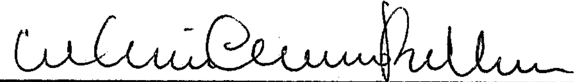
Sobre a Simulação Paralela de Alguns Modelos Complexos

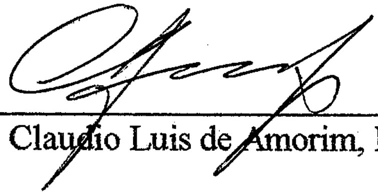
Raquel Coelho Gomes Pinto

Tese submetida ao corpo docente da Coordenação dos Programas de Pós-graduação em Engenharia da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências em Engenharia de Sistema e Computação.

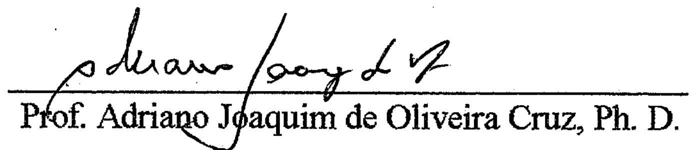
Aprovada por:



Prof. Valmir Carneiro Barbosa, Ph. D.
(Presidente)



Prof. Claudio Luis de Amorim, Ph. D.



Prof. Adriano Joaquim de Oliveira Cruz, Ph. D.

Rio de Janeiro, RJ - Brasil
Abril de 1994

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M. Sc.).

Sobre a Simulação Paralela de Alguns Modelos Complexos

Raquel Coelho Gomes Pinto
Abril, 1994

Orientador: Valmir Carneiro Barbosa
Programa: Engenharia de Sistemas e Computação

Um sistema complexo é, em geral, uma grande coleção de entidades distintas, que compartilham um conjunto mútuo de conexões dinâmicas. Um sistema complexo dinâmico evolui de acordo com um conjunto estatístico ou determinístico de regras que relacionam o estado do sistema em um momento anterior a um posterior. Temos como exemplo a atmosfera terrestre, na qual suas entidades básicas podem ser consideradas como a densidade do ar, pressão, velocidade, etc., e suas conexões podem ser definidas pelas equações básicas de movimento. O cérebro humano também pode ser considerado um sistema complexo, onde as entidades são os neurônios e as conexões são compostas pelas sinapses que interligam os neurônios. A principal característica destes sistemas é o seu comportamento complexo como um todo. Cada entidade possui um comportamento local simples, mas a interação entre elas gerada pela forma como são conectadas, torna o comportamento coletivo do sistema imprevisível. Portanto, é necessário simular o sistema para verificarmos seu comportamento perante diferentes entradas e alterações na sua topologia. Baseado nisto, desenvolvemos simuladores distribuídos de alguns sistemas complexos cujas estruturas e restrições são semelhantes. Implementamos a simulação dos seguintes modelos: rede neuronal binária de Hopfield, rede neuronal contínua de Hopfield, máquina de Boltzmann, rede Bayesiana e uma rede neuronal para resolução de sistemas lineares. A existências destes modelos tem como motivação principal a resolução de problemas de otimização combinatória de difícil solução e problemas de inteligência artificial. Com o propósito de testar e avaliar o desempenho dos simuladores, utilizamos os seguintes problemas: memória associativa, cobrimento mínimo de vértices, problema do caixeiro viajante, disambiguação léxica e resolução de alguns sistemas lineares.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of requeriments for the degree of Master of Science (M. Sc.).

On the Parallel Simulation of Some Complex Models

Raquel Coelho Gomes Pinto

April, 1994

Thesis Supervisor: Valmir Carneiro Barbosa
Department: Systems e Computational Engineering

