Inferência de Redes de Regulação Gênica usando Computação Paralela Híbrida

Jean Carlo Wai Keung Ma

SERVIÇO DE PÓS-GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:_____

Inferência de Redes de Regulação Gênica usando Computação Paralela Híbrida[1]

Jean Carlo Wai Keung Ma

Orientador: *Prof. Dr. Marco Aurélio Stefanes*

UFMS - Campo Grande
abril/2017

[1]Trabalho Realizado com Auxílio do CNPq Proc. No: 131798/2014-6

Agradecimentos

Agradeço aos meus pais, Ma Yick On e Kwok Wing Hung Ma, por estarem sempre ao meu lado apoiando e torcendo pela realização deste sonho.

A minha irmã, Regiane Tu Kun Ma, pelo incentivo na realização deste trabalho.

Ao professor Marco Aurélio Stefanes pelos ensinamentos, conselhos, ajuda, tempo dedicado e sempre direcionando meus estudos.

Ao professor Carlos Henrique Aguena Higa pela colaboração, apoio e oportunidade de dar continuidade no seu trabalho.

Aos professores e funcionários da FACOM que contribuíram de forma direta ou indiretamente para meu aperfeiçoamento profissional e pessoal.

Aos amigos que ganhei durante o mestrado: Angelo Maggioni e Silva, Hudson Fujikawa de Paula, Valter de Oliveira Ferlete, Camila Koike e Luiz Fernando Alvino.

Aos professores, colegas e funcionários que fazem parte dos laboratórios CTEI e PET-Sistemas.

Abstract

Inference is the process of clarifying the relationships formed between genic products/proteins through a mathematical model. Conceptually this is an ill-posed problem since from a sample of gene expression data it is possible to infer several consistent networks with this sample. Depending the algorithm the inference process can take hours and even days due to the size of the networks and the complexity of the algorithm. The sequential inference algorithm used in this work is based on the seed growing paradigm and has two steps: the seed growing step and the inference step. In this work, we have developed three parallel versions of this inference algorithm with the following approaches: CPU cluster, GPU/CUDA and hybrid. These versions present acceptable costs and were compared with the sequential algorithm.

Keywords: boolean networks, gene regulatory networks, inference, parallelism, hybrid algorithms, GPU, CUDA.

Resumo

A inferência é o processo de esclarecer as relações formadas entre produtos gênicos/proteínas por meio de um modelo matemático. Conceitualmente este é um problema mal-posto, uma vez que a partir de uma amostra de dados de expressão gênica é possível inferir diversas redes consistentes com essa amostra. Dependendo do algoritmo, o processo de inferência pode levar horas e até dias devido ao tamanho da rede e complexidade do algoritmo. O algoritmo sequencial de inferência utilizado neste trabalho baseia-se no paradigma de crescimento da semente e possui dois passos: passo de crescimento da semente e passo de inferência. Neste trabalho, desenvolvemos três versões paralelas desse algoritmo de inferência com as seguintes abordagens: *cluster* de CPUs, GPU/CUDA e híbrida. Essas versões apresentam custos aceitáveis sendo comparadas com o algoritmo sequencial.

Palavras-chave: redes booleanas, redes de regulação gênica, inferência, paralelismo, algortimos híbridos, GPU, CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API Interface de Programação de Aplicações (*Application Programming Interface*)

BFS Busca em Largura (*Breadth-First Search*)

BN Redes Booleanas (*Boolean Networks*)

CPU Unidade Central de Processamento (*Central Processing Unit*)

CSP Problema de Satisfação de Restrições (*Constraint Satisfaction Problem*)

CUDA *Compute Unified Device Architecture*

DNA Ácido Desoxirribonucleico (*Deoxyribonucleic Acid*)

GPGPU Computação de Propósito Geral em Unidade de Processamento Gráfico (*General-Purpose computing on Graphics Processing Units*)

GPU Unidade de Processamento Gráfico (*Graphics Processing Unit*)

GRN Rede de Regulação Gênica (*Gene Regulatory Networks*)

IFFS *Improved Forward Floating Selection*

MPI *Message Passing Interface*

OpenMP *Open Multi-Processing*

RNA Ácido Ribonucleico (*Ribonucleic Acid*)

SBS *Sequential Backward Selection*

SFFS *Sequencial Forward Floating Selection*

SFS *Sequential Forward Selection*

SIMD *Single Instruction Multiple Data*

TBN Redes Booleanas Limiarizadas (*Thresholded Boolean Networks*)

TBNp Redes Booleanas Limiarizadas com perturbação (*Thresholded Boolean Networks with perturbation*)

Lista de Algoritmos

CAPÍTULO

1

Introdução

Bioinformática é o termo introduzido por Hogeweg e Hesper (1978), que descreve os processos e técnicas computacionais utilizados para realizar estudos e análises de sistemas biológicos. Uma das áreas importantes da Bioinformática é a Biologia Sistêmica, cujo objetivo é entender o funcionamento dos organismos como um todo, e se caracteriza pelas relações construídas entre as partes constituintes (por exemplo, genes e proteínas) (Ideker *et al.*, 2001).

Modelos matemáticos foram propostos para representar a estrutura desses sistemas e ajudar a compreender as relações entre estes componentes celulares. Os modelos matemáticos de redes de regulação gênica (GRNs) possuem suas próprias características, podendo ser discretos ou contínuos, determinístico ou estocástico, entre outros. Vários modelos foram propostos ao longo dos anos, por exemplo, o modelo de Equações Diferenciais (Goodwin, 1963), o modelo de redes Bayesianas (Friedman *et al.*, 2000), redes Booleanas (Kauffman, 1969) e redes Booleanas probabilísticas (Shmulevich *et al.*, 2002a).

A inferência de redes de regulação gênica constitui um problema bastante conhecido em Biologia Sistêmica. O processo de inferência deve esclarecer as relações formadas a partir dos dados de expressão gênica através de um modelo matemático. Esses modelos ajudam a capturar o comportamento do sistema que está sendo modelado. O problema que aparece na modelagem da inferência é que várias redes podem ser consistentes e podem explicar os dados de expressão gênica. Higa *et al.* (2013) propõem um algoritmo de inferência capaz de inferir diversas redes que expliquem os dados de entrada. Esse algoritmo possui dois passos: crescimento da semente e inferência.

Convém lembrar que a inferência de redes é um processo que, dependendo do algoritmo, pode consumir horas e até dias. Para que possamos fazer a

inferência em um tempo razoável, plataformas de alto desempenho, como a utilização de *cluster* computadores e a Unidade de Processamento Gráfico (GPU) podem ser exploradas a fim de agilizar esse processo. Um *cluster* de computadores é visto como um sistema único para se referir a um grupo de computadores conectados em uma rede local dedicada, de alta velocidade, com o objetivo de reduzir o tempo de processamento de uma tarefa, dividindo-a em subtarefas e distribuindo-as entre os computadores (ou nós) conectados na rede local. A GPU, além de renderizar imagens, vem sendo utilizada para realizar cálculos de propósito geral. Essa prática, antes realizada pela CPU, é conhecida como Computação de Propósito Geral em Unidade de Processamento Gráfico (GPGPU).

1.1 Objetivos

A tarefa de inferir redes de regulação gênica é muito complexa. A medida em que aumenta o número de genes, o número de estados cresce exponencialmente e, em muitos casos, a inferência dessas redes torna-se computacionalmente inviável. O objetivo deste trabalho concentra-se em paralelizar o algoritmo de inferência (Higa *et al.*, 2013) com custos aceitáveis, utilizando as arquiteturas de *cluster* de CPUs, GPU e híbrida (múltiplas CPUs/GPUs); e comparar o desempenho de tempo das versões paralelas com o algoritmo sequencial.

1.2 Resultados

No passo de crescimento da semente, realizamos testes experimentais em nossa versão paralela utilizando *cluster* com 16 CPUs, e alcançamos *speedups* de até 7,3 para células *HeLa* e até 6,3 para a rede *In Silico*. Na implementação híbrida (CPUs/GPUs) com 8 CPUs/GPUs, conseguimos *speedups* de até 11 para células *HeLa* e *speedups* de até 8,1 para a rede *In Silico*. Também implementamos uma versão paralela utilizando apenas GPU, contudo, não conseguimos bons desempenhos em relação ao algoritmo sequencial.

No passo de inferência, implementamos uma versão paralela utilizando *cluster* de CPUs e os experimentos com 16 CPUs mostraram que a versão paralela é em torno de 10 vezes mais rápida em relação ao algoritmo sequencial.

1.3 Organização do Texto

O restante desta dissertação encontra-se organizado como segue. No capítulo 2 estão descritos os conceitos preliminares utilizados neste trabalho, como, os modelos de paralelismo, conceitos biológicos, os modelos de redes de regulação gênica, o conjunto de restrições utilizados no algoritmo de inferência e os algoritmos de seleção de características. No capítulo 3, está descrito o algoritmo sequencial de inferência. No capítulo 4, estão descritas as implementações realizadas neste trabalho. No capítulo 5, estão descritos os resultados dos experimentos realizados. No capítulo 6, estão descritas as conclusões deste trabalho.

CAPÍTULO

2

Preliminares

Este capítulo descreve alguns conceitos iniciais utilizados ao longo deste trabalho, como as arquiteturas paralelas, os conceitos biológicos, o modelo matemático utilizado para representar as redes de regulação gênica, o conjunto de restrições para gerar as linhas consistentes e o problema de seleção de características.

2.1 Arquiteturas Paralelas

Desde a década de 1980, com a crescente demanda por processamento computacional, a computação sequencial tem encontrado dificuldades em processar grandes volumes de dados. Na busca para expandir o poder de processamento, é possível utilizar o paralelismo computacional agregando novas unidades de processamento para realizar cálculos simultâneos e reduzir o tempo de computação dos dados.

Na computação paralela, os problemas que exigem grandes quantidades de processamento são divididos em subproblemas menores e resolvidos simultaneamente pelas unidades de processamento. O paralelismo pode ser explorado fazendo o uso de *cluster* de CPUs, que é um conjunto de computadores conectados em uma rede local de alta velocidade, com o objetivo de realizar tarefas simultâneas. Outra alternativa é explorar o uso da Unidade de Processamento Gráfico (GPU) para obter ganho de desempenho a baixo custo. Além disso, também é possível fazer o uso de arquiteturas híbridas, que mescla a utilização de CPUs e GPUs. Essas três arquiteturas são discutidas ao longo desta seção.

2.1.1 Cluster de CPUs

Um *cluster* de CPUs consiste de várias unidades de processamento que estão forte ou fracamente acopladas para trabalharem em conjunto como se fosse um único sistema. Em um sistema fortemente acoplado, os processadores estão próximos (no mesmo barramento) e compartilham os mesmos recursos de memória e dispositivos de entrada/saída e são gerenciados pelo mesmo sistema. Nos sistemas fracamente acoplados, os processadores estão distantes (conectados em uma rede de alta velocidade), funcionam de forma independente, possuem os próprios recursos e se comunicam mediante trocas de mensagens (Tanenbaum, 1984).

Segundo Pfister (1998), a ideia do *cluster* surgiu, por volta do ano de 1960, por meio dos usuários que não conseguiam armazenar seus dados em um único computador. O primeiro *cluster* produzido foi o B5000 (Organick, 1973), nos anos 1970, e era formado por 4 computadores, cada um com até 2 processadores sendo fortemente acoplados e compartilhavam o mesmo disco de armazenamento.

Entre 1993 e 1994, motivados pela crescente exigência na capacidade de processamento computacional e no custo elevado para adquirir supercomputadores, Sterling *et al.* (1995) iniciaram um projeto alternativo para a criação de um *cluster* denominado *Beowulf*. Um *cluster Beowulf* consiste em conseguir alto desempenho utilizando computadores convencionais conectados por uma rede dedicada, juntamente com um servidor mestre coordenando as tarefas para os demais computadores (escravos). As vantagens desse *cluster* consistem no baixo custo financeiro para implementação, escalabilidade do sistema e facilidade de manutenção dos nós. As desvantagens são que os programas devem ser escritos para sistemas paralelos usando troca de mensagens, a velocidade do *cluster* é limitada pela velocidade da rede. Além disso, se a rede ou algum dos computadores pararem, todas as aplicações executadas também ficam comprometidas.

Na arquitetura de *cluster* de CPUs, quando surge a necessidade de comunicação entre os vários processadores do *cluster*, é preciso utilizar um mecanismo para realizar essa comunicação. Um padrão muito utilizado é o MPI (*Message Passing Interface*) (Gropp *et al.*, 1996). O MPI corresponde a uma especificação de biblioteca que implementa rotinas de troca de mensagens entre os processadores (ponto-a-ponto), operações coletivas e globais.

Nas aplicações que utilizam a biblioteca MPI, o ambiente precisa ser inicializado chamando a função *MPI_Init* e finalizando com *MPI_Finalize*. Uma aplicação desenvolvida com a biblioteca MPI é constituída por um ou mais processos, os quais são identificados de forma única dentro de seu grupo e obtidos por meio da função *MPI_Comm_rank*. Os processos podem realizar co-

municação ponto-a-ponto (trocar mensagens com um processo específico) ou para um grupo de processos, podendo ser efetuada de maneira síncrona ou assíncrona, além de permitir enviar estruturas de dados personalizadas.

A biblioteca MPI é amplamente utilizada pela comunidade acadêmica, e está disponível para as linguagens C/C++, Fortran, Java, Python, entre outras, e se destaca pela portabilidade, praticidade, eficiência e flexibilidade.

Neste trabalho, utilizamos o termo *nó* para nos referirmos a uma unidade de processamento do *cluster* e, o termo *processos*, para as tarefas associadas a uma aplicação baseada no MPI, que são escalonadas entre os diversos nós do *cluster*. O termo *mestre* representa o nó responsável pela coordenação das atividades e *escravo* para os demais nós.

2.1.2 GPU

Entre os anos de 1960 e 1980, a computação paralela voltava-se apenas para o incremento de novas unidades de processamento e aumento da frequência de operação dos processadores. Após várias limitações na fabricação desses circuitos, como no tamanho físico dos transístores e restrições de energia e calor, os pesquisadores e fabricantes começaram a buscar novas alternativas para aumentar a capacidade de processamento (Sanders e Kandrot, 2010).

Ao longo dos anos de 1980, a companhia Silicon Graphics popularizou o uso de gráficos 3D para uma variedade de mercado, como em aplicações governamentais, científicas e indústria cinematográfica. Em 1992, a Silicon Graphics tornou pública a biblioteca OpenGL para programar aplicações em 3D. Em 1995, a Microsoft lançou o DirectX, uma API para manipular aplicações multimídia e que se tornou muito popular para o desenvolvimento de jogos do Windows (Bargen e Donnelly, 1998).

No início dos anos 1990, os aceleradores gráficos começaram a ganhar destaque no mercado para computadores pessoais com o surgimento de sistemas operacionais gráficos. A série *GeForce 256* foi lançada em 1999 pela Nvidia como a primeira linha de GPUs disponível no mercado para os usuários domésticos e tinha como principal diferencial as funções de iluminação e transformação de vértices acelerada por *hardware*. Tais funções, antes, eram executadas pela CPU. Mais tarde, em 2001, a Nvidia lançou a série *GeForce 3* com sombreamento programável e o padrão DirectX implementado (Sanders e Kandrot, 2010).

Como as APIs gráficas (OpenGL e DirectX) eram o único jeito de interagir com a GPU, os pesquisadores começaram a explorar o uso dela para propósito geral, formulando seus problemas como sendo um problema de renderização de *pixels*, atribuindo cores para valores numéricos e, depois, trazendo

a cor final do *pixel* como resultado. Apesar das limitações de recursos e de programação, esse experimento foi promissor em razão do alto *throughput*[1] aritmético das GPUs (Sanders e Kandrot, 2010).

Em 2007, a Nvidia introduziu uma plataforma e um modelo de programação paralela chamada CUDA, no qual os cálculos e instruções são executados diretamente nas GPUs da Nvidia. CUDA é uma extensão para a linguagem C, que permite o controle da execução de *threads* na GPU, gerenciando sua memória. Mais tarde, surgiu um modelo aberto de programação paralela heterogênea (CPUs, GPUs e outros dispositivos) chamado OpenCL[2]. Neste estudo utilizamos a plataforma CUDA em razão de ela apresentar melhor desempenho em relação ao OpenCL quando executado em uma GPU Nvidia.

O *hardware* da GPU Nvidia é composto por um conjunto de multiprocessadores (SMs), cada um contendo uma memória compartilhada (*shared memory*), de textura e constante, além de cada processador possuir um conjunto de registradores locais, como mostra a Figura 2.1.

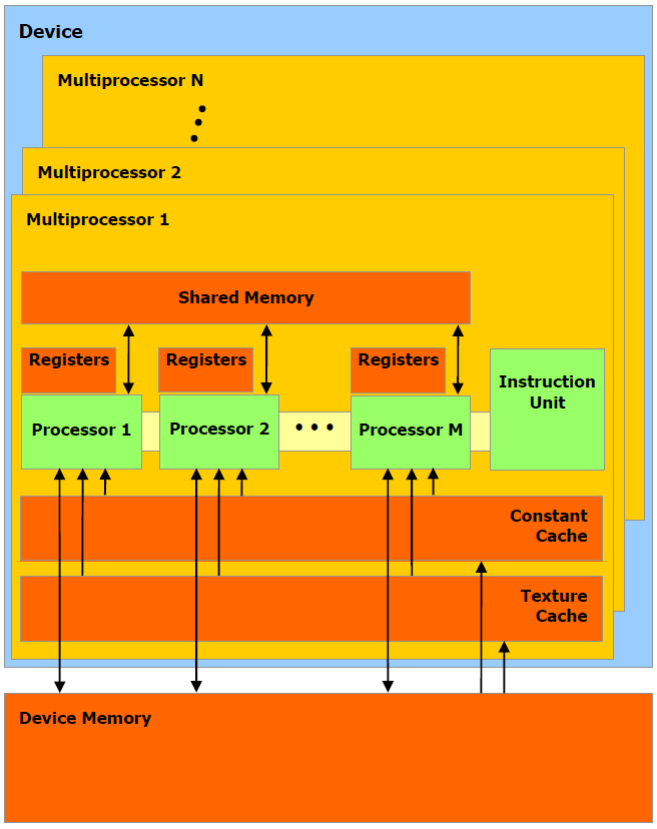


Figura 2.1: Modelo de *hardware* da GPU (CUDA NVIDIA, 2007, p.14).

Um programa escrito em CUDA possui duas partes: uma para ser executada na CPU, chamada de código *host*, e, outra, para ser executada na GPU, chamada de código *device*. Um *kernel* é uma função chamada pela CPU e

[1] quantidade de dados que um sistema pode processar em um determinado período de tempo

[2] https://www.khronos.org/opencl/

executada na GPU, especificando o número de *blocos* e de *threads* a serem utilizados durante sua execução. Por exemplo:

$$kernel\_teste <<< n,\ m >>> (arg1,\ arg2);$$
$$cudaDeviceSynchronize();$$

onde o *kernel*, chamado de $kernel\_teste$, é invocado pela CPU para ser executado na GPU com n blocos e m *threads*. Os argumentos $arg1$ e $arg2$ são variáveis declaradas na memória da GPU e são utilizados como parâmetros de entrada para o *kernel*. A chamada da função $cudaDeviceSynchronize()$ bloqueia a CPU até que todas as tarefas sejam completadas[3]. Outra função de sincronização bastante utilizada é a $\_\_syncthreads()$, e é utilizada apenas na GPU, funcionando como uma barreira para que todas as *threads* do mesmo bloco alcancem o mesmo ponto no algoritmo.

A execução das instruções baseia-se no modelo *Single Instruction Multiple Data* (SIMD), no qual uma única instrução é executada em múltiplos dados. As memórias em CUDA possuem diferentes tamanhos, velocidades e tipos de acesso, como mostrado na Figura 2.2. Os registradores são memórias de alta velocidade e acessados apenas via *threads*. A memória compartilhada é uma memória de rápido acesso para as *threads* que estão no mesmo bloco. A memória global, por sua vez, é uma memória lenta de leitura/escrita e pode ser acessada por todas as *threads* em execução. As memórias constante e de textura são memórias de alta velocidade, apenas para leitura, podendo ser lidas por todas as *threads*. A memória local não existe fisicamente, essa memória é utilizada pelo compilador para colocar, automaticamente, as variáveis quando não houver registradores disponíveis. A memória local reside na memória global da GPU e tem a mesma velocidade de acesso da memória global. Nesse sentido, otimizar o uso de memória em CUDA é uma técnica que vem sendo muito estudada, pois cada memória possui características distintas que podem influenciar no desempenho do programa.

2.1.3 Híbrido

Com o objetivo de aumentar o desempenho computacional, é possível combinar as arquiteturas de CPU e GPU e formar uma arquitetura híbrida. Em outras palavras, os multiprocessadores disponíveis nas GPUs são responsáveis por executar cálculos paralelos e os núcleos da CPU atuam como distribuidor e sincronizador de dados.

O primeiro supercomputador, baseado em arquitetura híbrida, a liderar o *ranking* do site Top500[4] de supercomputadores, foi o *Tianhe-1A*, na lista de

[3] A partir dos dispositivos com *compute capability* 3.5 essa função pode ser chamada tanto na CPU quanto na GPU.

