gene *vetor, int tamanho);
int EncontraMaior2(auxiliar *vetor, int tamanho);

// Inicio da funcao principal
```

```
int main(int argc, char *argv[])
{
  int meuId;          // numero de identificacao do processo.
  int nProcs;         // numero de processadores participantes.
  int nLinhas;        // numero de linhas da matriz de entrada.
  int nColunas;       // numero de colunas da matriz de entrada.
  int i, qS;
  int colDivisao;
  double tInicial, tFinal, tTotal, tMaior;
  char **M, *v;
  gene *E;            // ponteiro para um vetor de genes.
  cjt *S;             // ponteiro para uma colecao(vetor)
                      // de conjuntos.
  elemCjt *HS=NULL, *HSAux;

  struct{
    double time;
    int rank;
  }in, out;

  // Pedir ao sistema que faca o necessario para iniciar o MPI.
  MPI_Init(&argc, &argv);
  // Saber quantos processadores estao trabalhando.
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
  // Pegar a identificacao do processador
  MPI_Comm_rank(MPI_COMM_WORLD, &meuId);

  // INICIO DA ENTRADA E DISTRIBUICAO DE DADOS
  // Se for o processador root, le a entrada
  if(meuId == root){
    if(argc != 2){
      printf("Uso incorreto: DepAproxGParalela nome_do_arquivo");
      exit(1);
    }
    M = Leitura(argv[1], &nLinhas, &nColunas, &v);
  }

  // O processador root envia para todos os outros
  // processadores o numero de linhas da matriz
  MPI_Bcast(&nLinhas, 1, MPI_INT, root, MPI_COMM_WORLD);
  // o numero de colunas da matriz
  MPI_Bcast(&nColunas, 1, MPI_INT, root, MPI_COMM_WORLD);

  // Todos calculam a quantidade de genes(colunas da matriz)
  // pelos quais cada processador sera responsavel.
```

```
colDivisao = (nColunas - 1)/nProcs;

// Se nao for o root devera:
if(meuId != root){
  // alocar espaco para as linhas do pedaco da matriz
  // que lhe cabe.
  M = (char **)calloc(nLinhas, sizeof(char *));
  // alocar espaco para a coluna referente ao gene estudado.
  v = (char *)calloc(nLinhas, sizeof(char));
}

for(i=0;i<nLinhas;i++){
  if(meuId != root){
    // Aloca espaco para seu pedaco da matriz.
    M[i] = (char *)calloc(colDivisao, sizeof(char));
  }
  // O processador root divide e envia pedacos das linhas
  // da matriz para os outros processadores, enquanto esses
  // recebem e armazenam os pedacos.
  MPI_Scatter(M[i], colDivisao, MPI_CHAR, M[i], colDivisao,
              MPI_CHAR, root, MPI_COMM_WORLD);
}

// O processador root envia tambem o vetor v para os outros
// processadores enquanto esses recebem.
MPI_Bcast(v, nLinhas, MPI_CHAR, root, MPI_COMM_WORLD);

// FIM DA ENTRADA E DISTRIBUICAO DE DADOS

// Inicio da contagem do tempo
// Sincronizacao dos processadores
MPI_Barrier(MPI_COMM_WORLD);
tInicial = MPI_Wtime();

// O vetor E de informacoes sobre os genes tera tamanho
// colDivisao em cada processador. Abaixo ele esta sendo
// alocado.
E = (gene *)calloc(colDivisao, sizeof(gene));
// O vetor S terah tamanho (nLinhas*nLinhas) em cada
// processador. Abaixo ele esta sendo alocado.
S = (cjt *)calloc((nLinhas*nLinhas),sizeof(cjt));

MontaEstruturas(S, &qS, E, M, v, colDivisao, meuId, nLinhas);