A complex system is a large collection, in general, of disparate entities, which share a set of mutual dynamic connections. A dynamic complex system evolves according to a statistical or deterministic set of rules which relate the system at a later time to its state at an earlier time. Examples include the earth's atmosphere, in which the basic entities could be regarded as air density, pressure, velocity, etc., and its connections would be defined by the basic equations of motion. The human brain can be considered also a complex system, where the entities are the neurons and the connections could be the sinapses which link the neurons. The main feature of these systems concerns about their behavior as a whole. Each entity has a local simple behavior, but the way that they interact, generated by the topology of connection, make the collective behavior of the system impredictable. Therefore, it's necessary to simulate the system to verify how it will behave with different inputs and modifications on its topology. Based on this, we developed distributed simulators of some complex systems whose structure and constraints are similar. The implementation of the following models were buit: Hopfield's binary-neuron network, Hopfield's continuous-neuron network, Boltzmann machine, Bayesian network and a neural network for linear system solution. The existence of these models has as the main motivation their application in areas such as artificial intelligence and combinatorial optimization. We used the following problems to test and evaluate the performance of the simulators: content-addressable memory, minimum vertex cover problem, traveling salesman problem, lexical disambiguation and the linear system problem.

Sumário

Capítulo I
1. Introdução.....1
 1.1. Ambiente de Desenvolvimento.....2
 1.2. Resumo dos Capítulos.....3

Capítulo II
2. Modelos Baseados no Protocolo Distribuído de Escalonamento por Reversão de Arestas.....5
 2.1. Descrição do Protocolo.....5
 2.2. Redes Neuronais.....7
  2.2.1. Rede Neuronal Binária de Hopfield.....9
   2.2.1.1. Descrição do Modelo.....9
   2.2.1.2. Simulação Seqüencial.....11
   2.2.1.3. Simulação Paralela Distribuída.....13
    2.2.1.3.1. Algoritmo Original.....13
    2.2.1.3.2. Primeira Versão da Implementação.....16
    2.2.1.3.3. Segunda Versão da Implementação.....29
  2.2.2. Máquina de Boltzmann.....32
   2.2.2.1. Técnica de *Simulated Annealing*.....33
   2.2.2.2. O Modelo da Máquina de Boltzmann.....35
   2.2.2.3. Simulação Paralela Distribuída.....36
 2.3. Rede Bayesiana.....41
  2.3.1. Descrição do Modelo.....41
  2.3.2. Simulação do Modelo.....45

Capítulo III
3. Modelos Baseados no Protocolo Distribuído de Sincronização.....53
 3.1. Descrição do Sincronizador Alfa (α).....53
 3.2. Redes Neuronais.....54
  3.2.1. Rede Neuronal Contínua de Hopfield.....54
   3.2.1.1. Descrição do Modelo.....55
   3.2.1.2. Simulação Paralela Distribuída.....57
  3.2.2. Resolução de Sistemas Lineares.....64
   3.2.2.1. Descrição do Problema.....64
   3.2.2.2. Descrição do Modelo.....65
   3.2.2.3. Simulação Paralela Distribuída.....67

Capítulo IV
4. Resultados Obtidos.....73
 4.1. Memória Associativa.....75
  4.1.1. Rede Neuronal Binária de Hopfield.....75

4.1.2. Rede Neuronal Contínua de Hopfield 76
4.2. Problema de Cobrimento Mínimo de Vértices (CMV) 77
4.2.1. Máquina de Boltzmann 77
4.2.2. Rede Neuronal Binária de Hopfield 81
4.3. Resolução de Ambigüidade Léxica 82
4.4. Problema do Caixeiro Viajante 84
4.5. Resolução de Sistemas Lineares 86
4.6. Observações Finais 87

Referências Bibliográficas 89

Capítulo I

1. Introdução

O objetivo deste trabalho é desenvolver um ambiente de simulação em que seja possível simular paralelamente o comportamento de alguns sistemas complexos que possuem características comuns. Um sistema complexo, em geral, é uma coleção de um grande número de entidades distintas compartilhando um conjunto de conexões mútuas e dinâmicas. Um sistema complexo dinâmico evolui de acordo com um conjunto de regras estatísticas ou determinísticas que relacionam o estado do sistema em um tempo anterior a um posterior. Como exemplo temos a atmosfera terrestre na qual as entidades básicas poderiam ser a densidade do ar, pressão, velocidade, etc., e as conexões poderiam ser definidas pelas equações básicas de movimento. Um outro exemplo de um sistema complexo é uma sociedade humana, composta por homens e mulheres (entidades) e governada por um conjunto de conexões mais complicadas do que no exemplo anterior. Neste exemplo, uma entidade é tipicamente conectada à sua família e seus amigos, e conectada de forma mais fraca aos seus colegas relacionados ao trabalho e recreação.