[4] http://www.top500.org

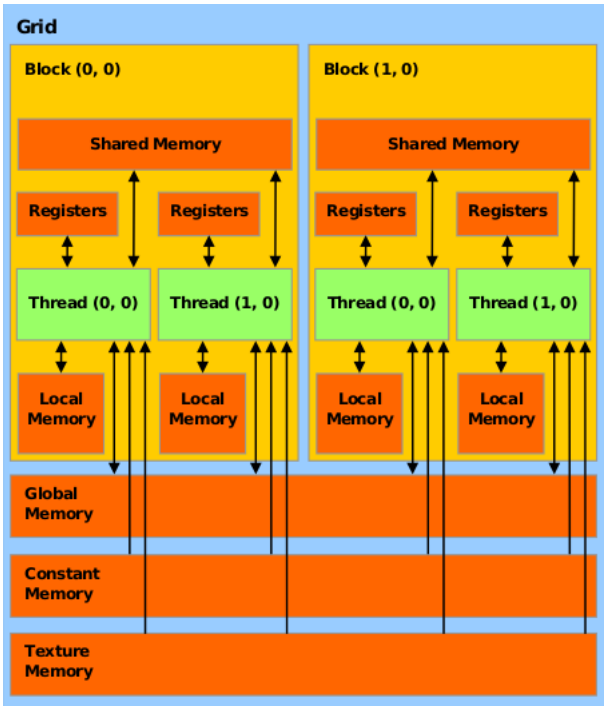


Figura 2.2: Modelo de memória da GPU (CUDA NVIDIA, 2007, p.11).

novembro de 2010. O *Tianhe-1A* era composto por 14.336 processadores Intel Xeon, 7.168 GPUs Nvidia Tesla e seu desempenho foi de 2.57 petaflop/s (quadrilhões de cálculos por segundo). Atualmente, dois supercomputadores, que estão entre os dez primeiros colocados do Top500, se baseiam em arquitetura híbrida, especificamente o *Titan*[5] e *Piz Daint*[6], sendo que esse último ganha destaque por ser o segundo mais eficiente em termos energéticos (razão entre o desempenho alcançado e energia gasta) da lista de novembro de 2016.

Em um ambiente com uma única GPU, para a execução de programas em CUDA, o usuário precisa explicitar a quantidade de memória requerida no dispositivo, transferência de dados para GPU, chamar o *kernel* na CPU e, então, trazer de volta esses resultados para a CPU. Quando as GPUs estão fisicamente distribuídas pela rede, é necessário algum mecanismo de comunicação para coordenar as tarefas. Tem sido muito utilizada a mesclagem da programação MPI/CUDA. A CPU assume o papel de controlador e comunicador entre os processos da aplicação (através do MPI) e a GPU é responsável por executar a computação dos dados (por meio do CUDA).

2.2 Conceitos Biológicos

A célula biológica é a unidade funcional e estrutural básica dos organismos vivos (procariotos e eucariotos) independente de o organismo ser simples

[5]https://www.olcf.ornl.gov/titan/
[6]http://www.cscs.ch/computers/piz_daint_piz_dora/index.html

ou complexo. As células possuem diversos mecanismos internos, como, divisão e diferenciação celular e novas células são formadas a partir de outras células existentes herdando todas as informações genéticas. O genoma conforma toda informação hereditária de um organismo que está codificada em seu DNA. Ressalta-se que a principal função do DNA é armazenar as informações genéticas necessárias para a síntese de proteínas realizada pelo RNA. Um gene é uma sequência específica de nucleotídeos capaz de desempenhar uma função no organismo (Nussbaum *et al.*, 2008).

2.2.1 DNA, RNA e Proteínas

O DNA é composto por nucleotídeos, os quais são formadas por uma base nitrogenada (adenina, citosina, guanina e timina) e moléculas de carbono e fosfato. A estrutura do DNA é formada por duas fitas de carbono e fosfato, girando em formato helicoidal, ligadas entre si por par de bases. O pareamento de bases é específico: a base adenina pareia somente com a base timina e a base guanina somente com a base citosina. Um grupo de fosfato, ribose e uma base nitrogenada (adenina, citosina, guanina e uracila) forma o RNA. A estrutura do RNA é formada por uma única fita, a qual é formada a partir de uma fita do DNA, tendo como uma das bases a uracila em vez da timina. A estrutura do DNA e RNA pode ser vista na Figura 2.3. O produto gênico final resultante da síntese do RNA, geralmente é uma proteína. As proteínas são constituídas por cadeias de aminoácidos que desempenham diversas funções nos organismos como construção de novos tecidos, defesa imunológica e transporte de moléculas (Nussbaum *et al.*, 2008).

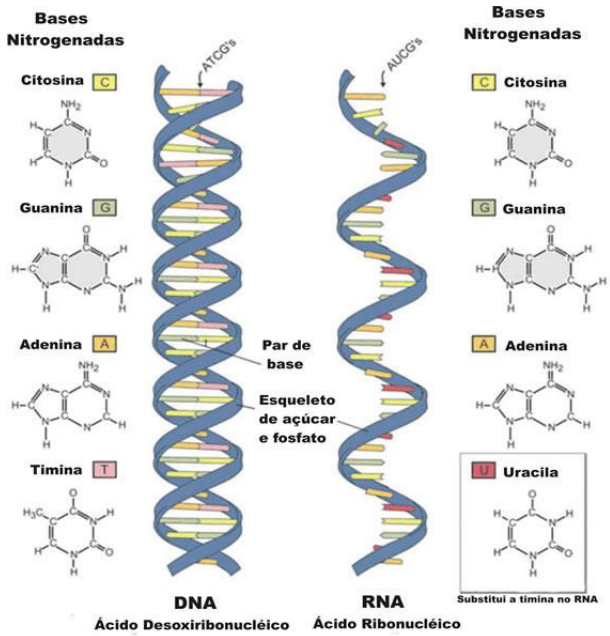


Figura 2.3: Estrutura molecular do DNA e RNA (Só Biologia, 2008).

O DNA, RNA e proteínas têm relações entrelaçadas. O DNA carrega as informações necessárias para a síntese de proteínas. Essas informações contidas nos genes de uma sequência de DNA só podem ser transcritas por uma molécula de RNA para a síntese de proteínas. Esse fluxo de informações é conhecido como Dogma Central da Biologia Molecular (Crick, 1970), Figura 2.4. Esse dogma também conceitua a replicação de DNA, a transcrição reversa (RNA para DNA) e a replicação de RNA.

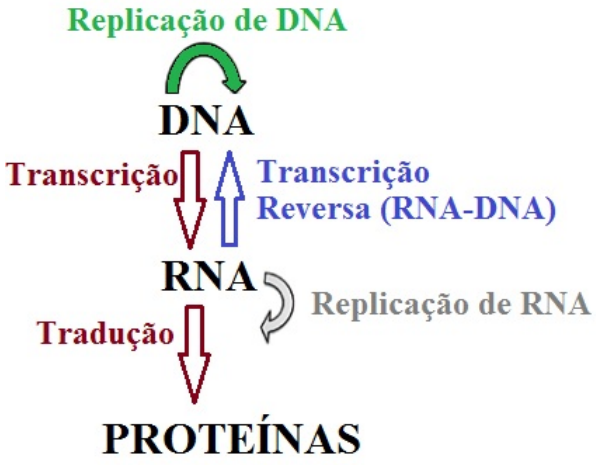


Figura 2.4: Dogma Central da Biologia Molecular (Educação, 2017).

2.2.2 Expressão Gênica

Define-se expressão gênica como o processo pelo qual a informação codificada por um gene é decodificada para a síntese do produto gênico. Esse processo é realizado em quatro passos: transcrição, *splicing* do RNA (eucariotos), tradução e modificação pós-traducional da proteína (Só Biologia, 2008).

Nos procariotos, o início do processo de transcrição do RNA gera o mRNA. Enquanto, nos eucariotos, a transcrição do RNA gera o pré-mRNA, necessitando de um conjunto de fatores de transcrição para se tornar o mRNA. O pré-mRNA é formado por *íntrons* (segmentos que não codificam proteínas) e *éxons* (segmentos que codificam proteínas). O processo de *splicing* remove todos os *íntrons* do pré-mRNA e une todos os *éxons* para produzir o mRNA. Após o *splicing*, o mRNA é decodificado pelos ribossomos para a produção de proteínas; essa etapa é conhecida como tradução. As modificações pós-traducional correspondem às mudanças químicas que ocorrem na biossíntese das proteínas; após a fase de tradução, por exemplo, a fosforilação adiciona fosfato em uma proteína (Alberts *et al.*, 2008).

DNA Microarrays

O nível de transcrição de uma grande quantidade de genes pode ser medido na tecnologia de DNA *microarray* (Schena *et al.*, 1995). Existem outras tecnologias de medição, como SAGE (*serial analysis of gene expression*) (Velculescu *et al.*, 1995) e RNA-Seq (Wang *et al.*, 2009). Para realizar a técnica de *microarray*, duas fitas de DNA são hibridizadas pela propriedade de complementaridade entre nucleotídeos. A reação em cadeia da polimerase (PCR) amplifica um gene conhecido, em seguida, coloca-se certa quantidade do DNA desse gene em uma lâmina de vidro em um certo ponto (chamado de *spot*). Vários *spots* são preenchidos de modo que vários genes possam ser analisados simultaneamente (Shalon *et al.*, 1996).

Para analisar a expressão gênica, uma enzima conhecida como transcriptase reversa transforma os mRNAs em cDNA (DNA complementar). Utiliza-se o fluoróforo (corante fluorescente) para marcar os cDNAs e, se determinado mRNA estiver presente na célula, o cDNA marcado correspondente irá ligar-se por hibridização à sua cadeia complementar na lâmina. Os cDNAs não hibridizados são removidos da lâmina por meio de um processo de lavagem. Assim, a lâmina passa por um processo de análise para medir o nível de expressão do gene. A medição do nível de expressão do gene é indicada pela intensidade do brilho dos *spots* quando o fluoróforo é estimulado por luz. Tal processo é chamado de *single-channel*, por usar somente um corante (Shalon *et al.*, 1996).

Em processos *double-channel*, utilizam-se dois corantes no cDNA: corante vermelho (Cy5) para marcar a amostra fonte, e corante verde (Cy3) para marcar a amostra de referência. Os cDNAs são misturados e hibridizados; *spots* com brilho na cor vermelha indicam que o gene está expresso na amostra fonte e o brilho verde indica a expressão na amostra de referência. Dessa forma, quando o gene está expresso em ambas as amostras, o *spot* apresenta coloração amarela (Shalon *et al.*, 1996).

O objetivo final dessa técnica é extrair o sinal de cada *spot* e gerar uma matriz numérica contendo as intensidades dos sinais, a partir da análise dos *spots*. Para esse objetivo ser alcançado, convém analisar os *spots* com processamento de imagem do *microarray* (no qual é possível extrair o sinal do *spot*), analisar sua variabilidade e avaliar sua qualidade. Em geral, as linhas da matriz correspondem aos genes e, as colunas, aos experimentos de microarray (Shalon *et al.*, 1996).

Os dados de expressão gênica providos por *microarrays* podem ser estacionários ou temporais (Shalon *et al.*, 1996). No primeiro, uma amostra não possui relação de tempo com as outras; no segundo, os dados temporais representam uma análise das expressões ao longo do tempo (Shalon *et al.*, 1996).

Neste trabalho, utilizamos dados temporais uma vez que oferecem uma visão mais completa do sistema do que dados estacionários (Sima *et al.*, 2009).

Discretização dos Dados

Os experimentos de *microarray* fornecem dados contínuos - matriz numérica com valores no domínio dos números reais. Neste trabalho, utilizamos dados discretos e, para isso, discretizamos os dados com o algoritmo BiKmeans (Li *et al.*, 2010), que é uma combinação do algoritmo Kmeans (MacQueen, 1967) com o CoKmeans (Kmeans aplicado em colunas).

Considere $E_{n\times m}$ uma matriz de expressões gênicas de n genes com m amostras. Dessa forma, o vetor $E(i,:)$ corresponde às expressões do gene i em todas as amostras. De forma similar, o vetor $E(:,j)$ denota a expressão de todos os genes na amostra j. O algoritmo Kmeans divide $E(i,:)$ em k intervalos e agrupa os valores próximos de expressão do gene i no mesmo intervalo. Enquanto o algoritmo CoKmeans divide $E(:,j)$ em k intervalos e agrupa os valores próximos da amostra j no mesmo intervalo.

O algoritmo BiKmeans implementa o Kmeans e CoKmeans com parâmetro $k+1$ e resulta em dois valores discretos para cada expressão gênica. Se o produto dos dois valores for igual ou maior que e^2 e menor que $(e+1)^2$, o valor discreto final dessa expressão é e, onde e é um inteiro positivo no intervalo 1 a k. Por exemplo, se uma expressão recebe valor 2 pelo Kmeans e 1 pelo CoKmeans com parâmetro $k+1=3$, o produto é $2\times 1=2$, que é maior que $1^2=1$ e menor que $(1+1)^2=4$, ou seja, $e=1$ e a expressão é atribuída ao primeiro intervalo.

		Kmeans	
	1	2	3
1	1	2	3
CoKmeans 2	2	4	6
3	3	6	9

Tabela 2.1: Os possíveis produtos de Kmeans e CoKmeans para $k+1=3$. O primeiro intervalo corresponde aos produtos 1 a 3 e o segundo intervalo aos produtos 4 a 9.

Neste trabalho, os dados são discretizados em dois níveis (0 e 1), assim, o algoritmo BiKmeans é utilizado com o parâmetro $k+1=3$. O primeiro intervalo é separado entre os valores 1 a 3 e o segundo intervalo entre os valores 4 a 9 da Tabela 2.1. Quando a expressão gênica está no primeiro intervalo, atribuímos o valor 0 para denotar que o gene está "inibido" ou "desligado". Se a expressão gênica estiver no segundo intervalo, atribuímos o valor 1 para denotar que o gene está "ativo" ou "ligado".

2.3 Modelos de Redes de Regulação Gênica

Diversos modelos matemáticos foram propostos para representar as estruturas e relações entre os genes. Dentre os modelos existentes, destacam-se Equações Diferenciais (Goodwin, 1963), o de redes Bayesianas (Friedman *et al.*, 2000), Booleanas (Kauffman, 1969) e Booleanas Probabilísticas (Shmulevich *et al.*, 2002a), cada um apresentando suas próprias características. Nesta seção, são descritas as definições sobre o modelo de Redes Booleanas e seu modelo derivado, o *Thresholded Boolean Network with Perturbation* (TBNp), o qual é utilizado neste trabalho.

2.3.1 Redes Booleanas

A Rede Booleana, ou simplesmente BN, é um modelo matemático discreto utilizado para modelar as interações de genes/proteínas mediante variáveis binárias, $x_i \in \{0, 1\}$, que definem o estado do gene x_i em um tempo t. Nesse modelo, os genes podem assumir apenas dois valores, 0 ou 1, significando que o gene está inativo ou ativo, respectivamente. (Kauffman, 1969).

Uma rede Booleana consiste em um conjunto de genes $X = \{x_1, x_2, \ldots, x_n\}$ para $i = 1, 2, \ldots, n$, e um conjunto de funções Booleanas $F = \{f_1, f_2, \ldots, f_n\}$, onde f_i é uma função Booleana do gene x_i. O valor do gene x_i no tempo $t + 1$, é determinado por k_i genes no tempo t por meio de uma função Booleana f_i (Shmulevich *et al.*, 2002a):

$$x_i(t+1) = f_i(x_{j_1}(t),\ x_{j_2}(t), \ldots,\ x_{j_{k_i}}(t)), \tag{2.1}$$

onde k_i é a conectividade do gene x_i, ou seja, o número de genes que regula o gene x_i. Nesse modelo, assumimos que todos os genes são atualizados de maneira síncrona pelas funções Booleanas.

Um estado da rede Booleana no tempo t constitui um vetor binário $\mathbf{x}(t) = (x_1(t), x_2(t), \ldots, x_n(t))$. Assim, nas redes Booleanas, temos 2^n estados possíveis, enumerados de $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{2^n-1}$. A aplicação de funções Booleanas no tempo t leva a rede para o estado no tempo $t + 1$ de maneira determinística. Essas transições de estado são representadas graficamente por um diagrama de transição de estados (Figura 2.5). Esse diagrama é um grafo no qual os estados da rede são representados pelos vértices e as funções Booleanas, que transitam do estado $\mathbf{x}_i \rightarrow \mathbf{x}_j$ são as arestas direcionadas de $\mathbf{x}_i$ para $\mathbf{x}_j$.

Os estados visitados ciclicamente são chamados de *atratores* e os estados que o levam para um atrator são chamados de estados *transientes*. Os atratores e os estados transientes que conduzem aos atratores formam a *bacia de atração* da rede; enquanto os atratores formados por um único estado são

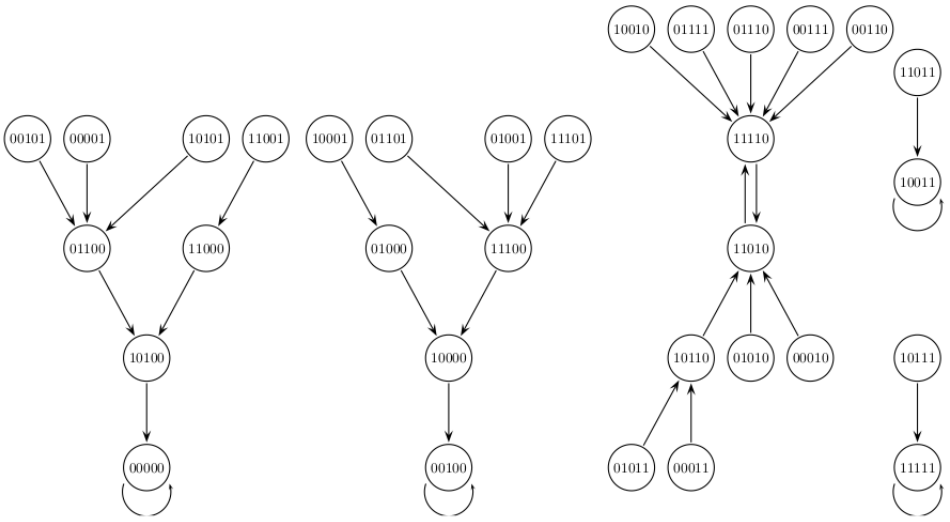


Figura 2.5: Exemplo de diagrama de transição de estados de uma BN com 5 genes, retirado de (Higa, 2011, p.20).

chamados de *singleton*; por exemplo, na Figura 2.5, os estados (00000) e (00100) são exemplos de dois atrator *singleton*. Os estados (11110) e (11010) formam um atrator, porém esse atrator é composto por dois estados e não pode ser chamado de atrator *singleton*.

2.3.2 Redes Booleanas Limiarizadas com perturbação

Uma Rede Booleana Limiarizada (do inglês *Thresholded Boolean Network*) consiste em um conjunto de genes $X = \{x_1, x_2, \ldots, x_n\}$, onde $x_i \in \{0, 1\}, i = 1, 2, \ldots, n$, e uma função Booleana f_i definida de acordo com uma matriz de regulação $A_{n\times n}$, onde $a_{i,j} \in \{-1, 0, 1\}$ (Li *et al.*, 2004). A matriz A indica o tipo de regulação que um gene x_j exerce em relação ao gene x_i, podendo ser:

$$a_{i,j} = \begin{cases} -1, & \text{para uma regulação negativa de } x_j \text{ para } x_i \\ 1, & \text{para uma regulação positiva de } x_j \text{ para } x_i \end{cases} \tag{2.2}$$

A regulação negativa inibe ou desativa um gene; enquanto a regulação positiva ativa ou liga um gene. Quando $a_{i,j} = 0$, o gene x_j não tem nenhuma relação com x_i. Um aspecto importante desse modelo é a auto-degradação de um gene, que ocorre quando $a_{i,i} = -1$. Chamamos de entrada $\mathcal{I}_i(t)$ do gene x_i no tempo t o seguinte somatório:

$$\mathcal{I}_i(t) = \sum_{j=1}^{n} a_{ij} x_j(t). \tag{2.3}$$

Assim, a função Booleana que define o valor do gene x_i no tempo $t + 1$ é dada

pela Equação (2.4):

$$x_i(t+1) = \begin{cases} 1, & \text{se} \quad \mathcal{I}_i(t) > \tau \\ 0, & \text{se} \quad \mathcal{I}_i(t) < \tau \\ x_i(t), & \text{se} \quad \mathcal{I}_i(t) = \tau, \end{cases} \tag{2.4}$$

onde τ é um limiar pré-definido. Neste trabalho, configuramos $\tau = 0$. O limiar indica os efeitos dos genes na dinâmica da rede. O estado do gene x_i é dito ativo quando sua entrada é positiva; inativo quando sua entrada é negativa e permanece o mesmo estado quando $\mathcal{I}_i(t) = 0$ (Li *et al.*, 2004). É possível notar que a entrada do gene x_i depende apenas da *i*-ésima linha da matriz A.

No modelo TBNp considera perturbações (estímulo externo: mutação ou variações de temperatura) nos genes. A perturbação nos genes possibilita que a rede mude de um estado para outro de forma aleatória (Shmulevich *et al.*, 2002b).