HS = HittingSet(S, E, qS, colDivisao, meuId, nProcs);
```

```
  HSAux = HS;
  if(meuId == root){
    printf("\nRESPOSTA:");
    while(HS != NULL){
      printf(" -- %d --", HS->rotulo);
      HS = HS->prx;
    }
  }

  for(i=0;i<nLinhas;i++)
    free(M[i]);
  free(M);
  free(v);
  free(E);
  free(S);
  free(HSAux);

  tFinal = MPI_Wtime();

  tTotal = (tFinal - tInicial);
  in.time = tTotal;
  in.rank = meuId;

  MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
             root, MPI_COMM_WORLD);

  if(meuId == root)
    printf("Tempo Total: %lf ", out.time);

  MPI_Finalize();

  return 0;
}

// Funcao Leitura: esta funcao le a matriz de entrada do arquivo
// fornecido.
//-----------------------------------------------------------------
char **Leitura(char *nomArq, int *qLin, int *qCol, char **coluna)
{
  FILE *fp;
  char **mat, aux;
  int i, j;

  // Tenta abrir o arquivo para leitura
```

```
  if((fp = fopen(nomArq,"r"))==NULL){
    printf("O arquivo nao pode ser aberto.");
    exit(1);
  }

  // Leitura do numero de linhas
  fscanf(fp,"%d",qLin);
  // Leitura do numero de colunas
  fscanf(fp,"%d",qCol);

  // Aloca as linhas da matriz
  mat = (char **)calloc((*qLin),sizeof(char *));
  // Aloca o vetor que armazena os niveis de expressao do gene
  // estudado
  (*coluna) = (char *)calloc((*qLin),sizeof(char));

  // Aloca as colunas da matriz
  for(i=0;i<(*qLin);i++){
    mat[i] = (char *)calloc((*qCol),sizeof(char));
    if(!mat[i]){
      printf("Erro na alocacao de memoria");
      exit(1);
    }
  }

  // Leitura da matriz
  aux = getc(fp);
  for(i=0;i<(*qLin);i++){
    for(j=0;j<(*qCol);j++){
      aux = getc(fp);
      if(j == (*qCol)-1)
        *((*coluna)+i) = aux;
      else
        mat[i][j] = aux;
      aux = getc(fp);
    }
  }

  // Tenta fechar o arquivo
  if(fclose(fp))
    printf("Erro ao fechar o arquivo");

  // Retorna a matriz lida do arquivo
  return(mat);
}
```

```
// Funcao MontaEstruturas: esta funcao constroi o vetor de genes
// e o vetor de conjuntos, de acordo com os niveis de expressao
// da matriz de entrada.
//-------------------------------------------------------------
void *MontaEstruturas(cjt *X, int *tamX, gene *K, char **Mat,
                      char *v, int tamK, int ident, int qLin)
{
  int i, j, k, auxiliar;

  // Para cada gene pelo qual e responsavel:
  for(i=0;i<tamK;i++){
    // calcula seu verdadeiro rotulo, onde rotulo e a
    // identificacao do gene entre todos os processadores.
    K[i].rotulo = (ident * tamK) + i;
    // inicialize com 0 suas ocorrencias.
    K[i].ocorrencia = 0;
    // inicialize com 0 o tamanho de sua lista de conjuntos
    // onde esta presente.
    K[i].qCjts = 0;
    // aloque espaco para armazenar a lista de conjuntos.
    K[i].listaCjts = (int *)calloc((qLin*qLin), sizeof(int));
  }

  // A principio o vetor de conjuntos do processador esta vazio.
  *tamX = 0;
  for(i=0;i<qLin;i++){
    for(j=i+1;j<qLin;j++){
      // se o nivel de expressao sofreu alteracao de uma linha
      // para a outra
      if(Difere1(v[i], v[j])){
        // surge um novo conjunto i,j
        X[*tamX].i1 = i;
        X[*tamX].j1 = j;
        X[*tamX].coberto = falso;
        // ainda nao tem nenhum gene que tem seu nivel de
        // expressao alterado entre i e j.
        X[*tamX].listaGenes = NULL;
        // verifica se existem outros genes que tambem tem seu
        // nivel de expressao alterado ou tem seu nivel de
        // expressao forcado a um valor alto ou baixo entre
        // i e j. Os genes encontrados serao elementos de
        // S[*tamS].listaGenes.
        for(k=0;k<tamK;k++){
          if(Difere2(Mat[i][k], Mat[j][k])){
```