A principal característica de um sistema complexo é o seu comportamento complexo como um todo. Cada entidade possui um comportamento local simples, mas a interação entre elas, gerada pela forma como são conectadas, torna o comportamento coletivo do sistema imprevisível. Portanto, é necessário simular o sistema para verificarmos seu comportamento frente a diferentes entradas e alterações na sua topologia.

Um processador concorrente é um exemplo de um sistema complexo, onde processamento concorrente é definido pelo uso de várias entidades, idênticas ou heterogêneas, trabalhando em conjunto para alcançar um mesmo objetivo. Em computação concorrente, as entidades são os computadores e o objetivo pode ser um cálculo científico ou uma aplicação de inteligência artificial.

Um exemplo de processador complexo (processador concorrente) é o cérebro humano. Um neurônio é um dispositivo alimentado por várias entradas. Alguns neurônios têm acima de 100.000 entradas detectadas através de um tipo de antena denominada dendritos. As conexões são posicionadas entre os neurônios e sinais químicos são transportados através delas. Este tipo de conexão é denominado sinapse química. Cada neurônio conecta-se com um certo número de neurônios através de seu axônio. Este é usado para transmitir informação a outros neurônios e para emitir dados sensoriais originários de vários dipositivos de entrada utilizados pelo cérebro para coletar informações. Um neurônio executa um complexo cálculo analógico baseado em suas entradas, gerando como resultado sinais a serem propagados a seus vizinhos.

O sistema nervoso é, portanto, um processador concorrente, cujas funções, na sua maioria, são executadas com um desempenho superior ao dos computadores digitais. Sendo assim, encontramos nas redes neuronais um computador concorrente poderoso e flexível que é, em muitos casos, a implementação mais sofisticada de um processamento concorrente.

Uma propriedade fundamental em processamento concorrente é a possibilidade do uso de um sistema complexo para solucionar, calcular ou simular outro sistema complexo. Referenciamos o primeiro sistema como o processador, e o último como o problema. Considerando computação concorrente, o processador (computador complexo) é um dispositivo feito pelo homem, que tem sido projetado e construído com uma grande variedade de entidades fundamentais (nós computacionais), e uma grande variedade de esquemas de interconexão.

Matematicamente, computação concorrente pode ser vista como o mapeamento de um sistema (o problema) em outro (o computador). Um importante subconjunto das propriedades de um sistema complexo é formado por aquelas que caracterizam sua estrutura geométrica ou topológica. Os problemas podem ser efetivamente tratados desde que o sistema computacional possua uma topologia que contenha ou seja mais rica do que a do problema.

Os modelos de sistemas complexos que implementamos são os seguintes: rede neuronal binária de Hopfield, rede neuronal contínua de Hopfield, máquina de Boltzmann, rede Bayesiana, e uma rede neuronal para resolução de sistemas lineares. A utilização de um destes modelos é realizada a partir de uma interface amigável que permite a escolha de qualquer um deles frente a um arquivo de entrada com as especificações dos parâmetros do sistema complexo em questão. Os modelos se subdivedem em duas classes. A primeira engloba a rede neuronal binária de Hopfield, a máquina de Boltzmann e a rede Bayesiana, formando um conjunto de sistemas cujas entidades são binárias, isto é, assumem valores 0 ou 1. Além disso, o conjunto de variáveis aleatórias que compõem a máquina de Boltzmann e a rede Bayesiana formam campos aleatórios de Markov (*Markov Random Field*). E o modelo da rede neuronal binária de Hopfield pode ser considerado basicamente uma rede binária similar ao dois outros modelos onde não utiliza-se as definições probabilísticas.

Os demais modelos compõem a segunda classe onde o comportamento local de suas entidades é controlado por equações diferenciais baseadas no parâmetro de tempo, caracterizando os modelos como sistemas síncronos.