A perturbação corresponde à troca de valor do gene e é determinada por um vetor de perturbação aleatório $\gamma \in \{0,1\}^n$. O valor 1 na i-ésima posição do vetor γ define que o valor do gene x_i deve ser trocado; caso contrário, não (Shmulevich *et al.*, 2002b). Os genes podem ser perturbados com probabilidade p, independente dos outros genes. Assim, a probabilidade do *i*-ésimo gene ser perturbado é $\Pr[\gamma_i = 1] = p$ para $i = 1, \ldots, n$. Portanto:

$$\Pr[\gamma = (0, \ldots, 0)] = (1-p)^n \tag{2.5}$$

Então, seja $\mathbf{x}(t)$ o estado da rede no tempo t, o próximo estado $\mathbf{x}(t+1)$ é dado por:

$$\mathbf{x}(t+1) = \begin{cases} \mathbf{x}(t) \oplus \gamma, & \text{com probabilidade } 1-(1-p)^n \\ \mathbf{f}(x_1(t), \ldots, x_n(t)), & \text{com probabilidade } (1-p)^n, \end{cases} \tag{2.6}$$

onde $\oplus$ corresponde à operação OU exclusivo (XOR) e $\mathbf{f}(x_1(t), \ldots, x_n(t))$ são as funções Booleanas aplicadas para cada gene (Shmulevich *et al.*, 2002b).

Uma linha consistente é uma atribuição de valores que satisfazem um conjunto de restrições (descrito na Seção 2.4). Uma rede consistente é formada por uma linha consistente para cada gene. A rede é dita *consistente* quando ela consegue reproduzir uma sequência de estados de uma série dada, considerando a perturbação nos genes. Diversas redes, porém, podem ser produzidas por uma mesma série temporal, como também pode não haver nenhuma rede (Andrade, 2012; Higa *et al.*, 2013).

2.4 O Conjunto de Restrições

Uma maneira ingênua de encontrar todas as redes consistentes consiste em realizar uma busca exaustiva, testando todas as possibilidades no seu espaço de busca, como no trabalho de Lau *et al.* (Lau *et al.*, 2007). Outra alternativa é explorar as propriedades do modelo TBNp como testar a consistência de uma linha separadamente, uma vez que as linhas das matrizes são independentes. Tal propriedade ajuda a reduzir o espaço de busca de 3^{n^2}, que é o número matrizes de regulação possíveis, para n problemas (um problema para a *i*-ésima linha da matriz A) de n variáveis com 3 relações possíveis da coluna j para linha i, reduzindo o espaço de busca para $n \times 3^n$ (Higa *et al.*, 2011; Higa, 2011).

Esta seção descreve o conjunto de restrições utilizado para encontrar as linhas consistentes da matriz de regulação. Uma linha consistente é uma atribuição de valores que satisfaz todos os conjuntos de restrições. Essas restrições baseiam-se em observações de m instantes de tempo (ou estados) de uma série temporal $\mathbf{T} = \{s(1), s(2), \ldots, s(m)\}$, onde $s(t) \in \{0, 1\}$ e $t = 1, \ldots, m$. O problema de encontrar linhas consistentes para um gene x_i pode ser modelado em forma de um Problema de Satisfação de Restrições (CSP), no qual uma solução consistente do CSP para o gene x_i representa uma linha consistente da matriz de regulação A para o gene x_i (Higa *et al.*, 2011).

De modo formal, um CSP é definido por 3 conjuntos (X, D, C), onde $X = \{x_1, x_2, \ldots, x_n\}$ é o conjunto de variáveis; $D = \{D_1, D_2, \ldots, D_n\}$ é o conjunto de domínios, onde D_i denota o conjunto com os possíveis valores que x_i pode assumir; e $C = \{C_1, C_2, \ldots, C_v\}$ é o conjunto de restrições empregado nas variáveis do conjunto X. Uma atribuição (ou solução) consistente do CSP é uma atribuição de valores presentes no domínio D_i para $\forall x_i \in X$ para $i = 1, \ldots, n$ que satisfazem todas as v restrições que estão dentro do conjunto de restrições C (Rossi *et al.*, 2006).

Uma observação importante é que por meio da Equação (2.3), sabe-se que cada linha da matriz de regulação A é independente das outras. Aplicando essa propriedade, encontramos apenas as linhas consistentes para o gene x_i ao invés de encontrar todas as soluções para a matriz. Dessa forma, o problema de interação entre n genes transforma-se em um conjunto de n CSPs. Nesse contexto, o problema está modelado da seguinte maneira: para cada gene x_i para $i = 1, \ldots, n$ temos um problema P_i definido pelo conjunto de variáveis $R_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,n})$ (correspondendo às n entradas da *i*-ésima linha da matriz A), conjunto de domínio $D_i = \{D_{i,1}, D_{i,2}, \ldots, D_{i,n}\}$ que representa os tipos de regulações possíveis, onde $D_{i,j} = \{-1, 0, 1\}$ para $j = 1, \ldots, n$, e um conjunto de três restrições descritos a seguir (Higa *et al.*, 2011).

2.4.1 Primeira Restrição

A primeira restrição analisa três estados consecutivos, $s(t-1)$, $s(t)$ e $s(t+1)$, de uma série temporal T. Observando os estados $s(t-1)$ e $s(t)$, se apenas um gene x_k é diferente, então qualquer gene x_i que tem seu valor alterado no estado $s(t)$ para $s(t+1)$ é regulado diretamente por x_k (Higa *et al.*, 2011). Por exemplo, considere a série temporal com quatro genes e cinco estados da Tabela 2.2:

	$x_1(t)$	$x_2(t)$	$x_3(t)$	$x_4(t)$
$s(1)$	1	1	0	0
$s(2)$	1	0	0	0
$s(3)$	1	0	1	0
$s(4)$	1	0	1	1
$s(5)$	0	0	1	1

Tabela 2.2: Exemplo de série temporal (Higa *et al.*, 2011).

Observando os estados $s(1)$ e $s(2)$, apenas o gene x_2 teve seu valor alterado de 1 para 0. Olhando, então, para $s(2)$ e $s(3)$, observamos que o gene x_3 alterou seu valor para 1. Pelo modelo TBNp, essa mudança no gene x_3 foi causada, necessariamente, pelo gene x_2, pois x_2 inibe x_3 quando $t = 1$ e quando x_2 tem seu valor alterado para 0 em $t = 2$, permite x_1 ativar x_3 em $t = 3$. Podemos fazer a seguinte proposição:

Proposição 2.4.1. *Sejam três estados consecutivos $s(t-1)$, $s(t)$ e $s(t+1)$ de uma TBNp. Se entre os estados $s(t-1)$ e $s(t)$ diferirem em apenas um único gene x_k, então, para cada gene x_i tal que $x_i(t) \neq x_i(t+1)$, temos que x_k regula diretamente x_i, ou seja, na matriz de regulação $a_{i,k} \neq 0$.*

Demonstração. Suponha que os estados $s(t-1)$ e $s(t)$ difiram apenas no gene x_k e que exista, pelo menos, um gene x_i tal que $x_i(t) \neq x_i(t+1)$. Então, os somatórios $\sum_j a_{ij}x_j(t-1)$ e $\sum_j a_{ij}x_j(t)$ devem possuir valores diferentes, uma vez que x_k possui valores diferentes em $s(t-1)$ e $s(t)$ e essa diferença é causada por x_k. Logo, $a_{ik} \neq 0$. □

2.4.2 Segunda Restrição

A segunda restrição considera dois tempos consecutivos $s(t)$ e $s(t+1)$. Pelas Equações (2.3) e (2.4), percebe-se que somente os genes expressos no tempo t podem alterar a expressão dos genes no tempo $t+1$. Apesar disso, não é possível determinar qual o tipo de regulação (ativação ou inibição). No entanto, a Equação (2.3) nos ajuda a encontrar o tipo de regulação (Higa *et al.*, 2011). Por exemplo, se observarmos que um gene x_i tem o seu valor alterado de 0

(tempo t) para 1 (tempo $t+1$), logo deduzimos que sua *entrada* no tempo t é positiva e que apenas os genes expressos no tempo t são responsáveis por essa *entrada* positiva.

Proposição 2.4.2. *De acordo com a Tabela 2.3 é possível gerar restrições de cada gene x_i, coerentes com as transições de $x_i(t)$ para $x_i(t+1)$ para as variáveis a_{ij} da matriz de regulação.*

Demonstração. Vamos demonstrar a restrição $x_i(t) = 0$ e $x_i(t+1) = 1$, então $\sum_{j:x_j(t)=1} a_{ij} > 0$. A única maneira de alterar o valor de um gene de 0 (no tempo t) para 1 (no tempo $t+1$) é quando $\sum_j a_{i,j}x_j(t) > 0$. Considerando que apenas os genes ativos no tempo t ($x_j(t) = 1$) influenciam no somatório, podemos reescrever essa restrição como $\sum_{j:x_j(t)=1} a_{ij} > 0$. As demais podem ser realizadas de maneira análoga. □

$x_i(t)$	$x_i(t+1)$	Restrição
0	0	$\sum_{j:x_j(t)=1} a_{ij} \leq 0$
0	1	$\sum_{j:x_j(t)=1} a_{ij} > 0$
1	0	$\sum_{j:x_j(t)=1} a_{ij} < 0$
1	1	$\sum_{j:x_j(t)=1} a_{ij} \geq 0$

Tabela 2.3: Restrições de transições de estados (Higa *et al.*, 2011).

Considere o estado $s(3)$ da Tabela 2.2, nele existem dois genes expressos x_1 e x_3. Esses genes são os únicos genes expressos e que podem contribuir para a entrada de cada gene no próximo instante. Observando o estado $s(4)$, apenas o gene x_4 teve seu valor alterado de 0 $(t = 3)$ para 1 $(t = 4)$. Então, de acordo com as Equações (2.3) e (2.4), a entrada deve ser positiva, isto é, $\sum_{j=1}^{4} a_{4,j}x_j(3) > 0$. Sabe-se que os únicos genes expressos no tempo $t = 3$ são os genes x_1 e x_3, temos que $a_{4,1} + a_{4,3} > 0$. Então, os valores de $a_{i,j}$ podem ser:

$$a_{4,1} = 0 \quad \text{e} \quad a_{4,1} = 1, \quad \text{ou}$$
$$a_{4,1} = 1 \quad \text{e} \quad a_{4,1} = 0, \quad \text{ou}$$
$$a_{4,1} = 1 \quad \text{e} \quad a_{4,1} = 1$$

2.4.3 Terceira Restrição

A terceira restrição observa dois pares de tempos consecutivos quaisquer da série temporal. Essa restrição verifica como os genes alteraram seus valores nos pares de tempos consecutivos mediante seus predecessores (Higa

et al., 2011). Sejam t_1 e t_2 dois instantes distintos, tal que:

$$\cdots \to s(t_1) \to s(t_1+1) \to \cdots \to s(t_2) \to s(t_2+1) \to \cdots$$

Suponha que os estados t_1 e t_2 sejam semelhantes. Os genes diferentemente expressos nos estados predecessores causaram a diferença dos genes nos estados $s(t_1+1)$ e $s(t_2+1)$. Por exemplo, suponha que $s(t_1)$ e $s(t_2)$ possuem apenas um gene diferente, nesse caso, o gene x_4:

$$s(t_1) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad s(t_2) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}.$$

E que a sucessão dos estados sejam:

$$\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \to \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \to \cdots \to \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \to \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Então, neste caso, a diferença entre $s(t_1+1)$ e $s(t_2+1)$ deve ser causada pelo diferença de expressão do gene x_4. Vamos considerar o gene x_1. No primeiro par de estados, ele é inibido (1 para 0) e, no segundo, não teve mudança de valor, permaneceu valendo 1. Seja I a entrada total do gene x_1 gerada pelos valores dos genes que estão em $s(t_1)$ e $s(t_2)$, M a entrada gerada por x_4 no tempo t_1 e $\bar{M}$ a entrada gerada por x_4 no tempo t_2. Portanto, a mudança do valor de x_i no primeiro par é explicada por:

$$I + M < 0 \tag{2.7}$$

para que o gene x_i seja inibido no tempo t_1 para t_1+1 e

$$I + \bar{M} \geq 0 \tag{2.8}$$

para que x_i continue valendo 1 no tempo t_2 para t_2+1. A relação de regulação do gene x_j sobre x_i é dado por a_{ij}, podemos calcular I, M e $\bar{M}$ da seguinte maneira:

$$I = (a_{1,1}a_{1,2}a_{1,3}) \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = a_{1,1} + a_{1,3} \tag{2.9}$$

$$M = a_{1,4} \cdot 0 = 0 \quad \text{e} \tag{2.10}$$

$$\bar{M} = a_{1,4} \cdot 1 = a_{1,4} \tag{2.11}$$

Então temos:

$$\begin{cases} I + M < 0 \\ I + \bar{M} \geq 0 \end{cases} \rightarrow \begin{cases} a_{1,1} + a_{1,3} + 0 < 0 \\ a_{1,1} + a_{1,3} + a_{1,4} \geq 0 \end{cases} \rightarrow \begin{cases} a_{1,4} > 0 \end{cases} \tag{2.12}$$

Isso implica que o valor de $a_{1,4} = 1$ na matriz de regulação, para que a rede TBNp seja consistente com a série temporal. Caso $s_(t_1)$ e $s_(t_1)$ se diferenciem em mais de um gene, é possível gerar algumas hipóteses sobre a regulação. Essa restrição gera um sistema de desigualdades com as entradas dos genes para cada combinação de pares de estados consecutivos.

Proposição 2.4.3. *Sejam $s_(t_1)$ e $s_(t_1)$ dois instantes de tempo distintos. As transições de $x_i(t_1)$ para $x_i(t_1+1)$ e de $x_i(t_2)$ para $x_i(t_2+1)$ geram restrições para variável a_{ij}, de acordo com a Tabela.*

Demonstração. Vamos fazer a prova da primeira restrição. O restante das restrições podem ser provadas de forma similar. Considere $x_i(t_1) = 1$, $x_i(t_1+1) = 0$, $x_i(t_2) = 0$ e $x_i(t_2 + 1) = 1$. Então,

$$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$$

Pela Proposição 2.4.2, considerando a transição de $x_i(t_1) = 1$ para $x_i(t_1+1) = 0$, temos que $\sum_{j:x_j(t_1)=1} a_{i,j} < 0$. O conjunto de índices de todos os genes expressos no tempo t_1, $E(t_1) = \{j : x_j(t_1) = 1\}$, pode ser escrito como uma união de dois conjuntos distintos:

$$E(t_1) = \{j : x_j(t_1) = 1 \text{ e } x_j(t_2) = 0\} \cup \{j : x_j(t_1) = 1 \text{ e } x_j(t_2) = 1\}. \tag{2.13}$$

Assim,

$$\sum_{j:x_j(t_1)=1} a_{i,j} = \sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} + \sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=1} a_{i,j} < 0 \tag{2.14}$$

De forma análoga, pela transição de $x_i(t_2) = 0$ para $x_i(t_2 + 1) = 1$, temos que:

$$\sum_{j:x_j(t_2)=1} a_{i,j} = \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} + \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=1} a_{i,j} > 0 \tag{2.15}$$

Subtraindo as duas últimas inequalidades, temos:

$$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0 \tag{2.16}$$

□

$x_i(t_1) \to x_i(t_1+1)$	$x_i(t_2) \to x_i(t_2+1)$	Restrições para $a_{i,j}$
$1 \to 0$	$0 \to 1$	$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$
$0 \to 1$	$0 \to 0$	$-\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} + \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$
$1 \to 1$	$0 \to 0$	$-\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} + \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} \leq 0$
$0 \to 0$	$0 \to 1$	$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$
$0 \to 1$	$1 \to 0$	$-\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} + \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$
$1 \to 1$	$1 \to 0$	$-\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} + \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$
$0 \to 0$	$1 \to 1$	$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} \leq 0$
$1 \to 0$	$1 \to 1$	$\sum_{j:x_j(t_1)=1 \text{ e } x_j(t_2)=0} a_{i,j} - \sum_{j:x_j(t_2)=1 \text{ e } x_j(t_1)=0} a_{i,j} < 0$

Tabela 2.4: Restrições geradas a partir da observação de pares de transição de estados (Higa *et al.*, 2011).

2.5 Seleção de Características

O passo de crescimento da semente, Seção 3.1, do algoritmo de inferência proposto por Higa *et al.* (Higa *et al.*, 2013) baseia-se no paradigma de crescimento da semente. Esse paradigma é visto como um problema de seleção de características. Nesta seção, apresentamos como esse problema é abordado formalmente e os algoritmos utilizados para resolver a seleção.

A seleção de característica é uma técnica conhecida na área de reconhecimento de padrões. Considerando X um conjunto total de características, o objetivo desse problema consiste em selecionar um subconjunto $Z \subset X$ utilizando algum critério. Nesse contexto, os genes são as características que desejamos selecionar, ou seja, $X = \{X_1, X_2, \ldots, X_n\}$. O paradigma de crescimento da semente consiste em "fazer crescer" uma semente de genes. Definimos uma semente de genes como sendo um subconjunto pequeno de genes e "crescer a semente" significa adicionar outros genes à semente, de acordo com algum critério estabelecido.

Os métodos de seleção de características são classificados em *métodos ótimos* e *sub-ótimos*. Os métodos ótimos, como a busca exaustiva, são apropriados quando o número de características é pequeno, pois esses métodos exploram todas as combinações do espaço de busca e é computacionalmente caro. Os métodos sub-ótimos, como a *Sequential Forward Selection* (SFS), são mais adequados quando o número de características é grande. Nesta seção, vamos

apresentar o método sub-ótimo *Improved Forward Floating Selection* (IFFS), que corresponde ao método *Sequential Forward Floating Selection* (SFFS) com um passo adicional de substituição de uma característica (Nakariyakul e Casasent, 2009).

2.5.1 Busca Sequencial

Antes de falarmos sobre o IFFS, vamos falar sobre os algoritmos que compõe o SFFS: o *Sequential Forward Selection* (SFS) e o *Sequential Backward Selection* (SBS).

Consideremos que o critério para avaliar as características é uma função critério J e que um valor maior de J é melhor do que um valor menor. O SFS constrói um subconjunto Z com d características, a partir de um subconjunto vazio. O algoritmo adiciona uma característica de modo incremental, ou seja, uma característica é adicionada por vez até o tamanho do subconjunto Z alcançar o valor d, essa abordagem é chamada de *bottom-up* (Whitney, 1971). O SBS realiza o processo inverso. O SBS começa com o conjunto X com todas as características e remove uma característica por vez até que o subconjunto resultante tenha tamanho d (abordagem *top-down*) (Marill e Green, 1963).

O problema desses dois algoritmos é que eles sofrem o efeito *nesting*. Por exemplo, se uma característica é adicionada no algoritmo SFS, ela não pode ser removida da solução do problema. Do mesmo modo, se o SBS descartar uma característica do conjunto, essa característica descartada não pode ser adicionada novamente. Para evitar esse efeito, foram propostos algoritmos em que as características são adicionadas e removidas sucessivamente, como no algoritmo (l, r) (Devijver e Kittler, 1982). Essa mesma ideia é utilizada em algoritmos de busca flutuante, como o SFFS (Pudil *et al.*, 1994).

2.5.2 Busca Sequencial Flutuante - SFFS

O algoritmo SFFS combina os algoritmos SFS e SBS para adicionar e remover uma característica. Um aspecto importante desse algoritmo é que a remoção só ocorre se o conjunto resultante for avaliado como melhor pela função critério. O IFFS inicia com um conjunto vazio e adiciona uma característica por vez utilizando o algoritmo SFS. Sempre que uma característica é adicionada ao conjunto, o algoritmo tenta remover outra de modo a encontrar um subconjunto melhor. Essa remoção é chamada de *remoção condicional*. O algoritmo para quando o tamanho do conjunto é Δ maior que d; dessa maneira, ainda é possível executar a remoção de características e preservar o tamanho d desejado do conjunto resultante. A Figura 2.6 mostra o fluxograma do algoritmo SFFS.

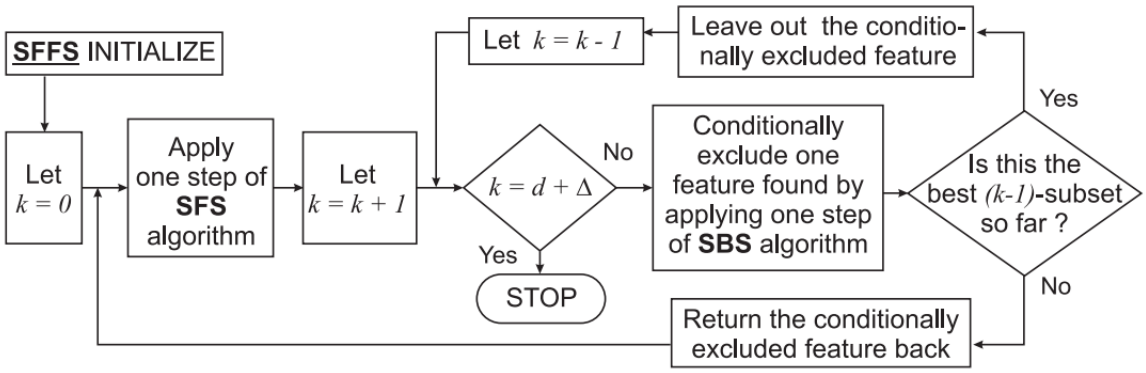


Figura 2.6: Fluxograma do algoritmo SFFS (Somol *et al.*, 2010).

2.5.3 Busca Sequencial Flutuante Melhorada - IFFS

O algoritmo *Improved Forward Floating Selection* (IFFS) (Nakariyakul e Casasent, 2009) é uma variação do algoritmo SFFS. Nessa versão, inclui-se um passo adicional para "substituir uma característica fraca" do conjunto. Esse passo adicional consiste em verificar se, ao substituir uma característica do subconjunto Z por outra do conjunto X, o valor da função critério J pode melhorar.