```
            auxiliar = K[k].ocorrencia;
            K[k].ocorrencia++;
            K[k].qCjts++;
            K[k].listaCjts[auxiliar] = *tamX;
            Insere(&(X[*tamX].listaGenes), k, K[k].rotulo);
          }
        }
        *tamX = *tamX + 1;
      }
    }
  }
}

// Funcao Difere1: verifica se o nivel de expressao difere,
// mas sem ser forcado a um valor alto ou baixo.
//-------------------------------------------------------------
enum booleano Difere1(char a, char b)
{
  if((a == '0' && b == '1') || (a == '1' && b == '0'))
    return verdadeiro;
  else
    return falso;
}

// Funcao Difere2: verifica se o nivel de expressao difere, ou
// foi forcado a um valor alto ou baixo.
//-------------------------------------------------------------
enum booleano Difere2(char a, char b)
{
  if(Difere1(a,b) == verdadeiro)
    return verdadeiro;
  else if((a == '0' && b == '+') || (a == '1' && b == '-'))
    return verdadeiro;
  else if((a == '+' && b == '0') || (a == '+' && b == '-'))
    return verdadeiro;
  else if((a == '-' && b == '1') || (a == '-' && b == '+'))
    return verdadeiro;
  else
    return falso;
}

// Funcao Insere: insere um elemento em uma lista ligada.
//-------------------------------------------------------------
void Insere(elemCjt **lista, int indice, int rotulo)
{
```

```
  elemCjt *auxiliar;

  if((*lista) == NULL){
    (*lista) = (elemCjt *)calloc(1, sizeof(elemCjt));
    (*lista)->indice = indice;
    (*lista)->rotulo = rotulo;
    (*lista)->prx = NULL;
  }
  else{
    auxiliar = (elemCjt *)calloc(1, sizeof(elemCjt));
    auxiliar->prx = (*lista);
    auxiliar->indice = indice;
    auxiliar->rotulo = rotulo;
    (*lista) = auxiliar;
  }
}

elemCjt *HittingSet(cjt *X, gene *K, int tamX, int tamK,
                    int ident, int qProcs)
{
  int i, j, indMaior, indice, *listaDono, tamLista,
      controle = 0, z;
  auxiliar *comunica, *pacote;
  MPI_Datatype tipoCriado;
  elemCjt *resposta = NULL, *lista;

  ConstroiTipoDado(comunica, &tipoCriado);
  ConstroiTipoDado(pacote, &tipoCriado);

  if(ident == root){
    comunica = (auxiliar *)calloc(qProcs, sizeof(auxiliar));
  }

  pacote = (auxiliar *)calloc(1, sizeof(auxiliar));

  while(tamX > 0){

     // Encontra o elemento(gene) de maior ocorrencia
    indMaior = EncontraMaior1(K, tamK);

    // Empacota o maior encontrado
    pacote->ocorrencia = K[indMaior].ocorrencia;
    pacote->rotulo = K[indMaior].rotulo;
    pacote->dono = ident;
```

```
MPI_Gather(pacote, 1, tipoCriado, comunica, 1, tipoCriado,
           root, MPI_COMM_WORLD);

// Se for o processador root
if(ident == root){
  // Encontra o gene de maior ocorrencia entre os ja
  // encontrados pelos outros processadores.
  indice = EncontraMaior2(comunica, qProcs);
  // Insere tal gene na resposta.
  Insere(&resposta, indice, comunica[indice].rotulo);
  // Descobre qual o dono do gene.
  indice = comunica[indice].dono;
}

// Comunica o dono para os outros processadores.
MPI_Bcast(&indice, 1, MPI_INT, root, MPI_COMM_WORLD);

// Se for o processador dono.
if(ident == indice){
  // Guarda a lista de conjuntos em que ele esta presente.
  listaDono = K[indMaior].listaCjts;
  // Guarda o tamanho dessa lista.
  tamLista = K[indMaior].qCjts;
}