Assim como o SFFS, o IFFS começa com um conjunto vazio e para quando seu tamanho atinge $d + \Delta$ características. Os algoritmos SFS e SBS são utilizados para adicionar e remover características do subconjunto Z. Além disso, o algoritmo IFFS verifica se, ao trocar uma característica do subconjunto Z por uma outra do conjunto X, o valor da função critério aumenta para um subconjunto do mesmo tamanho.

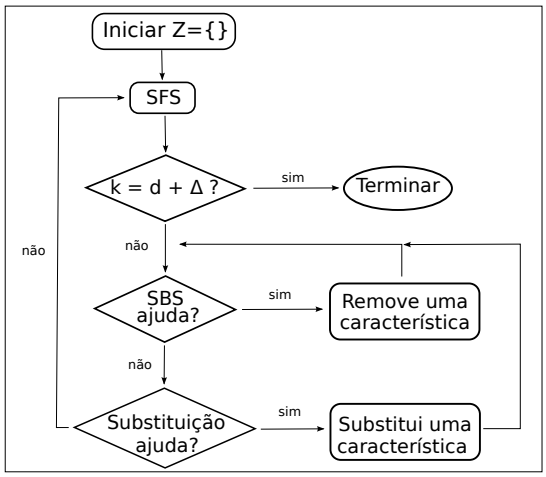


Figura 2.7: Fluxograma do algoritmo IFFS, figura adaptada (Nakariyakul e Casasent, 2009).

CAPÍTULO

3

Algoritmo de Inferência de Redes de Regulação Gênica

Este capítulo descreve o método de inferência proposto por Higa *et al.* (2013). Este método é baseado em uma heurística de seleção de característica para escolha de um conjunto de genes. Por se tratar de nossa referência base neste estudo, este capítulo relaciona-se fortemente com os trabalhos realizados por Higa *et al.* (2013; 2011). O método é executado em dois passos: crescimento da semente e inferência. Na Seção 3.1 é descrito o passo de crescimento da semente e na Seção 3.2 o passo de inferência.

3.1 Passo de Crescimento da Semente

O passo de crescimento da semente consiste em crescer um conjunto de genes S, chamado de semente, com o objetivo de encontrar outros genes que interagem com os genes da semente, formando algum processo biológico. A semente de genes $S = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ é um subconjunto pequeno com k genes do conjunto X.

O objetivo deste passo é identificar genes que tenham alguma interação com os genes da semente e adicioná-los a um subconjunto Z, tal que $Z = \{x_{i_{k+1}}, x_{i_{k+2}}, \ldots, x_{i_{k+d}}\}$, onde d (escolhido pelo usuário) é um valor pré-determinado de genes a serem adicionados em Z. Assim, a semente crescida é representada pelo subconjunto $G = S \cup Z$. Esse conjunto G é uma das entradas para o passo de inferência do algoritmo.

Percebe-se que o crescimento da semente é visto como um problema de

seleção de características e utiliza o algoritmo IFFS (Nakariyakul e Casasent, 2009) para selecionar os genes. A entrada para o algoritmo é uma semente de genes S, um conjunto X com todos os genes, um conjunto da semente crescida Z (inicialmente vazio), uma série temporal T e um parâmetro d para indicar o tamanho máximo do conjunto Z. O algoritmo de crescimento da semente é dividido em quatro passos e os passos de execução são vistos da seguinte maneira:

- *Passo 1 (Inicialização):* Iniciar o conjunto $Z = \emptyset$.
- *Passo 2 (SFS):* Encontrar o melhor gene candidato do conjunto $X - (S \cup Z)$ através da função critério. Se $|Z| = d$ ou se não encontrar nenhum candidato, então, terminar o algoritmo. Senão, ir para o Passo 3.
- *Passo 3 (SBS):* Esse passo consiste em encontrar o gene menos significativo $\bar{x}$ de Z e verificar se o conjunto $Z \setminus \bar{x}$ melhora caso esse gene menos significativo seja retirado. Se melhorar, o gene $\bar{x}$ é removido e se repete o Passo 3. Senão, ir para Passo 4.
- *Passo 4 (Substituição):* Substituir cada gene presente em Z por todos os genes candidatos em $X - (S \cup Z)$ e verificar se a substituição resulta em uma pontuação melhor do que a pontuação do conjunto Z atual. Se o conjunto melhorar, então, o gene é substituído e retorna-se ao Passo 3. Senão, ir para o Passo 2.

O tempo de complexidade do passo de crescimento da semente é dado pela complexidade da função critério para avaliar os genes, como descrito na Seção 3.1.4, e, no pior caso, temos $O((2^b.\binom{k+d}{b})^{k+d})$, onde $k+d$ é a quantidade de genes presentes em $G = S \cup Z$ e b é a conectividade de entrada máxima que um gene pode assumir. A saída do algoritmo é o subconjunto G com $k + d$ genes e é utilizado como entrada para o Passo de Inferência na Seção 3.2.

3.1.1 Consistência dos Dados

A consistência dos dados garante que a combinação do gene candidato x_c com os genes de $S \cup Z$ gerem redes booleanas limiarizadas com perturbação (TBNps) consistente, de acordo com a série temporal T. Assim, para um gene candidato qualquer x_c, podemos gerar matrizes consistentes, utilizando os três conjuntos de restrições, descritos na Seção 2.4, considerando apenas os genes presentes nos conjuntos $S \cup Z \cup \{x_c\}$. Uma TBNp consistente com o conjunto $S \cup Z$, dado por $|S| = k$ e $|Z| = l$, é uma matriz de regulação $A_{\ell \times \ell}$, onde $\ell = k+l+1$.

A desvantagem desse passo é que o número de soluções cresce exponencialmente a medida em que ℓ aumenta. Para calcular o número de soluções

e entropia, Higa *et al.* (2013) levaram em consideração apenas as soluções consistentes com a série de dados temporais $\mathbf{T}$.

3.1.2 Número de Matrizes de Regulação

O número de matrizes geradas é um dos parâmetros considerados pela função critério quando o gene candidato x_c é avaliado com a semente. Matematicamente, a quantidade de soluções geradas fornece grande variabilidade de soluções, quando o passo de inferência é executado. Biologicamente, a variabilidade possibilita que os mecanismos de regulação possam atuar de diversas maneiras com outras situações. Por isso, foi mantida essa variedade na possibilidade de soluções.

Nesse passo, introduz-se um parâmetro η, onde $0 < \eta < n$, para indicar se as conectividades de entrada mínima a e máxima b devem ser consideradas. A variável η é introduzida porque, quando ℓ é um número pequeno, poucas soluções podem ser encontradas se considerarmos a conectividade a e b. Então, quando $\ell < \eta$, todas as soluções são contadas, sem considerar os parâmetros de conectividade dos genes. Utilizamos $\eta = 5$.

Dado que um gene $x_i \in S \cup Z \cup \{x_c\}$ pode possuir m_i linhas consistentes, se selecionarmos uma linha para cada gene e montarmos uma matriz de regulação consistente A, o número total de matrizes consistentes é dado pelo seguinte produto (3.1).

$$\delta_c = \prod_{i=1}^{\ell} m_i \tag{3.1}$$

Se $\ell < \eta$, todas as soluções são contabilizadas, ou seja, o número máximo de matrizes $A_{\ell\times\ell}$ é 3^{ℓ^2}. Quando $\ell \geq \eta$, consideramos os parâmetros de conectividade a e b para cada linha r_i da matriz A. Assim, cada linha pode ter $2^j \binom{\ell}{j}$ configurações possíveis, para $j = a, a+1, \ldots, b$. O número máximo possível de matrizes para o conjunto $S \cup Z \cup \{x_c\}$ é dado pela seguinte Equação (3.2).

$$\Delta_c = \begin{cases} 3^{\ell^2}, & \text{se } \ell < \eta \\ \displaystyle\sum_{j=a}^{b} \left[2^j \binom{\ell}{j}\right]^{\ell}, & \text{se } \ell \geq \eta \end{cases} \tag{3.2}$$

A proporção de matrizes consistentes $\mathcal{P}(c)$ é dada pela Equação (3.3):

$$\mathcal{P}(c) = \begin{cases} 0, & \text{se } \delta_c = 0 \\ \lg \delta_c / \lg \Delta_c, & \text{outro caso} \end{cases} \tag{3.3}$$

É possível perceber que se $\delta_c = 1$, então $\mathcal{P}(c) = 0$, isso ocorre quando encontramos apenas uma solução consistente. Nesse caso, dizemos que o conjunto

$S \cup Z \cup \{x_c\}$ é ruim, uma vez que ele apresenta baixa variabilidade de soluções. Caso dois genes possuam $\Delta_{c_1} = \Delta_{c_2}$, o algoritmo vai escolher o candidato que apresentar maior proporção de matrizes consistentes $\mathcal{P}(c)$.

3.1.3 Entropia das Soluções

A entropia é uma métrica utilizada para medir o grau de incerteza das relações entre os genes presentes no conjunto $S \cup Z \cup \{x_c\}$. Por exemplo, quando uma aresta está presente em um gene x_i que ativa o gene x_j em toda solução consistente, logo essa aresta é dita requirida e não existe incerteza sobre ela. Portanto, o algoritmo tende a inferir relações entre os genes com baixa entropia das relações.

Essa métrica é associada a uma variável aleatória $X_{i,j}$ para os possíveis valores de conexão $\{-1, 0, 1\}$ do gene x_j para x_i, tal que $x_j, x_i \in S \cup Z \cup \{x_c\}$. Sejam $\alpha_{i,j}(c), \beta_{i,j}(c)$ e $\gamma_{i,j}(c)$, respectivamente, a quantidade de vezes que ocorrem conexões de ativação (1), sem conexão (0) e inibição (-1) do gene x_j para o gene x_i. Logo, a probabilidade de ocorrer uma conexão pode ser calculada sobre os possíveis valores que a variável aleatória $X_{i,j}$ assumir. Então, a probabilidade de o gene x_j ativar x_i dá-se pela Equação (3.4). De forma análoga, é possível calcular a probabilidade de o gene x_j não possuir conexão ou inibir x_i, $P(X_{i,j} = 0)$ e $P(X_{i,j} = -1)$.

$$P(X_{i,j} = 1) = \frac{\alpha_{i,j}(c)}{\alpha_{i,j}(c) + \beta_{i,j}(c) + \gamma_{i,j}(c)} \tag{3.4}$$

A entropia do gene x_j para x_i é calculada pela Equação (3.5):

$$H(X_{i,j}) = -\sum_{\lambda} P(X_{i,j} = \lambda) \lg(P(X_{i,j} = \lambda)) \tag{3.5}$$

onde $\lambda \in \{-1, 0, 1\}$. A entropia pode alcançar valores $H(X_{i,j}) < -\lg(1/3)$, o valor máximo é alcançado quando $H(X_{i,j}) = -\lg(1/3)$, isso ocorre se $\alpha_{i,j}(c) = \beta_{i,j}(c) = \gamma_{i,j}(c)$. Para maximizar a pontuação do gene com menor incerteza de relações, calcula-se a média de $-\lg(1/3) - H(X_{i,j})$ para todos os genes x_j, pela Equação (3.6):

$$H_i = \frac{1}{\ell} \sum_{j} -[\lg(1/3) + H(X_{i,j})] \tag{3.6}$$

Portanto, a entropia média das soluções para o conjunto $S \cup Z \cup \{x_c\}$ pode ser obtida por meio de (3.7):

$$\mathcal{H}(c) = \frac{1}{\ell} \sum_{i} H_i \tag{3.7}$$

3.1.4 Função Critério

O gene candidato x^* é selecionado por meio do maior valor da função critério dentre todos os genes candidatos $x_c \in X - (S \cup Z)$, de acordo com a Equação (3.8):

$$x^* = \arg\max_{x_c \in X-(S \cup Z)} J(c), \tag{3.8}$$

O cálculo da função critério $J(c)$, na Equação (3.9), considera os dados vindos da expressão gênica do gene candidato x_c com a semente de genes S e o conjunto atual de genes Z. O gene candidato selecionado será o que apresentar maior variabilidade de redes TBNps consistentes e menor incerteza das relações entre os genes. Obtém-se a pontuação da função critério J a avaliando o número de matrizes consistentes (Seções 3.1.1 e 3.1.2) e a entropia das soluções (Seção 3.1.3) com o gene candidato x_c.

$$J(c) = \omega \mathcal{P}(c) + (1 - \omega)\mathcal{H}(c) \tag{3.9}$$

A função critério avalia o conjunto $S \cup Z \cup \{x_c\}$, onde $\mathcal{P}(c)$ é o número de soluções consistentes; $\mathcal{H}(c)$ é a entropia das soluções e ω é um fator de balanço entre $\mathcal{P}(c)$ e $\mathcal{H}(c)$. Se $\omega = 1$, o algoritmo favoreceria os candidatos com mais quantidade de soluções. Se $\omega = 0$, o algoritmo seria a favor dos genes com baixa entropia sem se preocupar com a quantidade de soluções geradas pelo conjunto. Por essas razões, $\omega = 0.5$.

3.2 Passo de Inferência

O objetivo do passo de inferência consiste em construir uma rede que explique os dados de entrada. Entretanto, é possível gerar diversas redes quem explicam os dados de entrada. A entrada para o algoritmo de inferência é uma série de dados temporais **T**, um subconjunto de genes $G = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$, obtido na saída do passo de crescimento da semente, um parâmetro $\mathcal{M}$ definido pelo usuário para indicar a quantidade de redes a serem geradas, a conectividade de entrada mínima a e máxima b que um gene pode ter e escolher entre gerar redes contendo ou não um gene hub (gene que possui grande conectividade de saída).

Cada gene x_i pode possuir diversas linhas consistentes, formando um conjunto de linhas consistentes $R_i = \{r_i^1, r_i^2, \ldots, r_i^{m_i}\}$. Uma TBNp gerada a partir do subconjunto G é uma matriz de regulação $A_{k \times k}$, onde cada linha r_i da matriz A é uma solução para o gene $x_i \in G, i = i_1, i_2, \ldots, i_k$, obtida mediante restrições apresentadas na Seção 2.4. Dado que a matriz possui k genes, a quantidade

de matrizes A que podem ser geradas é 3^{k^2}. Vale destacar que as linhas são geradas independentemente, cada linha da matriz A pode ter 3^k configurações e como temos k linhas, são testados $k \times 3^k$ possibilidades.

Considerando que o gene $x_i \in G$ tem m_i linhas consistentes, o número total de matrizes consistentes (TBNps) que podem ser geradas é dado pela Equação (3.10):

$$\prod_{i=1}^{k} m_i \tag{3.10}$$

O passo de inferência realiza um procedimento de amostragem nas redes consistentes e analisa a estrutura e a dinâmica de cada rede. A análise da rede consiste em verificar a conectividade e quantidade de componentes conectados (análise da estrutura) e os atratores e as bacias de atração (análise da dinâmica) da rede. Os passos de execução do algoritmo são vistos da seguinte maneira:

- *Passo 1:* Encontrar R_i linhas consistentes para cada gene $x_i \in G$ e considerar uma perturbação no tempo t, se for o caso.
- *Passo 2:* Selecionar aleatoriamente uma linha consistente de R_i para cada gene $x_i \in G$, considerando as conectividade a e b e a possibilidade de selecionar um gene hub, e montar uma matriz de regulação aleatória com esses dados.
- *Passo 3:* Verificar se a rede montada possui apenas um componente. Se tiver, ir para o Passo 4. Senão, retornar ao Passo 2.
- *Passo 4:* Identificar os atratores e calcular a entropia da rede.
- *Passo 5:* Verificar se o número de redes montadas é igual a $\mathcal{M}$. Se for igual, ir para o Passo 6. Senão, retornar ao Passo 2.
- *Passo 6:* Selecionar 10% das redes que possuem menor valor de entropia.

Assim como no passo de crescimento da semente, o passo de inferência utiliza a abordagem CSP para gerar as R_i linhas consistentes do gene $x_i \in G$. No pior caso do passo 1, temos $O((2^b.\binom{k}{b})^k)$, onde b é a conectividade de entrada máxima e k é o número de genes em G. Para analisar a dinâmica da rede no passo 4, consideram-se todos os possíveis estados que a rede pode assumir; no pior caso, o passo 4 custa $O(2^k)$. A saída do algoritmo é um conjunto de redes TBNps com baixa entropia, ou seja, baixa incerteza sobre suas relações.

Os parâmetros de conectividade de entrada mínima e máxima de um gene são descritos na seção 3.2.1. Na seção 3.2.2, descreve-se a quantidade de componentes conectados. Na seção 3.2.3, são descritas as bacias de atração e os atratores da rede. E na seção 3.2.4, é descrita a entropia utilizada para avaliar a rede gerada.

3.2.1 Conectividade

A conectividade k_i do gene x_i representa o número de genes que ativam ou inibem o gene x_i e, consequentemente, fazem parte da função booleana f_i para o gene x_i. Em outras palavras, k_i é o número de elementos não-zeros da i-ésima linha da matriz de regulação A. A conectividade de entrada dos genes é configurada para ter valores mínimo (a) e máximo (b), de modo que $a \leq k_i \leq b$, onde $i = 1, \ldots, k$ e $0 < a \leq b$.

3.2.2 Componentes Conectados

O grafo induzido pela matriz de regulação pode apresentar mais de um componente conexo, essa verificação é realizada por meio de um algoritmo de busca em largura (BFS) no grafo induzido. É mais interessante inferir redes onde o grafo induzido pela matriz de regulação apresenta apenas um componente conexo do que inferir redes com vários componentes conexos. Inferir matrizes em que os grafos induzidos possuem apenas um componente conexo representa inferir redes em que existe somente um mecanismo de regulação principal. A presença de mais de um componente conexo significa inferir redes com diversos módulos de regulação, não constituindo o objetivo do trabalho.

3.2.3 Atratores e Bacias de Atração

O interesse em analisar a dinâmica da rede é inferir redes que são potencialmente significativas no sentido biológico. A análise da dinâmica da rede consiste em identificar os atratores estáveis. Um atrator é dito estável se esse possuir grandes bacias de atração e é formado por uma baixa quantidade de estados. Os estados que não são atratores, mas que se dirigem a um atrator, formam as bacias de atração de um atrator (Kauffman, 1969).

Em sistemas com poucos atratores, é natural que haja grandes bacias de atração. Em tais sistemas, se ocorrer uma perturbação, há uma grande chance de a rede ser levada para algum outro estado da mesma bacia e alcançar o mesmo atrator. Por essa razão, redes com poucos atratores e grandes bacias de atração são ditas como estáveis (Kauffman, 1969).

Considerando uma matriz de regulação A, os atratores dessa matriz são $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_r$ e suas respectivas bacias de atração são $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_r$. O número de estados do atrator $\mathcal{A}_i$ com bacia $\mathcal{B}_i$ é dado por $|\mathcal{A}_1|(|\mathcal{B}_i|)$. A probabilidade de encontrarmos um estado arbitrário na rede com bacia $\mathcal{B}_i$, onde $i = 1, 2, \ldots, r$, se dá por (3.11):

$$P(Y_i = 1) = \frac{|\mathcal{B}_i|}{\sum_{j=1}^{r} |\mathcal{B}_j|}, \tag{3.11}$$

onde Y_i é uma variável binária aleatória para indicar que o estado da bacia $\mathcal{B}_i$ é $(Y_i = 1)$.

3.2.4 Entropia

A entropia é uma medida de incerteza associada a uma variável aleatória. Nesse caso, a entropia mede o quão estável é a rede gerada. A entropia é calculada por meio da probabilidade de encontrar um estado arbitrário na bacia de atração $\mathcal{B}_i$, para $i = 1, 2, \ldots, r$, da seguinte maneira:

$$H_A = -\sum_{i=1}^{r} P(Y_i = 1) \lg[P(Y_i = 1)]. \tag{3.12}$$

A entropia alcança valor máximo quando todas as bacias têm o mesmo tamanho, o que representa uma péssima solução. Se $|\mathcal{B}_1| \gg |\mathcal{B}_2|$, isso indica que uma bacia possui mais estados que a outra. O algoritmo favorece redes que apresentam baixa entropia; logo essas redes tendem a ter uma bacia com grande atração e outras bacias menores com pouca atração.

CAPÍTULO

4

Implementações Paralelas

Este capítulo descreve os detalhes das três implementações paralelas do algoritmo de crescimento da semente o qual é baseado no método de seleção IFFS. A primeira implementação utiliza *cluster* de CPUs que se comunicam por meio de trocas de mensagens; a segunda utiliza uma GPU e a terceira é implementada em um ambiente híbrido, onde o processamento é alternado entre a execução em CPUs e em GPUs. Além disso, este capítulo detalha, ainda, a implementação do passo de inferência, que usa um *cluster* de CPUs.

A abordagem com CPUs utiliza a biblioteca Gecode[1] em cada CPU para resolver as restrições e gerar as linhas consistentes. Essa biblioteca não pode ser utilizada pela GPU, pois suas funções de busca são inerentemente sequenciais e não utilizam os mesmos conceitos e propósitos da GPU. Considerando as dificuldades de paralelização dos métodos empregados pelo Gecode em GPU, reescrevemos funções para serem executadas em GPU, com objetivos similares às funções encontradas no Gecode.

A semente S, conjunto de genes X, série temporal $\mathbf{T}$ e um valor d constituem parâmetros que são definidos na entrada do programa pelo usuário. Esses parâmetros estão explicados na Seção 3.1. O conjunto de genes candidatos C é formado pelo conjunto X, que contém todos os genes, sem aqueles que estão na semente S, ou seja, o conjunto de genes candidatos C é obtido por meio de $C = X - S$. A semente S é totalmente distribuída para todas as unidades de processamento antes mesmo de iniciar a seleção de característica.

[1] biblioteca em C++ que resolve problemas de satisfação de restrição, disponibilizando modelos de variáveis, funções de busca e propagadores de restrição.

4.1 Abordagem Utilizando Cluster de CPUs

4.1.1 Crescimento da Semente

Este passo consiste em determinar para cada gene, dentro do conjunto de genes candidatos, uma pontuação entre ele e um conjunto de genes dos quais já se conhecem as relações, a semente. Como o objetivo é de conhecer as redes de interação entre esses genes, o conjunto de genes é dividido entre as unidades de processamento, sendo que cada um é responsável por determinar o melhor gene dentre aqueles que lhes foram atribuídos. Desse modo, cada CPU terá o melhor gene da sua parte do conjunto. Para fazer a seleção de genes, o crescimento da semente utiliza o algoritmo IFFS que é composto pelos algoritmos: *Sequential Forward Selection* – SFS, *Sequential Backward Selection* – SBS – e um passo de substituição. Tais métodos são abordados utilizando *cluster* de CPUs e são descritos separadamente nas seções seguintes.

4.1.1.1 Sequential Forward Selection - SFS

O método de seleção SFS identifica um gene candidato que melhor interage com os genes da semente e do conjunto Z. A entrada do Algoritmo 1 é um conjunto de genes da semente S, um conjunto de genes Z que, inicialmente, é vazio, um conjunto de genes candidatos C e a série temporal $\mathbf{T}$.

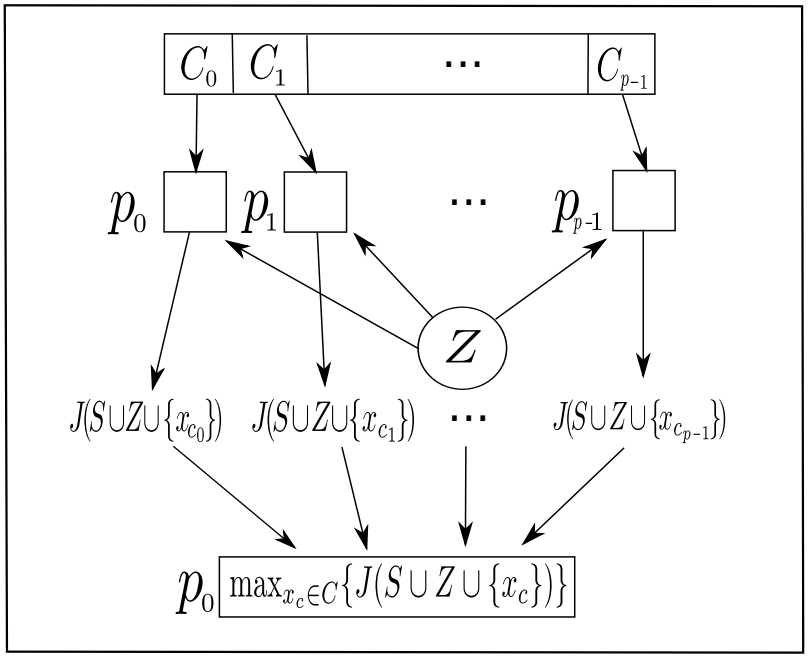


Figura 4.1: SFS utilizando *cluster* de CPUs.

A abordagem proposta está ilustrada na Figura 4.1. Esse método inicia com o nó mestre distribuindo o conjunto de genes Z para todos os nós. Em seguida, o nó mestre distribuí o conjunto de genes candidatos C entre os p nós

do *cluster*, de modo que o subconjunto C_i seja atribuído ao nó p_i. O próximo passo, executado por todos os nós, é encontrar o melhor gene candidato x_c dentre todos os genes presentes no subconjunto C_i entre a linha 5 até 13. O gene candidato x_c é adicionado temporariamente ao conjunto Z para calcular a pontuação de $S \cup Z$. A pontuação encontrada é comparada com a maior pontuação já obtida entre os genes presentes em C_i.

Algoritmo 1: Descrição do algoritmo SFS_CPUs

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene x_c do conjunto C que possui maior pontuação para o conjunto $S \cup Z \cup \{x_c\}$.
Resultado: Insere o melhor gene candidato no conjunto Z.

```
1  distribuir todo o conjunto Z para todos os nós;
2  distribuir C igualmente entre os p nós, de forma que o subconjunto C_i
   seja atribuído ao nó p_i;
3  score_local ← 0.0;
4  gene_local ← −1;
5  foreach x_c ∈ C_i do
6      Z = Z ∪ {x_c} ;                          /* adicionar x_c em Z */
7      score ← J(S ∪ Z);
8      if score > score_local then
9          gene_local ← x_c;
10         score_local ← score;
11     end
12     Z = Z \ {x_c} ;                          /* remover x_c em Z */
13 end
14 determinar x_c = max_{x_c∈C}{J(S ∪ Z ∪ {x_c})};
15 C = C \ {x_c};
16 Z = Z ∪ {x_c};
```

Quando o melhor gene de cada subconjunto C_i é conhecido, o nó mestre determina a maior pontuação entre todos os nós na linha 14. Essa pontuação corresponde ao melhor gene candidato x_c de todo o conjunto C. Após o melhor gene candidato ser encontrado, o gene x_c é removido de C e adicionado em Z.

4.1.1.2 Sequential Backward Selection - SBS

O método SBS verifica se o conjunto Z (conjunto de genes que já foram adicionados) melhora quando se remove um gene dele, ou seja, se a pontuação do conjunto de tamanho $|S \cup Z| - 1$ é maior em relação ao conjunto de tamanho $|S \cup Z|$. A entrada para esse método é o conjunto de genes S, Z, C e a série temporal **T**.

O Algoritmo 2 se inicia com o nó mestre distribuindo todo o conjunto Z e sua pontuação para todos os nós. O próximo passo é distribuir o conjunto Z entre os p nós do *cluster*, de modo que o subconjunto Z_i seja atribuído ao nó

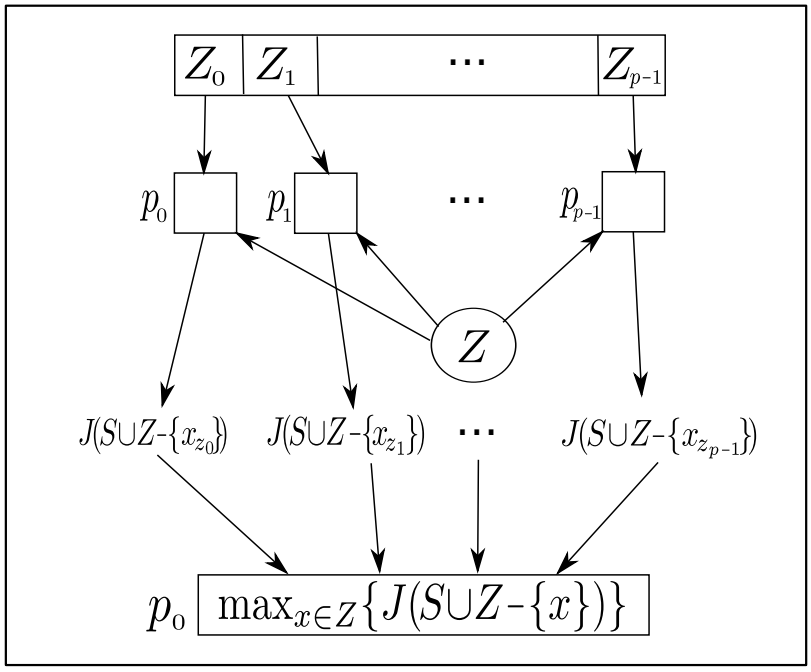


Figura 4.2: SBS utilizando *cluster* de CPUs.

p_i. O subconjunto Z_i contém os genes que devem ser removidos do conjunto Z para calcular a função critério do conjunto $S \cup Z - \{x\}$, onde $x \in Z_i$. A variável $score\_local$ inicia recebendo a pontuação para o conjunto da semente crescida e a variável $gene\_fraco\_local$ inicia com valor -1, para indicar que não foi encontrado nenhum gene fraco do subconjunto Z_i. Em seguida, inicia-se o cálculo da função critério sem os genes em Z_i. Os nós removem temporariamente um gene x de Z, calculam a pontuação sem esse gene e comparam com a maior pontuação obtida até o momento na linha 5 à 13. Após cada nó encontrar o gene mais fraco de cada subconjunto Z_i, o nó mestre determina qual o gene mais fraco entre os subconjuntos Z_i, e esse gene representa o mais fraco de todo o conjunto Z. Então, o gene x é removido de Z e adicionado novamente no conjunto de genes candidatos C. A Figura 4.2 ilustra o funcionamento do Algoritmo 2.

4.1.1.3 Passo de Substituição

O crescimento da semente possui um passo adicional que corresponde a um método de substituição. Esse método consiste em tentar encontrar a melhor forma, ou pontuação, de um conjunto. Essa técnica substitui todos os genes presentes no conjunto Z pelos genes que estão no conjunto de genes candidatos C, de forma tal que aumente a pontuação sem alterar o tamanho do conjunto Z.

O passo de substituição, descrito no Algoritmo 3, é ilustrado na Figura 4.3, começa com o nó mestre transmitindo os conjuntos Z, C e a pontuação do conjunto Z para todos os nós. Na sequência, o nó mestre distribuí o conjunto

Algoritmo 2: Descrição do algoritmo SBS_CPUs

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene fraco do conjunto Z.
Resultado: Remoção ou não do gene fraco do conjunto Z.

```
1  distribuir todo o conjunto Z e sua pontuação para todos os nós;
2  distribuir Z igualmente entre os p nós, de forma que o subconjunto Z_i
   seja atribuído ao nó p_i;
3  score_local ← J(S ∪ Z);
4  gene_fraco_local ← −1;
5  for each x ∈ Z_i do
6      Z = Z \ {x};
7      score ← J(S ∪ Z);
8      if score > score_local then
9          score_local ← score;
10         gene_fraco_local ← x;
11     end
12     Z = Z ∪ {x};
13 end
14 determinar x = max_{x∈Z}{J(S ∪ Z − {x})};
15 Z = Z \ {x};
16 C = C ∪ {x};
```

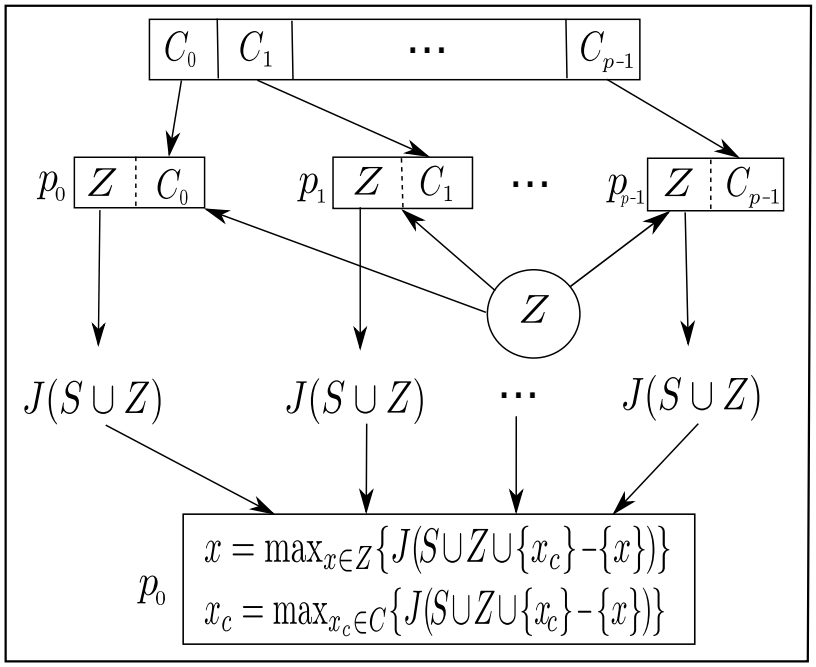


Figura 4.3: Passo de substituição utilizando *cluster* de CPUs.

C entre p nós, de forma que o subconjunto C_i seja atribuído ao nó p_i. A variável $score\_local$ armazena a pontuação de $S \cup Z$ e as variáveis $gene\_fraco\_local$ e $gene\_forte\_local$ guardam, respectivamente, um gene fraco $x \in Z$ e um gene forte $x_c \in C_i$. Em seguida, os nós calculam a pontuação de $S \cup Z$ substituindo o gene $x \in Z$ por um gene candidato $x_c \in C_i$, comparando com o valor da variável $score\_local$. Se a pontuação desse novo conjunto for maior, o gene x é

considerado como um gene fraco e x_c como um gene forte. Depois que os genes de todos os C_i foram avaliados, o nó mestre determina a maior pontuação do melhor gene candidato x_c que melhor substituiu o gene x do conjunto Z. Por fim, o nó mestre substitui o gene x pelo gene x_c em Z e substitui o gene x_c pelo gene x no conjunto C.

Algoritmo 3: Descrição do algoritmo $Substituicao\_CPUs$

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene fraco x do conjunto Z e um gene candidato x_c de C.
Resultado: Substituir o gene x por x_c somente se a pontuação obtida com o gene $x_c \in Z$ é maior que a pontuação do conjunto Z com o gene x.

distribuir todo o conjunto Z e sua pontuação para todos os nós;
distribuir C igualmente entre os p nós, de forma que o subconjunto C_i seja atribuído ao nó p_i;
$score\_local \leftarrow J(S \cup Z)$;
$gene\_fraco\_local \leftarrow -1$;
$gene\_forte\_local \leftarrow -1$;
for *each* $x \in Z$ **do**
    foreach $x_c \in C_i$ **do**
        trocar x por x_c em Z;
        $score \leftarrow J(S \cup Z)$;
        if $score > score\_local$ **then**
            $gene\_fraco\_local \leftarrow x$;
            $gene\_forte\_local \leftarrow x_c$;
            $score\_local \leftarrow score$;
        end
        trocar x_c por x em Z;
    end
end
determinar gene fraco $x = \max_{x \in Z}\{J(S \cup Z \cup \{x_c\} - \{x\})\}$;
determinar gene forte $x_c = \max_{x_c \in C}\{J(S \cup Z \cup \{x_c\} - \{x\})\}$;
substituir x por x_c em Z;
substituir x_c por x em C;

4.1.2 Inferência

O passo de inferência consiste em montar redes de regulação gênica a partir dos dados de entrada. O passo de inferência é um procedimento de amostragem realizado para montar $\mathcal{M}$ redes geradas aleatoriamente, dos quais 10% das redes com menor entropia representam a saída do algoritmo. Essa tarefa é distribuída entre os p nós do *cluster*, cada um dos nós é responsável por montar $\mathcal{M}/p$ redes. O passo de inferência está descrito no Algoritmo 4.

O primeiro passo é calcular novamente a função critério para a semente crescida; porém, dessa vez, as linhas consistentes de cada gene são armaze-

Algoritmo 4: Descrição do algoritmo $Inferencia\_CPUs$

Entrada: Conjunto de genes da semente crescida e um parâmetro $\mathcal{M}$.
Saída: Uma lista com 10% de redes com baixa entropia.

```
1  encontrar as linhas consistentes para ∀x_i ∈ S ∪ Z;
2  distribuir as linhas consistentes para todos os nós;
3  redes_solicitadas ← M/10;
4  redes_para_montar ← M/p;
5  for redes_geradas = 1 to redes_para_montar do
6      A ← montar matriz aleatória;
7      verificar se matriz A possui apenas um componente conexo;
8      entropia_rede ← calcular entropia H(A);
9      if redes_geradas < redes_solicitadas then
10         armazenar a matriz A no vetor;
11         if redes_geradas = redes_solicitadas then
12             construir heap;
13         end
14     end
15     else
16         if entropia_rede < entropia_raiz_do_heap then
17             trocar a rede que está na raiz do heap pela nova rede montada;
18             reconstruir heap;
19         end
20     end
21 end
22 ordenar o vetor com heapsort;
23 determinar os 10% de redes com baixa entropia com o nó mestre;
```

nadas na memória do mestre e distribuídas para todos os nós, nas linhas 1 e 2 do Algoritmo 4. Em seguida, todos os nós montam $\mathcal{M}/p$ redes, selecionando aleatoriamente uma linha consistente para cada linha da matriz. A rede montada é representada por uma matriz quadrada e essa matriz é considerada apenas se apresentar um componente conectado; essa verificação é realizada por meio do algoritmo de busca em largura (BFS) no grafo induzido da matriz gerada.

As redes criadas pelos nós estão armazenadas em um vetor de tamanho igual a 10% de $\mathcal{M}$. Quando o limite do vetor é alcançado, a estrutura de dados é organizada como um *heap*, na linha 12. Desse modo, a raiz do *heap* sempre vai manter a maior entropia da rede armazenada, ou seja, a pior rede armazenada até o momento. Então, compara-se a próxima rede montada apenas com a raiz do *heap* e, se for melhor, a pior rede armazenada é removida e a nova rede é adicionada na raiz do *heap* (linha 17), para, então na linha 18, reconstruir o *heap*. A Figura 4.4 ilustra o trabalho realizado pelos nós no passo de inferência.

Na linha 22, após todos os nós montarem as $\mathcal{M}/p$ redes, o vetor é ordenado de forma crescente, utilizando *heapsort* pelos nós. Após a ordenação do vetor

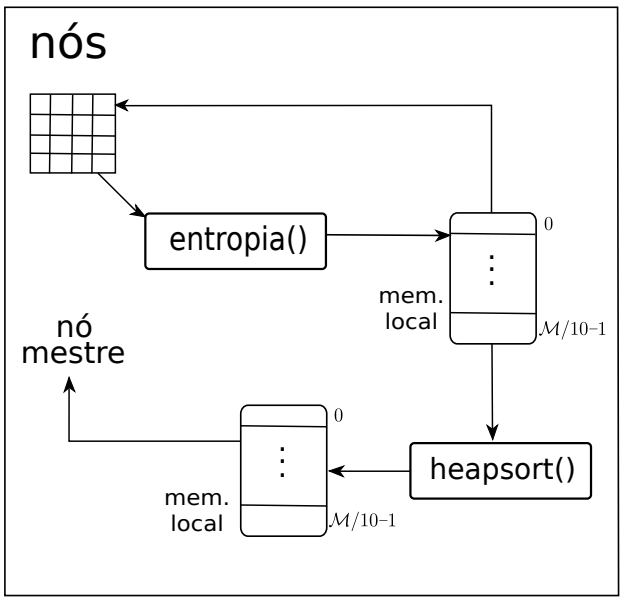


Figura 4.4: Funcionamento do nós no passo de inferência utilizando *cluster* de CPUs.

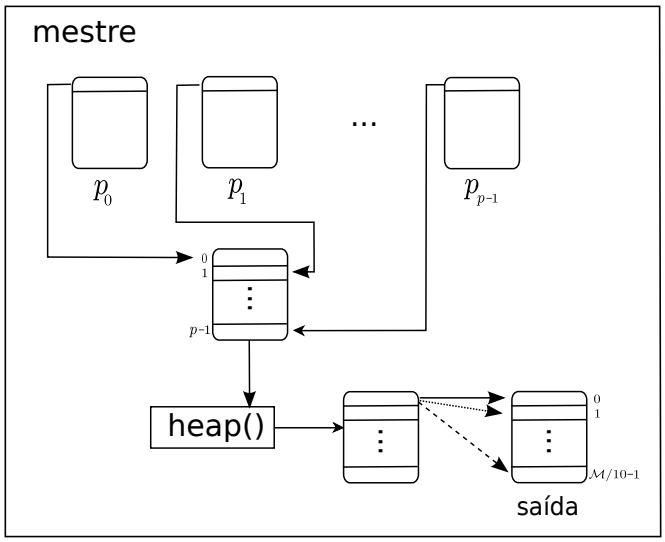


Figura 4.5: Determinando as redes com baixa entropia no passo de inferência utilizando *cluster* de CPUs.

de redes pelos nós, o nó mestre determina as redes com baixa entropia na linha 23. Para determinar as melhores redes entre todos os nós, o nó mestre junta as redes geradas pelos outros nós e cria uma estrutura de *min-heap* com a rede de menor entropia de cada nó. Assim, após a manutenção do *min-heap*, a rede que estiver na raiz será aquela com a menor entropia entre todos os nós. Em seguida, remove-se a rede do *heap* que é selecionada como umas redes para a saída do algoritmo. O número de redes com baixa entropia e o índice que corresponde à rede que estava na raiz são incrementados. E o processo se repete até que os 10% de redes com baixa entropia sejam encontrados. Esse esquema está representado na Figura 4.5.

4.2 Abordagem Utilizando GPU

Essa implementação explora a paralelização do cálculo da função critério, descrito na Seção 3.1.4, dos genes para o conjunto $S \cup Z$. Nessa paralelização, o cálculo da pontuação é distribuído entre as *threads* do bloco; assim, cada gene do conjunto $S \cup Z$ é atribuído a uma *thread* e o bloco de *threads* realiza o processamento do conjunto de genes. O crescimento da semente com a abordagem para a execução na GPU precisou de funções extras que exercessem funcionalidades semelhantes às funções utilizadas pela biblioteca Gecode. Essas funções extras são abordadas na Seção 4.2.2.1.