// O dono manda o tamanho de sua lista para todos os outros
// processadores.
MPI_Bcast(&tamLista, 1, MPI_INT, indice, MPI_COMM_WORLD);

// Se nao for o dono, aloca espaco para a lista do dono que
// sera recebida.
if(ident != indice)
  listaDono = (int *)calloc(tamLista, sizeof(int));

// O dono manda a lista para todos.
MPI_Bcast(listaDono, tamLista, MPI_INT, indice,
          MPI_COMM_WORLD);

// Percorre a lista.
while(tamLista > 0){
  // Toma um indice de X por vez.
  z = listaDono[controle];
  // Verifica se ainda nao esta coberto.
  if(X[z].coberto == falso){
    X[z].coberto = verdadeiro;
```

```
        tamX--;
        // Toma a lista de elementos do conjunto em questao.
        lista = X[z].listaGenes;
        while(lista){
          j = lista->indice;
          K[j].ocorrencia--;
          lista = lista->prx;
        }
      }
      tamLista--;
      controle++;
    }
    controle = 0;
    if(ident != indice)
      free(listaDono);
  }
  free(pacote);
  if(ident == 0)
    free(comunica);
  return resposta;
}

// Funcao ConstroiTipoDado: esta funcao constroi um novo tipo de
// dado para o MPI.
//-----------------------------------------------------------------
void ConstroiTipoDado(auxiliar *indata,
                      MPI_Datatype *message_type_ptr)
{
  int block_lengths[3];

  MPI_Aint displacements[3];
  MPI_Aint addresses[4];
  MPI_Datatype typelist[3];

  typelist[0] = MPI_INT;
  typelist[1] = MPI_INT;
  typelist[2] = MPI_INT;

  block_lengths[0] = block_lengths[1] = block_lengths[2] = 1;

  MPI_Address(indata, &addresses[0]);
  MPI_Address(&(indata->ocorrencia), &addresses[1]);
  MPI_Address(&(indata->rotulo), &addresses[2]);
  MPI_Address(&(indata->dono), &addresses[3]);
```

```
  displacements[0] = addresses[1] - addresses[0];
  displacements[1] = addresses[2] - addresses[0];
  displacements[2] = addresses[3] - addresses[0];

  MPI_Type_struct(3, block_lengths, displacements, typelist,
                  message_type_ptr);
  MPI_Type_commit(message_type_ptr);
}

// Funcao EncontraMaior1: esta funcao encontra o elemento de
// maior ocorrencia de um vetor de genes
int EncontraMaior1(gene *vetor, int tamanho)
{
  int i=1, indMaior = 0, maior = vetor[0].ocorrencia;

  while(i < tamanho){
    if(vetor[i].ocorrencia > maior){
      maior = vetor[i].ocorrencia;
      indMaior = i;
    }
    i++;
  }
  return indMaior;
}

// Funcao EncontraMaior2: esta funcao encontra o elemento de
// maior ocorrencia de um vetor auxiliar
int EncontraMaior2(auxiliar *vetor, int tamanho)
{
  int i=1, indMaior = 0, maior = vetor[0].ocorrencia;