4.2.1 Crescimento da Semente

A série temporal é um dos parâmetros de entrada para a seleção dos genes. Devido à realização de diversas operações de leitura sobre esses dados, a série é armazenada na memória de textura da GPU e reduzida com somente os genes presentes no conjunto de genes $S \cup Z$.

	$x_1(t)$	$x_2(t)$	$x_3(t)$	$x_4(t)$
$s(1)$	1	1	0	0
$s(2)$	1	0	0	0
$s(3)$	1	0	1	0
$s(4)$	1	0	1	1
$s(5)$	0	0	1	1

Tabela 4.1: Exemplo de série temporal com 4 genes.

Considere uma série temporal T com 4 genes e 5 instantes de tempo (Tabela 4.1) e o conjunto $S \cup Z = \{x_1, x_2, x_4\}$. A série simplificada ou reduzida é composta somente com os valores da expressão de genes que estão no conjunto $S \cup Z$, como mostrado na Tabela 4.2. Removemos transições de estados repetidos $s(t)$ e $s(t+1)$, uma vez que não houve tempo suficiente para ocorrerem mudanças nas expressões dos genes como nos instantes $s(2)$ e $s(3)$ da Tabela 4.2.

	$x_1(t)$	$x_2(t)$	$x_4(t)$
$s(1)$	1	1	0
$s(2)$	1	0	0
$s(3)$	1	0	0
$s(4)$	1	0	1
$s(5)$	0	0	1

Tabela 4.2: Série temporal reduzida com os genes de $S \cup Z$.

Como a memória de textura possui restrições em seu tamanho e são realizadas diversas comparações na série temporal, decidimos converter a série binária para decimal (Tabela 4.3) e utilizar operações *bit-a-bit* para realizar as comparações de forma mais rápida.

	$x_1, x_2, x_4(t)$
$s(1)$	6
$s(2)$	4
$s(3)$	5
$s(4)$	1

Tabela 4.3: Série temporal reduzida e convertida em decimal.

A Figura 4.6 mostra o fluxo de instruções utilizado pelos algoritmos SFS, SBS e passo de substituição na abordagem para uma única GPU. As diferenças entre esses três algoritmos estão nas funções A e B realizadas pela CPU. Por exemplo, para a execução do SFS, as funções A são formadas pela função de produzir as séries temporais T adicionando um gene candidato $x_c \in C$ em $S \cup Z$ e pelas funções de gerenciamento de memória (alocação de variáveis e transferência de dados) do *host* (CPU) para a memória do *device* (GPU). As funções B são funções que trazem as pontuações dos diferentes conjuntos $S \cup Z$ localizados na memória da GPU para a memória da CPU e funções para determinar o gene candidato x_c que têm a maior pontuação calculada, removê-lo de C e adicioná-lo em Z.

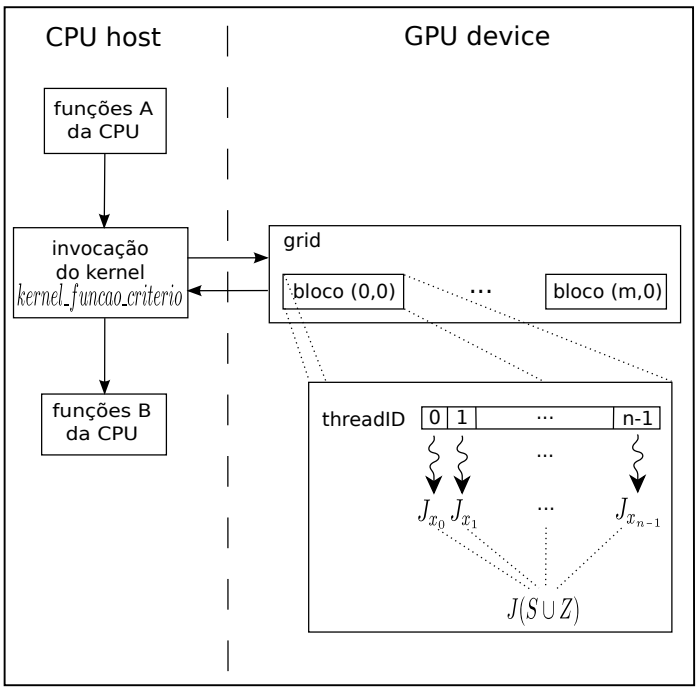


Figura 4.6: Fluxo de instruções dos algoritmos SFS, SBS e passo de substituição utilizando uma GPU.

4.2.1.1 Sequential Forward Selection - SFS

O método de seleção SFS na GPU calcula a pontuação de cada gene candidato $x_c \in C$ adicionando-o ao conjunto $S \cup Z$. O Algoritmo 5 mostra a implementação utilizando CPU e GPU. Inicialmente, produzimos todas as séries temporais possíveis com os genes do conjunto $S \cup Z \cup \{x_c\} \mid \forall x_c \in C$ e transferimos para a memória da GPU. Em seguida, executamos o *kernel* $kernel\_funcao\_criterio$, descrito na Seção 4.2.2, com $|C|$ blocos de *threads* de tamanho igual a $|S \cup Z| + 1$ para calcular a pontuação do gene $x_c \in C$ com o conjunto $S \cup Z$. Nesse momento, na linha 7, a CPU é bloqueada pela instrução de sincronização até o *kernel* terminar de calcular a função critério $J(S \cup Z)$. Após o cálculo da função critério na GPU, as pontuações dos genes candidatos são transferidas da GPU para CPU. O melhor gene candidato x_c é determinado pela maior pontuação entre todos os genes candidatos. Após determinar o melhor gene candidato, o gene x_c é removido do conjunto C e inserido no conjunto Z.

Algoritmo 5: Descrição do algoritmo SFS_GPU

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene candidato x_c do conjunto C que possui maior pontuação para o conjunto $S \cup Z \cup \{x_c\}$.
Resultado: Adiciona o melhor gene candidato ao conjunto Z.

1 produzir as séries temporais $T_{S \cup Z \cup \{x_c\} \mid \forall x_c \in C}$;
2 transferir $T_{S \cup Z \cup \{x_c\}}$ para memória da GPU;
3 $n\_threads \leftarrow |S \cup Z| + 1$;
4 $n\_blocos \leftarrow |C|$;
5 alocar um vetor dev_score na memória global da GPU;
6 $kernel\_funcao\_criterio <<< n\_blocos,\ n\_threads >>> (dev\_score)$;
7 $cudaDeviceSynchronize()$;
8 transferir $dev\_score(S \cup Z \cup \{x_c\})$, onde $x_c \in C$, da GPU para CPU em $host\_score$;
9 determinar $x_c = \max_{x_c \in C}\{host\_score(S \cup Z \cup \{x_c\})\}$;
10 $C = C \setminus \{x_c\}$;
11 $Z = Z \cup \{x_c\}$;

4.2.1.2 Sequential Backward Selection - SBS

O método SBS calcula a pontuação do conjunto $S \cup Z$ removendo cada gene x, presente no conjunto Z. Na abordagem utilizando GPU, o Algoritmo 6 inicia produzindo uma série temporal para todos os conjuntos possíveis de $S \cup Z$, removendo apenas um gene $x \in Z$ e transferimos para a memória da GPU. Em seguida, o *kernel* $kernel\_funcao\_criterio$ é executado com $|Z|$ blocos de *threads* e cada bloco tem $|S \cup Z| - 1$ *threads* para calcular a pontuação do conjunto $S \cup Z - \{x\} \mid x \in Z$. O algoritmo espera o *kernel* $kernel\_funcao\_criterio$ ser

finalizado. Em seguida, a pontuação dos conjuntos $S \cup Z - \{x\}$ é transferida da GPU para CPU. O gene mais fraco x é determinado através da maior pontuação para o conjunto $S \cup Z - \{x\}$. A pontuação do conjunto $S \cup Z - \{x\}$ é comparada com a pontuação do conjunto $S \cup Z$, e o gene x é removido de Z se a pontuação de $S \cup Z - \{x\}$ é maior que a pontuação de $S \cup Z$.

Algoritmo 6: Descrição do algoritmo SBS_GPU

Entrada: Conjunto de genes S, C e Z e a série temporal T.
Saída: Um gene x considerado o mais fraco do conjunto Z.
Resultado: Remove ou não o gene fraco do conjunto Z.

```
1  produzir as séries temporais T_{S∪Z−{x_i} | ∀x_i∈Z};
2  transferir T_{S∪Z−{x_i}} para memória da GPU;
3  n_threads ← |S ∪ Z| − 1;
4  n_blocos ← |Z| ;
5  alocar um vetor dev_score na memória global da GPU;
6  kernel_funcao_criterio <<< n_blocos, n_threads >>> (dev_score);
7  cudaDeviceSynchronize();
8  transferir dev_score(S ∪ Z − {x_i}), onde x_i ∈ Z, da GPU para CPU em
   host_score;
9  determinar x = max_{x_i∈Z}{host_score(S ∪ Z − {x_i})};
10 if J(S ∪ Z − {x}) > J(S ∪ Z) then
11     Z = Z \ {x};
12     C = C ∪ {x};
13 end
```

4.2.1.3 Passo de Substituição

O passo de substituição na abordagem para a GPU substitui todos os genes do conjunto Z por todos os genes candidatos do conjunto C. Cada bloco de *thread* utilizado nessa abordagem calcula a pontuação de um conjunto diferente de $S \cup Z$. O Algoritmo 7 inicia produzindo todas as séries temporais, realizando a substituição, com apenas os valores das expressões gênicas dos genes em $S \cup Z$ e transferimos as séries produzidas para a memória da GPU. Em seguida, definimos o tamanho dos blocos e a quantidade de blocos de *threads* que será utilizado pelo *kernel* $kernel\_funcao\_criterio$. O *kernel* $kernel\_funcao\_criterio$ é executado para calcular a pontuação dos conjuntos $S \cup Z$. Após terminar a execução do *kernel*, determinamos o gene fraco x de Z e o gene forte x_c de C que, ao substituir x por x_c no conjunto Z, maximiza a pontuação do conjunto $S \cup Z$. Por fim, comparamos a pontuação do conjunto $S \cup Z \cup \{x_c\} - \{x\}$ com o conjunto $S \cup Z$ e se a pontuação for maior, efetuamos a substituição de x por x_c no conjunto Z.

Algoritmo 7: Descrição do algoritmo $Substituicao\_GPU$

Entrada: Conjunto de genes S, Z, C e a série temporal $\mathbf{T}$.
Saída: Um gene x considerado fraco do conjunto Z e um gene candidato x_c do conjunto C.
Resultado: Substitui o gene x por x_c somente se a pontuação obtida com o gene $x_c \in Z$ é maior que a pontuação do conjunto Z com o gene x.

produzir as séries temporais $T_{S \cup Z}$, trocando $x_i \in Z$ por $x_c \in C$ em $S \cup Z$;
transferir $T_{S \cup Z}$ para memória da GPU;
$n\_threads \leftarrow |S \cup Z|$;
$n\_blocos \leftarrow |Z| * |C|$;
alocar um vetor dev_score na memória global da GPU;
$kernel\_funcao\_criterio <<< n\_blocos,\ n\_threads >>> (dev\_score)$;
$cudaDeviceSynchronize()$;
transferir $dev\_score(S \cup Z)$ da GPU para CPU em $host\_score$;
determinar $x = \max_{x \in Z}\{host\_score(S \cup Z \cup \{x_c\} - \{x\})\}$;
determinar $x_c = \max_{x_c \in C}\{host\_score(S \cup Z \cup \{x_c\} - \{x\})\}$;
if $J(S \cup Z \cup \{x_c\} - \{x\}) > J(S \cup Z)$ **then**
    substituir x_c por x em C;
    substituir x por x_c em Z;
end

4.2.2 Função Critério na GPU

Nesta seção, descrevemos como resolvemos as restrições, descritas na Seção 2.4, na GPU. Em seguida, descrevemos o *kernel* utilizado para o cálculo da função critério do conjunto $S \cup Z$.

4.2.2.1 Restrição de Relação e Restrição Linear

A busca por soluções de um CSP envolve testar valores dentro do domínio das variáveis, de maneira tal que satisfaça todas as restrições do conjunto de restrições. Esse comportamento não é ideal para a GPU, uma vez que todas as *threads* devem seguir o mesmo fluxo de instrução durante sua execução. Para evitar os desvios de instruções, mantemos na memória de textura uma tabela de tamanho $n3^n$ com todas as 3^n possibilidades de solução e uma tabela com 3^n entradas para validar uma solução nas *threads*.

O primeiro conjunto de restrições limita o tipo de relação que o gene x_j pode ter em relação ao gene x_i. Essa restrição é chamada de restrição de relação, uma vez que o gene x_j só pode apresentar um tipo de relação com o gene x_i, ativando-o ou inibindo-o. Utilizamos o paralelismo dinâmico [2] para lançar um novo *kernel*, com o objetivo de verificar se a linha da solução satisfaz essa

[2] característica que permite às *threads* criarem novos *kernels*, chamando-os de *kernels* filho. Essa característica está disponível somente em GPUs com *compute capability* 3.5 ou superior.

restrição.

O segundo e terceiro conjuntos de restrições utilizam restrições lineares para definir uma relação linear entre as variáveis. A restrição linear é uma expressão matemática onde os termos lineares (coeficientes multiplicados pelas variáveis) são somados e o valor resultante é forçado a ser maior, maior ou igual, menor, menor ou igual ou exatamente igual a um valor que está do outro lado da expressão. Nesse contexto, os coeficientes são as expressões dos genes e as variáveis são os tipos de regulação que os genes x_j, para $j = 1, \ldots, n$, podem assumir em relação a um gene x_i. Lançamos um novo *kernel* para calcular esse somatório e invalidar as entradas das soluções que não obedecem à restrição.

4.2.2.2 Kernel da Função Critério

O *kernel* da função critério tem como parâmetro de entrada um vetor dev_score para armazenar as pontuações dos diferentes conjuntos $S \cup Z$. O Algoritmo 8 descreve a execução do cálculo da função critério na GPU. Os genes e os conjuntos são identificados pelas variáveis $index\_gene\_solution$ e $index\_solution$, respectivamente. Nesse *kernel*, em cada bloco possui uma *thread* principal identificada pela $index\_gene\_solution$, cujo valor é 0. Essa *thread* é responsável por iniciar variáveis e unir os resultados do conjunto $S \cup Z$, quando necessário. O *kernel* inicia executando o conjunto de restrições na GPU. Em seguida, executamos um novo *kernel*, chamado de $kernel\_prop\_matriz$, para determinar as m_i linhas consistentes do gene x_i e para calcular a proporção de matrizes consistentes do gene x_i ($prop\_matriz_i$), de acordo com a Equação (3.3). A chamada da função $cudaDeviceSynchronize()$ bloqueia o dispositivo até que todas as tarefas executadas anteriormente sejam completadas. Depois, calcula-se a entropia H_i do gene x_i utilizando a Equação (3.6). Adicionamos, na memória compartilhada, uma variável de controle $shr\_scoreIsZero$, para informar a todas as *threads* do bloco que um gene não apresentou linhas consistentes e, portanto, o conjunto $S \cup Z$ deve ser considerado ruim e sua pontuação deve ser 0. Se todos os genes apresentarem linhas consistentes, a proporção de matrizes consistentes e a entropia do conjunto $S \cup Z$ são determinados por meio da soma dos valores encontrados para cada gene $x_i \in S \cup Z$. Esses dois somatórios devem ser operações atômicas, uma vez que as variáveis $shr\_\mathcal{P}$ e $shr\_\mathcal{H}$ estão localizadas na memória compartilhada e não deve haver inconsistência na leitura e escrita dos valores. Por fim, calculamos a pontuação do conjunto $S \cup Z$ de acordo com a Equação 3.9.

Algoritmo 8: Descrição do *kernel* $kernel\_funcao\_criterio$

Entrada: Um vetor dev_score localizado na memória da GPU.
Saída: Pontuação do conjunto $S \cup Z$ no vetor dev_score.

```
1  index_solution ← blockIdx.x;
2  index_gene_solution ← threadIdx.x;
3  if index_gene_solution = 0 then
4      dev_score[index_solution] ← 0;
5      shr_scoreIsZero ← 0 ; /* Variável de controle para o conjunto
       S ∪ Z */
6  end
7  executar as restrições na GPU;
8  kernel_prop_matriz <<< n_blocos, n_threads >>> (m_i, prop_matriz_i);
9  cudaDeviceSynchronize();
10 entropia_i ← H_i;
11 if m_i = 0 then
12     atomicAdd(&shr_scoreIsZero, 1);
13 end
14 __syncthreads();
15 if shr_scoreIsZero = 0 then
16     atomicAdd(&shr_P, prop_matriz_i);
17     atomicAdd(&shr_H, entropia_i);
18     __syncthreads();
19     if index_gene_solution = 0 then
20         dev_score[index_solution] ← ω(shr_P) + (1 − ω)(shr_H);
21     end
22 end
```

4.3 Abordagem Híbrida

Com o objetivo de melhorar o desempenho da solução híbrida em relação à abordagem em GPU, estendemos o algoritmo e mesclamos a utilização da CPU e GPU para calcular a pontuação de $S \cup Z$. Nessa abordagem, introduzimos um parâmetro ϕ que, de acordo com a quantidade de genes na semente crescida, indica se o local de execução do cálculo da função critério deve ser alterado entre CPUs e GPUs. Em nossos testes, utilizamos $\phi = 8$, uma vez que, quando a semente cresce, o número de soluções possíveis crescem exponencialmente, não sendo suportados pelas GPUs.

4.3.1 Crescimento da Semente

O crescimento da semente nessa abordagem faz uso dos algoritmos descritos nas Seções 4.1 e 4.2. Os trechos de instruções referentes à comunicação entre processos e operações de conjuntos utilizados nos algoritmos SFS, SBS e o passo de substituição das abordagens de *cluster* de CPUs, utilizando GPU, foram removidos, uma vez que as instruções de comunicação são realizadas

antes que esses algoritmos sejam chamados. As operações de conjuntos são realizadas posteriormente, devido a que os algoritmos SFS, SBS e o passo de substituição irão retornar o melhor gene de um subconjunto para, depois, encontrar o melhor gene do conjunto todo. Esse esquema está representado na Figura 4.7 e é utilizado pelos três algoritmos.

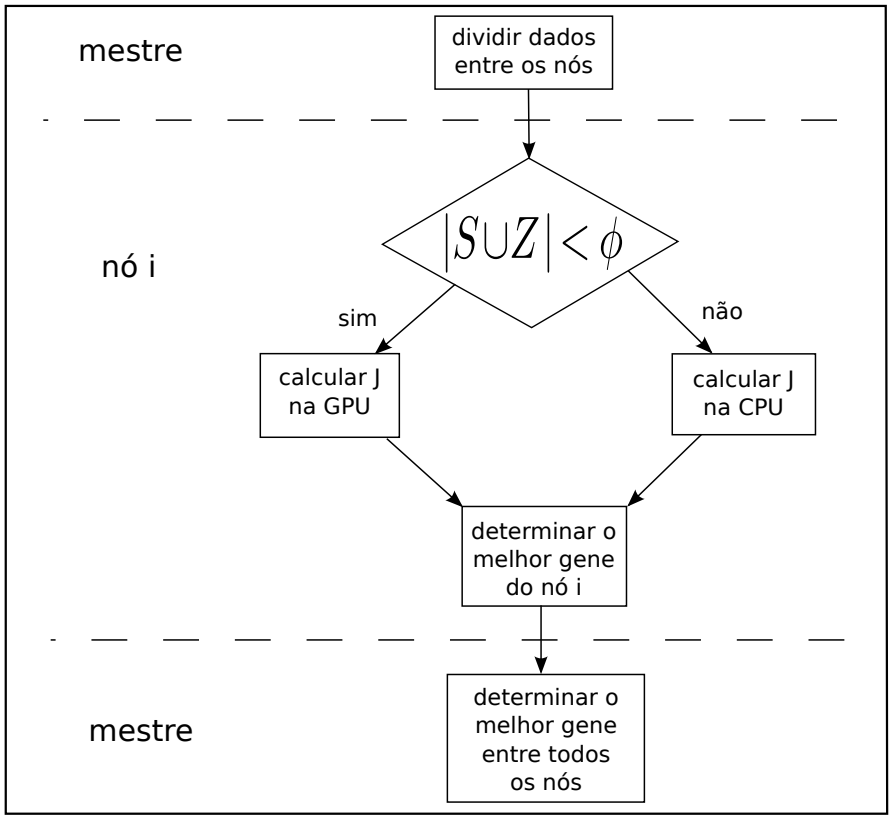


Figura 4.7: Fluxo de instruções dos algoritmos SFS, SBS e passo de substituição utilizando CPUs/GPUs.

Por exemplo, para a execução do SFS na abordagem híbrida, o nó mestre distribui todo o conjunto Z para todos os nós e divide o conjunto C entre os nós. Na sequência, os nós decidem onde as funções critério dos conjuntos $S \cup Z$ vão ser calculadas (CPU ou GPU) e qual é o melhor gene candidato referente ao subconjunto que lhe foi atribuído. Então, o nó mestre determina o melhor gene entre todos os nós quese refere ao melhor gene candidato x_c do conjunto C, remove x_c do conjunto C e adiciona x_c no conjunto Z.