  while(i < tamanho){
    if(vetor[i].ocorrencia > maior){
      maior = vetor[i].ocorrencia;
      indMaior = i;
    }
    i++;
  }
  return indMaior;
}
```

Referências Bibliográficas

[1] S. Aluru. http://www.ee.iastate.edu/∼aluru/cb.html.

[2] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms 2:198-203*, 1981.

[3] D. D. L. Bowtell. Options available – from start to finish – for obtaining expression data by microarray. *Nature Genetics Supplement*, 21:25–32, January 1999.

[4] P. O. Brown. Exploring the new world of the genome with DNA microarrays. *Nature Genetics Supplement*, 21:33–37, January 1999.

[5] E. N. Cáceres, H. Mongelli, and S. W. Song. Algoritmos paralelos usando cgm/pvm/mpi: uma introdução. In *Anais da XX Jornada de Atualização em Informática*. SBC, 2001.

[6] V. G. Cheung and *et al.* Making and reading microarrays. *Nature Genetics Supplement*, 21:15–19, January 1999.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1999.

[8] C. Debouck and P. N. Goodfellow. DNA microarrays in drug discovery and development. *Nature Genetics Supplement*, 21:48–50, January 1999.

[9] F. Dehne. Coarse grained parallel algorithms. *Algorithmica*, 24(3/4):173–176, 2000.

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

[11] M. X. T. Delgado. http://www.vision.ime.usp.br/∼cage/Beowulf/, 2001.

[12] P. D'haeseleer. *Reconstructing gene networks from large scale gene expression data*. PhD thesis, The University of New Mexico, 2000.

[13] P. D'haeseleer, S. Liang, and R. Somogyi. Gene expression data analysis and modeling. *Tutorial notes from Pacific Symposium on Biocomputing*, 1999.

[14] P. D'haeseleer, S. Liang, and R. Somogyi. Genetic network inference: From co-expression clustering to reverse engineering. *Bioinformatics 16(8):707-726*, 2000.

[15] D. J. Duggan, M. Bittner, Y. Chen, P. Meltzer, and J. M. Trent. Expression profiling using cDNA microarrays. *Nature Genetics Supplement*, 21:10–14, January 1999.

[16] M. B. Eisen and P. O. Brown. DNA arrays for analysis of gene expression. In S. M. Weissman, editor, *cDNA Preparation and Characterization*, volume 303 of *Methods in Enzymology*, pages 179–205. Academic Press, 1999.

[17] C. G. Fernandes, F. K. Miyazawa, M. R. Cerioli, and P. Feofiloff, editors. *Uma Introdução Sucinta a Algoritmos de Aproximação*. IMPA, 2001.

[18] M. R. Garey and D. S. Johnson. *Computers and intractability - A guide to the theory of NP-completeness*. W.H. Freeman and Company, 21th edition, 1999.

[19] D. S. Hochbaum. *Aproximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[20] T. E. Ideker, V. Thorsson, and R. M. Karp. Discovery of regulatory interactions through perturbation: inference and experimental design. *Pacific Symposium on Biocomputing 5:302-313*, 2000.

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

[22] S. Jha, O. Sheyner, and J. M. Wing. Minimization and reliability analyses of attack graphs. *IEEE Symposium on Security and Privacy*, 2002.

[23] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences 9:256-278*, 1974.

[24] E. S. Lander. Array of hope. *Nature Genetics Supplement*, 21:3–4, January 1999.

[25] S. Liang, S. Fuhrman, and R. Somogyi. Reveal, a general reverse engineering algorithm for inference of genetic network architectures. *Pacific Symposium on Biocomputing 3:18-29*, 1998.

[26] J. Meidanes and J. C. Setubal. *Introduction to Computacional Molecular Biology*. PWS Publishing Co., 1997.

[27] C. Y. Nasu. Algoritmos bsp/cgm para computação de circuitos de euler em grafos. Master's thesis, UFMS, 2002.

[28] D. P. Ruchkys and S. W. Song. A parallel aproximation hitting set algorithm for gene expression analysis. *14th Symposium on Computer Architecture and High Performance Computing 75-81*, 2002.

[29] D. P. Ruchkys and S. W. Song. A parallel solution to infer genetic network architectures in gene expression analysis. *International Journal of High Performance Computing Applications*, To appear.

[30] A. Silvescu and V. Honavar. Temporal boolean network models of genetic networks and their inference from gene expression time series. *Atlantic Symposium on Computational Biology, Genome Information Systems and Technology*, 2001.

[31] M. J. Simmons and D. P. Snustad. *Principles of genetics*. Wiley, 2nd edition, 1999.

[32] D. Thieffry and R. Thomas. Qualitative analysis of gene networks. *Pacific Symposium on Biocomputing 3:77-88*, 1998.

[33] J. Tsang. Gene expression, DNA arrays, and genetic networks. Unpublished, 1999.

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