4.3.1.1 Sequential Forward Selection - SFS

A seleção de um gene, na abordagem híbrida, descrito no Algoritmo 9, inicia distribuindo-se o conjunto Z para todos os nós. Logo depois, dividimos o conjunto de genes candidatos C entre os p nós, formando os subconjuntos $C_0, C_1, \ldots, C_{p-1}$, de forma tal que o nó p_i receba o subconjunto C_i. Em seguida, verificamos se o tamanho da semente com um gene candidato é menor que ϕ; se for menor, então calculamos a função critério de todos os genes de C_i

na GPU, por meio da chamada ao algoritmo $SFS\_GPU()$. Senão, calculamos a função critério para todos os genes do subconjunto C_i na CPU por meio do algoritmo $SFS\_CPUs()$. As variáveis $gene\_local$ e $score\_local$ armazenam, respectivamente, o melhor gene candidato x_c do subconjunto C_i e sua pontuação $J(S \cup Z \cup \{x_c\})$. Após o melhor gene candidato de cada subconjunto C_i ser encontrado, determinamos o melhor gene candidato x_c de todo o conjunto C. Por fim, o gene x_c é removido do conjunto C e adicionado ao conjunto Z.

Algoritmo 9: Descrição do algoritmo $SFS\_hibrido$

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene x_c do conjunto C que possui maior pontuação para o conjunto $S \cup Z \cup \{x_c\}$.
Resultado: Insere o melhor gene candidato no conjunto Z.

```
1  distribuir todo o conjunto Z para todos os nós;
2  distribuir C igualmente entre os p nós, de forma que o subconjunto C_i
   seja atribuído ao nó p_i;
3  if |S ∪ Z| + 1 < φ then
4      SFS_GPU(S, Z, C_i, gene_local, score_local);
5  else
6      SFS_CPUs(S, Z, C_i, gene_local, score_local);
7  end
8  encontrar o melhor gene candidato x_c entre todos C_i;
9  C = C \ {x_c};
10 Z = Z ∪ {x_c};
```

4.3.1.2 Sequential Backward Selection - SBS

A remoção de um gene na abordagem híbrida, descrita no Algoritmo 10, inicia distribuindo-se todo o conjunto Z e a sua pontuação para todos os nós. Logo depois, dividimos o conjunto Z entre os p nós, de forma que o subconjunto Z_i seja atribuído ao nó p_i. Em seguida, verificamos se o tamanho da semente crescida sem um gene é menor que ϕ; se for menor, então calculamos a função critério sem todos os genes do subconjunto Z_i na GPU, através da chamada ao algoritmo $SBS\_GPU()$. Senão, calculamos a função critério sem os genes do subconjunto Z_i na CPU por meio do algoritmo $SBS\_CPUs()$. As variáveis $gene\_local$ e $score\_local$ armazenam, respectivamente, o gene x, considerado o mais fraco, do subconjunto Z_i e sua pontuação $J(S \cup Z - \{x\})$. Após o gene mais fraco de todos os subconjuntos Z_i serem encontrados, determinamos o gene mais fraco x de todo o conjunto Z. Por fim, se a pontuação da semente crescida sem o gene x for maior que a pontuação da semente com o gene x, então, removemos o gene x do conjunto Z.

Algoritmo 10: Descrição do algoritmo $SBS\_hibrido$

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene fraco do conjunto Z.
Resultado: Remoção ou não do gene fraco do conjunto Z.

```
1  distribuir todo o conjunto Z e sua pontuação para todos os nós;
2  distribuir Z igualmente entre os p nós, de forma que o subconjunto Z_i
   seja atribuído ao nó p_i;
3  if |S ∪ Z| − 1 < φ then
4      SBS_GPU(S, Z_i, gene_local, score_local);
5  else
6      SBS_CPUs(S, Z_i, gene_local, score_local);
7  end
8  encontrar o gene mais fraco x entre todos Z_i;
9  if J(S ∪ Z − {x}) > J(S ∪ Z) then
10     Z = Z \ {x};
11     C = C ∪ {x};
12 end
```

Algoritmo 11: Descrição do algoritmo $Substituicao\_hibrido$

Entrada: Conjunto de genes S, Z, C e a série temporal **T**.
Saída: Um gene fraco x do conjunto Z e um gene candidato x_c de C.
Resultado: Substituir o gene x por x_c somente se a pontuação obtida com o gene $x_c \in Z$ é maior que a pontuação do conjunto Z com o gene x.

```
1  distribuir todo o conjunto Z e sua pontuação para todos os nós;
2  distribuir C igualmente entre os p nós, de forma que o subconjunto C_i
   seja atribuído ao nó p_i;
3  if |S ∪ Z| < φ then
4      Substituicao_GPU(S, Z, C_i, gene_fraco_local, gene_forte_local, score_local);
5  else
6      Substituicao_CPUs(S, Z, C_i, gene_fraco_local, gene_forte_local, score_local);
7  end
8  determinar gene fraco x = max_{x∈Z}{J(S ∪ Z ∪ {x_c} − {x})};
9  determinar gene forte x_c = max_{x_c∈C}{J(S ∪ Z ∪ {x_c} − {x})};
10 if J(S ∪ Z ∪ {x_c} − {x}) > J(S ∪ Z)) then
11     substituir x por x_c em Z;
12     substituir x_c por x em C;
13 end
```

4.3.1.3 Passo de Substituição

O passo de substituição, na abordagem híbrida, descrito no Algoritmo 11, inicia distribuindo-se todo o conjunto Z e a sua pontuação para todos os nós. Logo depois, dividimos o conjunto C entre os p nós, de forma que o subconjunto C_i seja atribuído ao nó p_i. Em seguida, verificamos se o tamanho da semente crescida é menor que ϕ, se for menor, então calculamos a função critério, substituindo todos os genes do conjunto Z pelos genes do subconjunto

C_i na GPU, mediante chamada ao algoritmo $Substituicao\_GPU()$. Senão, calculamos a função critério, substituindo todos os genes do conjunto Z pelos genes do subconjunto C_i na CPU por meio do algoritmo $Substituicao\_CPUs()$. As variáveis $gene\_fraco\_local$, $gene\_forte\_local$ e $score\_local$ armazenam, respectivamente, um gene fraco x do conjunto Z, o melhor gene candidato x_c do subconjunto C_i e a pontuação $J(S \cup Z)$, substituindo x por x_c, onde $x_c \in C_i$. Após o gene melhor gene candidato de todos os subconjuntos C_i serem encontrados, determinamos o melhor gene candidato x_c de todo o conjunto C que melhor substitui o gene x do conjunto Z. Por fim, se a pontuação da semente crescida substituindo o gene x pelo gene x_c é maior que a pontuação da semente com o gene x, então substituímos o gene x pelo gene x_c no conjunto Z e substituímos o gene x_c pelo gene x no conjunto C.

CAPÍTULO

5

Resultados Experimentais

5.1 Ambiente de Testes

A base de dados utilizada para teste foram a das células *HeLa* (Whitfield *et al.*, 2002) e *In Silico* (Prill *et al.*, 2010) da competição DREAM4 (2009) (DREAM, 2009). A primeira base de dados contém 2 redes: a primeira com 20 genes e 12 amostras de tempo e, a segunda, com 20 genes e 14 amostras de tempo. A segunda base de dados possui 100 genes e 21 amostras de tempo. Os dados de expressão gênica foram discretizados utilizando o algoritmo BiKmeans. Configuramos a semente inicial com 3 genes e adicionamos até 8 genes com o algoritmo de seleção de características IFFS para a realização dos testes. Executamos 3 vezes cada um dos experimentos com as 3 redes para cada implementação e calculamos a média aritmética desses tempos de execução para medir o desempenho das implementações realizadas.

A implementação, baseada em *cluster* de CPUs, foi realizada em um *cluster* com 38 nós com processadores Intel(R) Xeon(R) CPU X3440@2.53GHz de 4 núcleos, 4 GB de memória RAM e sistema operacional Rocks 6.1.1. Os testes com a abordagem, utilizando uma GPU e com *cluster* de CPUs/GPUs, foram realizados na *Amazon EC2* (*Amazon Elastic Compute Cloud*) que é um serviço da *Amazon* para computação em nuvem. Convém esclarecer que a *Amazon EC2* fornece instâncias (máquinas virtuais) com amplas configurações para propósitos específicos. A instância utilizada foi o p2.8xlarge com processador Intel Xeon E5-2686 v4 (32 vCPUs), 488 GB de memória RAM e 8 GPUs Nvidia K80.

5.2 Desempenho das Abordagens

O passo de crescimento da semente e o de inferência foram implementados separadamente. Nesta seção, apresentamos a comparação do desempenho sequencial com as implementações paralelas do passo de crescimento da semente e passo de inferência de forma separada.

5.2.1 Passo de Crescimento da Semente

5.2.1.1 Abordagem com Cluster de CPUs

Esta seção descreve os experimentos realizados com o algoritmo paralelo, com abordagem em *cluster* de CPUs. Denotamos p como sendo o número de processadores. Primeiramente, executamos os testes com a rede 1 das células *HeLa*, utilizando o algoritmo sequencial com 17 experimentos ou sementes diferentes. Esses experimentos correspondem aos experimentos biológicos realizados por Higa *et al.* (2013). Em seguida, executamos a mesma rede com o algoritmo paralelo implementando, utilizando *cluster* de CPUs.

HeLa rede 1 Semente #	Número de Processadores				
	1	2	4	8	16
1	70,5	42,0	23,4	15,8	12,2
2	97,0	57,0	31,1	17,5	15,9
3	25,4	13,5	9,3	5,3	4,6
4	38,5	19,5	15,4	7,5	7,2
5	33,8	17,1	13,4	7,3	6,7
6	0,1	0,1	0,1	0,1	0,1
7	20,2	11,3	9,6	7,5	6,7
8	49,5	27,2	18,5	8,5	7,6
9	0,2	0,1	0,1	0,1	0,1
10	0,2	0,1	0,1	0,1	0,1
11	46,4	23,8	15,8	8,4	7,9
12	21,7	12,0	8,3	4,3	4,2
13	20,7	10,8	6,4	3,9	3,4
14	53,9	29,3	16,6	9,4	7,4
15	0,2	0,1	0,1	0,1	0,1
16	28,4	16,0	8,9	4,9	4,5
17	21,3	11,3	6,3	3,6	3,2

Tabela 5.1: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo paralelo com abordagem em *cluster* de CPUs com 2, 4, 8 e 16 processadores para a rede 1 das células *HeLa* com 17 experimentos.

A Tabela 5.1 mostra o resultado das execuções, comparando-se o algoritmo sequencial (para 1 processador) com a implementação paralela em *cluster* de CPUs com 2, 4, 8 e 16 processadores. A Figura 5.1 mostra os *speedups* alcançados em relação ao algoritmo sequencial. Em nossa implementação,

conseguimos *speedups* de, aproximadamente, 2 quando $p = 2$. Esse valor se manteve crescente, todavia, não houve ganhos significativos de *speedup* ao aumentar o número de processadores de 8 para 16, pois, devido ao tamanho da rede, a quantidade de computação de cada nó é pequena, limitando, assim, o ganho. Nos casos em que os tempos de execução apresentaram baixos valores (semente #6, #9, #10 e #15), o algoritmo IFFS não conseguiu adicionar nenhum gene na semente e o algoritmo sequencial identifica isso rapidamente através do Gecode. Nesses casos, as soluções paralelas não obtiveram ganhos significativos.

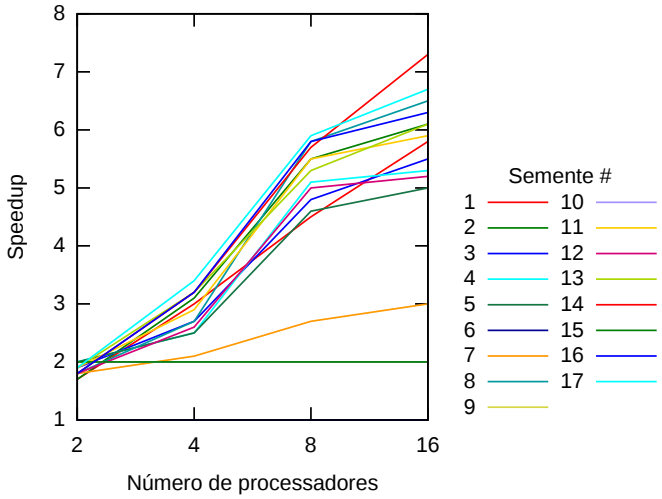


Figura 5.1: *Speedups* do algoritmo paralelo em *cluster* de CPUs da rede 1 das células *HeLa*.

Em seguida, executamos nosso algoritmo para a rede 2 das células *HeLa* com 10 experimentos aleatórios. A Tabela 5.2 mostra o tempo de execução, considerando 2, 4, 8 e 16 processadores. A Figura 5.2 mostra os *speedups* alcançados em relação ao algoritmo sequencial.

HeLa rede 2 Semente #	Número de Processadores				
	1	2	4	8	16
1	110,7	65,7	35,7	19,6	18
2	8,9	5,2	2,8	1,7	1,5
3	119,9	70,9	40	22,9	20,1
4	130,8	75	42,9	22,6	22,7
5	27,9	15,7	10,3	5,1	4,9
6	27,9	15,9	10,4	5,1	4,8
7	32,4	18,8	10,8	6,2	5
8	28,2	15,9	10,5	5,2	4,9
9	28,1	16,1	10,4	5,2	4,9
10	62,6	34,1	20,8	10,3	10

Tabela 5.2: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo paralelo com abordagem em *cluster* CPUs com 2, 4, 8 e 16 processadores para a rede 2 das células *HeLa* com 10 experimentos

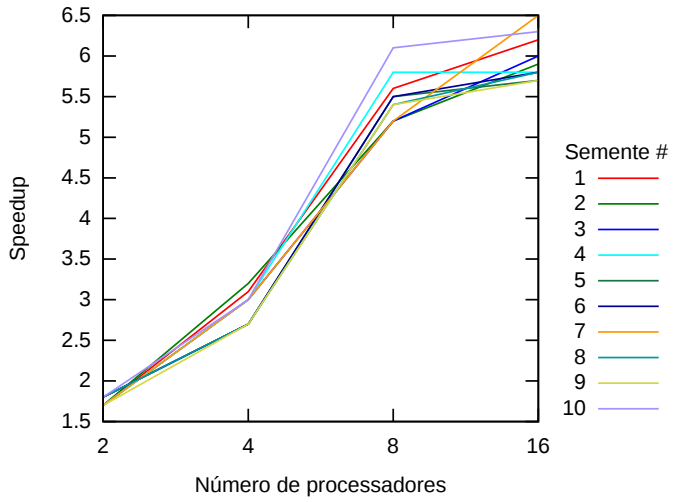


Figura 5.2: *Speedups* do algoritmo paralelo em *cluster* de CPUs da rede 2 das células *HeLa*.

É possível perceber que, nos experimentos realizados, conseguimos *speedup* de, aproximadamente, 2 quando o passo de crescimento da semente é executado com 2 processadores. Dobrando o número de processadores para 4, os *speedups* variaram entre 2,7 e 3,2. Quando testamos para $p = 8$, os *speedups* dobraram em relação a 4 processadores e alcançaram valores entre 5,2 e 6,1. Aumentando o número de processadores para 16, não conseguimos dobrar o *speedup* em relação a 8 processadores, os *speedups* para $p = 16$ variaram entre 5,7 a 6,5, em razão da pouca quantidade de cálculos em cada nó.

In Silico Semente #	Número de Processadores				
	1	2	4	8	16
1	23,3	13,7	7,6	5,1	4,5
2	128,2	82,4	45,2	24,6	20,7
3	70,8	37,8	25,6	14,5	12,2
4	127,9	83,9	47	24,5	21,5
5	71,1	37,3	25,4	14,6	12,3
6	133,4	84,7	46,5	25,6	23,5
7	178	102,8	56,3	30,8	26
8	196,5	106,4	69,6	37,2	32,9
9	130,2	74,7	40,7	22,8	20,7
10	129,5	77,0	41,5	23,6	20,5

Tabela 5.3: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo paralelo com abordagem em *cluster* CPUs com 2, 4, 8 e 16 processadores para a rede *In Silico* com 10 experimentos.

A última base de dados testada foi a *In Silico* com 10 sementes aleatórias. A Tabela 5.3 mostra o tempo de execução da implementação em *cluster* de CPU com 2, 4, 8 e 16 processadores. O gráfico da Figura 5.3 mostra os *speedups* adquiridos em relação ao algoritmo sequencial. Comparando-se com o desempenho sequencial, quando dobramos o número de processadores, o

speedup chegou próximo de 2 nas sementes #3 e #5. Quando aumentamos o número de processadores para 4, os *speedups* variaram entre 2,7 e 3,2. Ao testar para 8 processadores, observamos, em alguns casos, que o tempo de execução diminuiu, aproximadamente, pela metade, em relação ao tempo de execução com 4 processadores. O mesmo não ocorreu quando aumentamos o número de processadores para 16, apesar de o tempo de execução de $p = 16$ ser menor em relação aos tempos de execução de $p = 8$.

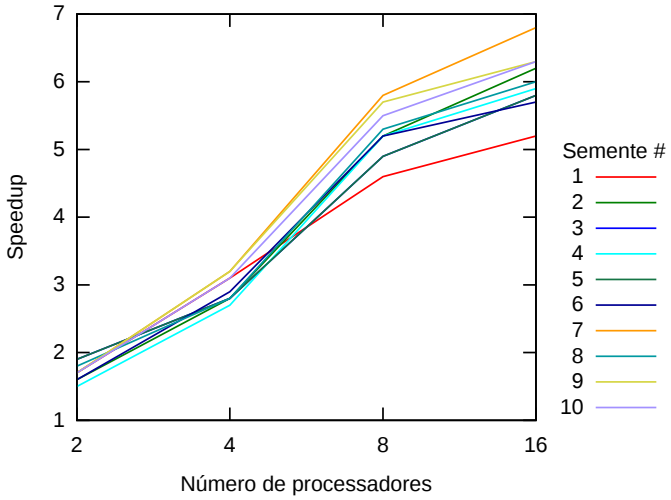


Figura 5.3: *Speedups* do algoritmo paralelo em *cluster* de CPUs da rede *In Silico*.

Observando os *speedups* nas 3 bases de dados executando o algoritmo paralelo em *cluster* de CPUs, notamos que há uma escalabilidade entre os valores de *speedup*. Quando aumentamos o número de processadores para 16, esperávamos alcançar *speedups* de, aproximadamente, 2 quando comparado para 8 processadores. Entretanto, não observamos isso. Nesse caso, notamos que alguns processadores ficaram ociosos devido ao tamanho da rede e, por isso, não apresentaram o ganho esperado ao dobrar o número de processadores.

5.2.1.2 Abordagem com Ambiente Híbrido

Nesta seção, descrevemos os experimentos realizados com o ambiente híbrido, no qual o passo de crescimento da semente utiliza as CPUs/GPUs para calcular a função critério dos genes presentes em $S \cup Z$.

Foram realizados, também, testes considerando apenas uma GPU. Entretanto, os *speedups* encontrados não foram bons, tendo tempos de execução de 2 a 50 vezes superior em relação ao algoritmo sequencial. Observamos que o tempo de execução dos testes para uma GPU aumentava significativamente quando o tamanho da semente crescida era superior a 7. Então, introduzimos o parâmetro ϕ para verificar a quantidade de genes na semente crescida e calcular a função critério pela CPU utilizando o Gecode.

Na abordagem híbrida, o número de CPUs é igual ao número de GPUs. Primeiro, testou-se a rede 1 das células *HeLa* com o algoritmo híbrido para 1, 2, 4 e 8 CPUs/GPUs. Os tempos de execução do algoritmo sequencial (identificado pela coluna CPU) com a abordagem híbrida são mostrados na Tabela 5.4.

HeLa rede 1 Semente #	CPU	CPUs/GPUs			
		1	2	4	8
1	70,5	52,6	31,8	22,1	15,7
2	97	63,6	42	28,6	17,1
3	25,4	17,1	10,6	6,1	4,5
4	38,5	25,8	15,8	9	6,6
5	33,8	23,9	13,6	8,5	5,4
6	0,1	0,6	0,6	0,6	0,8
7	20,2	13,6	7,2	6	5,3
8	49,5	32	16,5	10,5	4,5
9	0,2	0,5	0,6	0,7	0,9
10	0,2	0,6	0,6	0,7	0,9
11	46,4	28,6	15,7	8,2	5,7
12	21,7	14,4	7,8	5,3	4,1
13	20,7	14,2	8,5	4,9	4
14	53,9	37,5	19,1	12,2	8,3
15	0,2	0,9	0,8	0,8	1
16	28,4	18,1	10,6	7,2	5,8
17	21,3	15,1	10	6,3	5,2

Tabela 5.4: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo híbrido com 1, 2, 4 e 8 CPUs/GPUs para a rede 1 das células *HeLa* com 17 experimentos.

É possível observar que houve *speedup* de, aproximadamente, 2 quando utilizamos apenas 1 CPU/GPU. Ao aumentar o número de CPUs/GPUs para 2, o tempo de execução diminuiu quase pela metade em relação a 1 CPU/GPU na semente #8. O gráfico da Figura 5.4 mostra os *speedups* alcançados para 1, 2, 4 e 8 CPUs/GPUs. Para essa rede, o maior *speedup* alcançado foi 11, na semente #8, quando executamos com 8 CPUs/GPUs.

A Tabela 5.5 mostra os tempos de execução do algoritmo sequencial e híbrido com 1, 2, 4 e 8 CPUs/GPUs para rede 2 das células *HeLa*. A Figura 5.5 mostra o gráfico dos *speedups* alcançados pela Tabela 5.5. Observa-se que, em casos onde há pouco processamento de dados, o algoritmo sequencial é mais rápido que o híbrido com 1 CPUs/GPUs.

Por fim, comparamos o algoritmo sequencial com o híbrido para a rede *In Silico*. Os tempos de execução obtidos são mostrados na Tabela 5.6. Observa-se que com 1 CPU/GPU, os tempos de execução não alcançaram o tempo sequencial. Pela Figura 5.6, nota-se que, a partir de 2 CPUs/GPUs, o tempo de execução do algoritmo híbrido começa a superar o tempo de execução do algoritmo sequencial.

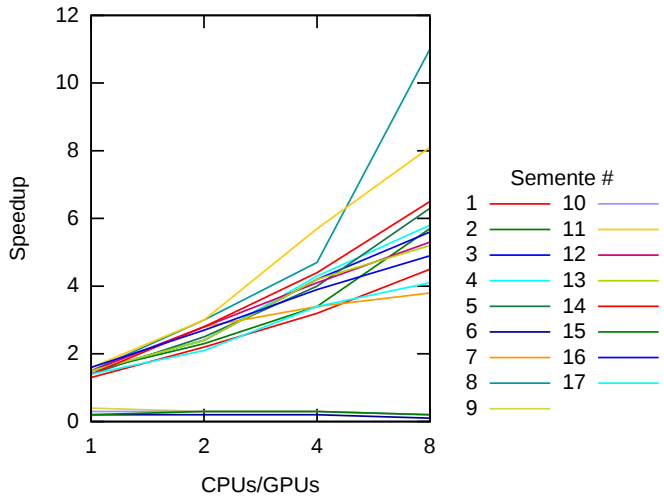


Figura 5.4: *Speedups* do algoritmo paralelo híbrido da rede 1 das células *HeLa*.

HeLa rede 2 Semente #	CPU	CPUs/GPUs			
		1	2	4	8
1	110,7	78,9	44	30,1	15,4
2	8,9	12,1	6,1	3,7	2,9
3	119,9	84,9	46,7	32,3	16,4
4	130,8	96,5	52,5	35,3	18,6
5	27,9	24,3	13,4	8,3	5,8
6	27,9	24,3	13,5	8,4	5,8
7	32,4	37,9	17,5	9,5	6,8
8	28,2	24,6	13,8	8,5	5,8
9	28,1	24,5	13,6	8,5	5,8
10	62,6	62,1	30,7	15,6	10,5

Tabela 5.5: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo híbrido com 1, 2, 4 e 8 CPUs/GPUs para a rede 2 das células *HeLa* com 10 experimentos.

Quando executamos os testes para 1 CPU/GPU, o tempo de execução do algoritmo híbrido não foi inferior ao tempo do algoritmo sequencial. Apesar disso, o comportamento do tempo de execução do algoritmo híbrido é decrescente de acordo com o número de CPUs/GPUs. Esse comportamento mostra que o algoritmo híbrido é escalável com o número de CPUs/GPUs.

Vale ressaltar que, no algoritmo sequencial, se um gene $x_i \in S \cup Z$ não apresentar linhas consistentes, o cálculo da função critério $J(S \cup Z)$ é parado imediatamente, não precisando verificar se o gene $x_{i+1} \in S \cup Z$ possui ou não linhas consistentes. Essa característica não é incorporada ao algoritmo para GPU, uma vez que todas as *threads* devem executar a mesma tarefa. De modo geral, se existissem bibliotecas para a resolução de restrições em GPU, os *speedups* obtidos seriam maiores tanto para uma GPU quanto para o algoritmo híbrido.

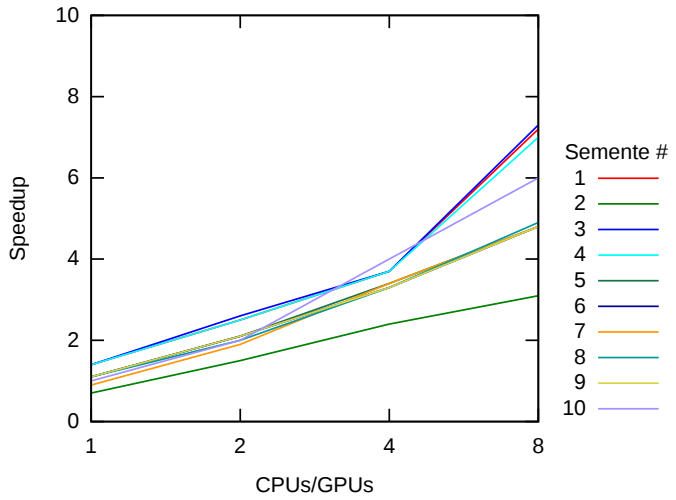


Figura 5.5: *Speedups* do algoritmo paralelo híbrido da rede 2 das células *HeLa*.

In Silico Semente #	CPU	CPUs/GPUs			
		1	2	4	8
1	23,3	128,3	127,1	63,8	11,9
2	128,2	218,1	88,7	40,2	23,4
3	70,8	563,3	219,2	96,6	41
4	127,9	192,8	86,9	43,7	22,4
5	71,1	567,3	218,9	88,8	40,1
6	133,4	285,4	118,3	64,4	26,8
7	178	305,4	121,5	50,8	22,1
8	196,5	435,3	181	83,3	38,7
9	130,2	356,1	152,4	67,6	31,1
10	129,5	357,1	145,5	62,9	31,1

Tabela 5.6: Tabela comparativa do desempenho, em segundos, do algoritmo sequencial com o algoritmo híbrido com 1, 2, 4 e 8 CPUs/GPUs para a rede *In Silico* com 10 experimentos.

5.2.2 Passo de Inferência

Esta seção descreve os experimentos realizados com o algoritmo de inferência paralelo em *cluster* de CPUs. O passo de inferência é uma etapa de amostragem, na qual são geradas $\mathcal{M}$ redes e apenas 10% das $\mathcal{M}$ redes montadas que são interessantes (com baixa entropia) representam a saída desse passo.

Durante a realização deste experimento, vimos que, para diferentes sementes, os tempos de execução eram semelhantes se o valor de $\mathcal{M}$ fosse igual para todos. A fim de avaliar o desempenho do passo de inferência, em vez de testar com diferentes sementes, testamos para diferentes quantidades de redes a serem montadas. A quantidade de redes montadas foram $\mathcal{M} = 1000, 2000, 3000, 4000$ e 5000.

A Tabela 5.7 mostra os resultados do tempo de execução do algoritmo sequencial e algoritmo paralelo em *cluster* de CPUs. Pela Figura 5.7a, é possível

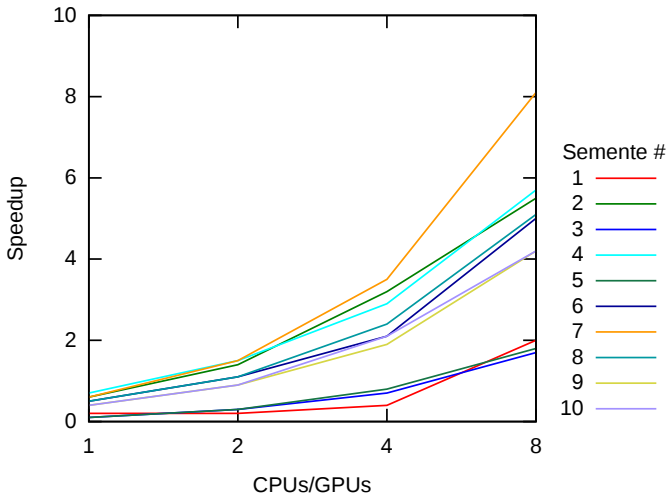
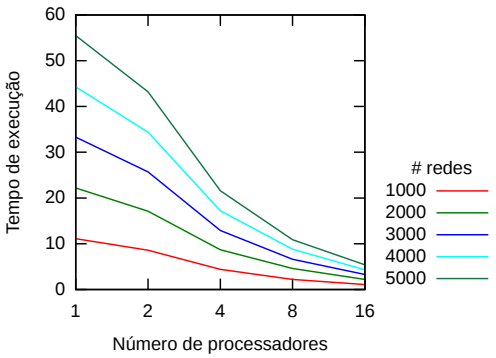


Figura 5.6: *Speedups* do algoritmo paralelo híbrido da rede *In Silico*.

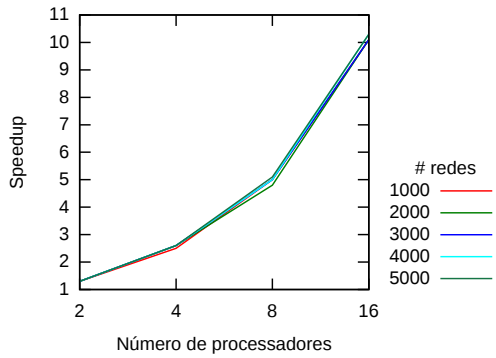
observar que o tempo de execução do algoritmo sequencial é proporcional à quantidade de redes montadas. Percebemos, também, que esse comportamento está presente no tempo de execução do algoritmo paralelo para 2, 4, 8 e 16 CPUs, realizando o passo de amostragem para 1000, 2000, 3000, 4000 e 5000 redes.

# redes montadas	Número de Processadores				
	1	2	4	8	16
1000	11,1	8,6	4,4	2,2	1,1
2000	22,2	17,1	8,7	4,6	2,2
3000	33,3	25,7	12,9	6,6	3,3
4000	44,3	34,4	17,2	8,8	4,3
5000	55,5	43,2	21,6	10,9	5,4

Tabela 5.7: Tabela comparativa do desempenho, em segundos, do algoritmo de inferência sequencial com o algoritmo paralelo com abordagem em *cluster* CPUs com 2, 4, 8 e 16 processadores.



(a) Tempo de execução.



(b) *Speedups* obtidos.

Figura 5.7: Comparação entre o desempenho do algoritmo de inferência sequencial e o algoritmo de inferência paralelo.

Conforme a Figura 5.7b, os *speedups* são bem semelhantes para diferen-

tes quantidades de redes montadas. Era esperado que para $p = 2$, o *speedup* fosse mais próximo de 1 do que de 2, e isso de fato ocorreu. O motivo para esse comportamento com $p = 2$ é que, além de cada processador montar $\mathcal{M}/p$ redes, era necessário realizar uma ordenação local das redes geradas, transferir esses dados para o processador mestre e, então, encontrar os 10% de redes que possuem baixa entropia.

A Figura 5.7b mostra que alcançamos *speedups* superiores a 10 com 16 processadores. Outro ponto observado na Figura 5.7b é que o algoritmo é escalonável com a quantidade de CPUs utilizada.

CAPÍTULO

6

Conclusões

Neste trabalho, estudou-se o problema de inferir redes de regulação gênica. O objetivo deste trabalho foi propor uma primeira solução paralela para o algoritmo criado por Higa *et al.* (2013). Esse algoritmo é baseado no paradigma de crescimento da semente e é executado em dois passos: crescimento da semente e inferência. Para o passo de crescimento da semente foram propostas três soluções paralelas: a primeira é uma abordagem que utiliza *cluster* de CPUs, a segunda que utiliza uma única GPU e a terceira solução é uma abordagem híbrida, que mescla o uso da CPU e da GPU. Para o passo de inferência, propôs-se uma solução que utiliza *cluster* de CPUs.

Inicialmente, estudou-se as arquiteturas paralelas disponíveis, os conceitos biológicos envolvidos no problema de inferência, os modelos de rede de regulação gênica que existem, o conjunto de restrições utilizado para reduzir o espaço de busca das soluções de um gene e o problema de seleção de características. Em seguida, estudou-se o funcionamento dos dois passos do algoritmo de inferência.

No passo de crescimento da semente, desenvolveram-se três soluções paralelas. A primeira solução utiliza *cluster* de CPUs, no qual o cálculo da função critério de diferentes sementes é realizado pelos nós do *cluster*. A segunda solução utiliza uma GPU e, nessa abordagem, um gene é atribuído a uma *thread* e o conjunto $S \cup Z$ é atribuído a um bloco de *thread*. Então, o cálculo da função critério desse conjunto é realizado por todas as *threads* do bloco. A terceira solução mescla o uso de CPUs e GPUs para calcular a função critério das diferentes sementes. Para o passo de inferência, desenvolveu-se uma solução paralela, utilizando *cluster* de CPUs. Essa solução divide a tarefa de montar e calcular a entropia de $\mathcal{M}$ redes entre os p nós do *cluster*.

Conforme os resultados apresentados, utilizando *cluster* de CPUs, alcançamos *speedup* de 7,3 e, na abordagem híbrida, alcançamos *speedup* de 11, no passo de crescimento da semente. Por fim, no passo de inferência, alcançamos *speedups* superiores a 10, utilizando *cluster* de CPUs. Além disso, os algoritmos acima mostraram-se escaláveis com o número de dispositivos disponíveis.

Conforme relatado na Seção 5.2.1.2, a solução paralela com uma GPU não apresentou resultados satisfatórios em comparação à solução sequencial e em relação as outras soluções paralelas propostas para o crescimento da semente. O grande obstáculo para essa abordagem foi não haver um *solver* em CUDA para resolver as restrições. Pensando em melhorar a solução proposta para uma única GPU, uma sugestão de trabalho futuro é, assim que surgir um *solver* em CUDA, reimplementar o *kernel* $kernel\_funcao\_criterio$ utilizando as funções desse *solver* para resolver as restrições.

Além disso, outras técnicas podem ser exploradas para melhorar os *speedups* obtidos, como, realizar o balanceamento de carga em arquiteturas híbridas e explorar melhor o uso/limitações dos diferentes tipos de memória e utilizar as instruções de *warps* (funções *warp shuffle*) para a abordagem em GPU.

Referências Bibliográficas

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., e Walter, P. (2008). *Molecular biology of the cell*. Garland Publishing, 5th edition. Citado na página 12.

Andrade, T. P. d. (2012). *Interações gênicas usando redes booleanas limiarizadas modeladas como um problema de satisfação de restrições*. Dissertação de mestrado, Universidade de São Paulo. Citado na página 17.

Bargen, B. e Donnelly, P. (1998). *Inside DirectX: In-depth Techniques for Developing High-performance Multimedia Applications*. Microsoft Press, Redmond, WA, USA. Citado na página 7.

Crick, F. (1970). Central Dogma of Molecular Biology. *Nature*, 227(5258):561–563. Citado na página 12.

CUDA NVIDIA (2007). *Compute unified device architecture programming guide*, 1st edition. Citado nas páginas 8 e 10.

Devijver, P. A. e Kittler, J. (1982). *Pattern Recognition: A Statistical Approach*. Prentice Hall. Citado na página 24.

DREAM (2009). Dream: Dialogue for reverse engineering assessments and methods, 2009. http://dreamchallenges.org/project-list/dream4-2009/. [Online; accessed 11-January-2017]. Citado na página 55.

Educação, M. (2017). Dogma Central da Biologia Molecular - Mundo Educação. http://mundoeducacao.bol.uol.com.br/biologia/dogma-central-biologia-molecular.htm. [Online; accessed 17-February-2017]. Citado na página 12.

Friedman, N., Linial, M., Nachman, I., e Pe'er, D. (2000). Using bayesian networks to analyze expression data. *Journal of computational biology*, 7(3-4):601–620. Citado nas páginas 1 e 15.

Goodwin, B. C. (1963). *Temporal Organization in Cells: A Dynamic Theory of Cellular Control Processes.* Academic Press. Citado nas páginas 1 e 15.

Gropp, W., Lusk, E., Doss, N., e Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828. Citado na página 6.

Higa, C. H., Andrade, T. P., e Hashimoto, R. F. (2013). Growing seed genes from time series data and thresholded boolean networks with perturbation. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 10(1):37–49. Citado nas páginas 1, 2, 17, 23, 27, 29, 56, e 65.

Higa, C. H., Louzada, V. H., Andrade, T. P., e Hashimoto, R. F. (2011). Constraint-based analysis of gene interactions using restricted boolean networks and time-series data. In *BMC proceedings*, volume 5 (Suppl 2), page S5. BioMed Central Ltd. Citado nas páginas 18, 19, 20, e 23.

Higa, C. H. A. (2011). *Inferência de redes de regulação gênica utilizando o paradigma de crescimento de sementes*. Tese de doutorado, Universidade de São Paulo. Citado nas páginas 16, 18, e 27.

Hogeweg, P. e Hesper, B. (1978). Interactive instruction on population interactions. *Computers in biology and medicine*, 8(4):319–327. Citado na página 1.

Ideker, T., Galitski, T., e Hood, L. (2001). A new approach to decoding life: systems biology. *Annual review of genomics and human genetics*, 2(1):343–372. Citado na página 1.

Kauffman, S. A. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of theoretical biology*, 22(3):437–467. Citado nas páginas 1, 15, e 33.

Lau, K.-Y., Ganguli, S., e Tang, C. (2007). Function constrains network architecture and dynamics: A case study on the yeast cell cycle boolean network. *Physical Review E*, 75(5):051907. Citado na página 18.

Li, F., Long, T., Lu, Y., Ouyang, Q., e Tang, C. (2004). The yeast cell-cycle network is robustly designed. *Proceedings of the National Academy of Sciences of the United States of America*, 101(14):4781–4786. Citado nas páginas 16 e 17.

Li, Y., Liu, L., Bai, X., Cai, H., Ji, W., Guo, D., e Zhu, Y. (2010). Comparative study of discretization methods of microarray data for inferring transcriptional regulatory networks. *BMC bioinformatics*, 11(1):520. Citado na página 14.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif. University of California Press. Citado na página 14.

Marill, T. e Green, D. (1963). On the effectiveness of receptors in recognition systems. *IEEE transactions on Information Theory*, 9(1):11–17. Citado na página 24.

Nakariyakul, S. e Casasent, D. P. (2009). An improvement on floating search algorithms for feature subset selection. *Pattern Recognition*, 42(9):1932–1940. Citado nas páginas 24, 25, e 28.

Nussbaum, R. L., McInnes, R. R., e Thompson, H. F. (2008). *Thompson & Thompson Genética Médica*. Elsevier, 7th edition. Citado na página 11.

Organick, E. I. (1973). *Computer System Organization: The B5700/B6700 Series (ACM Monograph Series)*. Academic Press, Inc., Orlando, FL, USA. Citado na página 6.

Pfister, G. F. (1998). *In Search of Clusters*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition. Citado na página 6.

Prill, R. J., Marbach, D., Saez-Rodriguez, J., Sorger, P. K., Alexopoulos, L. G., Xue, X., Clarke, N. D., Altan-Bonnet, G., e Stolovitzky, G. (2010). Towards a rigorous assessment of systems biology models: the DREAM3 challenges. *PloS one*, 5(2):e9202. Citado na página 55.

Pudil, P., Novovičová, J., e Kittler, J. (1994). Floating search methods in feature selection. *Pattern recognition letters*, 15(11):1119–1125. Citado na página 24.

Rossi, F., Van Beek, P., e Walsh, T. (2006). *Handbook of constraint programming*. Elsevier. Citado na página 18.

Sanders, J. e Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional. Citado nas páginas 7 e 8.

Schena, M., Shalon, D., Davis, R. W., e Brown, P. O. (1995). Quantitative monitoring of gene expression patterns with a complementary dna microarray. *Science*, 270(5235):467. Citado na página 13.

Shalon, D., Smith, S. J., e Brown, P. O. (1996). A dna microarray system for analyzing complex dna samples using two-color fluorescent probe hybridization. *Genome research*, 6(7):639–645. Citado na página 13.

Shmulevich, I., Dougherty, E. R., Kim, S., e Zhang, W. (2002a). Probabilistic boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274. Citado nas páginas 1 e 15.

Shmulevich, I., Dougherty, E. R., e Zhang, W. (2002b). Gene perturbation and intervention in probabilistic boolean networks. *Bioinformatics*, 18(10):1319–1331. Citado na página 17.

Sima, C., Hua, J., e Jung, S. (2009). Inference of gene regulatory networks using time-series data: a survey. *Current genomics*, 10(6):416–429. Citado na página 14.

Somol, P., Novovicová, J., e Pudil, P. (2010). *Efficient feature subset selection and subset size optimization*. INTECH Open Access Publisher. Citado na página 25.

Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., e Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press. Citado na página 6.

Só Biologia (2008). DNA e RNA. http://www.sobiologia.com.br/conteudos/quimica_vida/quimica15.php. [Online; accessed 17-February-2017]. Citado nas páginas 11 e 12.

Tanenbaum, A. S. (1984). *Structured computer organization*. Prentice Hall PTR, 5th edition. Citado na página 6.

Velculescu, V. E., Zhang, L., Vogelstein, B., e Kinzler, K. W. (1995). Serial analysis of gene expression. *Science*, 270(5235):484. Citado na página 13.

Wang, Z., Gerstein, M., e Snyder, M. (2009). RNA-seq: a revolutionary tool for transcriptomics. *Nature reviews genetics*, 10(1):57–63. Citado na página 13.

Whitfield, M. L., Sherlock, G., Saldanha, A. J., Murray, J. I., Ball, C. A., Alexander, K. E., Matese, J. C., Perou, C. M., Hurt, M. M., Brown, P. O., e Botstein, D. (2002). Identification of genes periodically expressed in the human cell cycle and their expression in tumors. *Molecular biology of the cell*, 13(6):1977–2000. Citado na página 55.

Whitney, A. W. (1971). A direct method of nonparametric measurement selection. *Computers, IEEE Transactions on*, 100(9):1100–1103. Citado na página 24.