1.1. Ambiente de Desenvolvimento

Um sistema distribuído é uma coleção de computadores independentes em adição a alguma facilidade de comunicação para troca de mensagens. Uma computação distribuída

se constitui de uma coleção de processos em dois ou mais computadores, que precisam se comunicar para alcançar algum objetivo. Tendo isto em vista, para implementarmos uma simulação distribuída dos modelos mencionados, é necessário um ambiente de programação paralelo. Utilizamos como *hardware* o NCP I [1] computador paralelo de alto desempenho em desenvolvimento no Núcleo de Computação Paralela da COPPE/Sistemas. Como o seu projeto ainda está em desenvolvimento, utilizamos um protótipo na implementaçao do simulador, composto por uma rede de 8 nós de processamento interligados ponto a ponto em uma topologia de hipercubo.

Cada nó de processamento é um microprocessador Transputer T800 com memória local de 8 Mbytes. A única forma de troca de informação entre os nós é por meio de troca de mensagens. O ambiente em que o simulador foi desenvolvido, é voltado especialmente para o Transputer e denomina-se *Transputer Development System* (TDS) [2] que é executado sob o IBM DOS. A linguagem de programação paralela associada ao Transputer é a linguagem OCCAM [3, 4].

Esta linguagem permite a recepção não-determinística de mensagens através do comando ALT adicionada à inclusão de recepção de mensagens como guardas. Este comando é composto por várias alternativas dentre as quais o comando seleciona uma a ser executada, cuja condição esteja satisfeita. Estas condições são chamadas guardas.

Por outro lado, toda comunicação entre processos tem capacidade zero, isto é, tanto o emissor quanto o receptor ficam bloqueados até a conclusão da comunicação. Este bloqueio pode ocasionar *deadlocks* dificultando assim a programação. Um dos meios de se prevenir contra *deadlocks* é o uso de *buffers* para o armazenamento das mensagens desbloqueando o emissor, e o receptor retira as mensagens deste *buffer*. Uma outra desvantagem da linguagem OCCAM, é que esta não permite a recursão. No decorrer da apresentação do simulador, descrevemos recursivamente alguns procedimentos, que na realidade tiveram que ser alterados de forma a utilizarem estruturas de pilha para simular a recursão.

A rede de Transputers não oferece nenhuma ferramenta de roteamento de mensagens, de forma que, se um processador deseja enviar uma mensagem para outro processador com o qual não esteja diretamente conectado, é necessário fazer um roteamento por *software* desta mensagem. Por este motivo, utilizamos um *processo de comunicação* [5] implementado em OCCAM cuja função é tratar do roteamento das mensagens pela rede.

1.2. Resumo dos Capítulos

A implementação de todos os modelos estudados tem uma estrutura procedural bem semelhante de forma que a descrição do primeiro modelo é mais detalhada, e a descrição da implementação dos outros diferem na estrutura interna de alguns processos.

No capítulo II apresentamos os modelos que têm em comum certas restrições de implementação que são solucionadas através do uso do protocolo de escalonamento por reversão de arestas. Os modelos são os seguintes: rede neuronal binária de Hopfield, Máquina de Boltzmann e rede Bayesiana.

No capítulo III descrevemos os sistemas complexos estudados que são originalmente sistemas distribuídos síncronos. Como na realidade não é possível se implementar este tipo de sistema, utilizamos como solução o sincronizador α, que é um protocolo cuja função é simular o comportamento síncrono do sistema. Os modelos são os seguintes: rede neuronal contínua de Hopfield e rede neuronal para resolução de sistemas lineares.

No capítulo IV descrevemos os problemas utilizados para coleta de medidas referentes ao *speedup* do sistema. Ilustramos os resultados através de gráficos e apresentamos algumas conclusões relacionadas com a implementação do simulador.

Capítulo II

2. Modelos Baseados no Protocolo Distribuído de Escalonamento por Reversão de Arestas

Neste capítulo será apresentado o estudo e desenvolvimento de simuladores distribuídos de alguns modelos de sistemas complexos que têm certas restrições em comum. Inicialmente mostraremos uma solução para estas restrições possibilitando uma implementação assíncrona dos modelos em questão, e em seguida apresentaremos os modelos.

As restrições acima mencionadas se referem à ordem de operação dos elementos do sistema complexo em questão. As restrições são as seguintes:

- os elementos do sistema devem operar um de cada vez;
- ao longo do tempo os elementos devem operar com a mesma freqüência.

Estas restrições a princípio denotam uma impossibilidade em relação ao paralelismo que se deseja empregar na simulação do modelo. Mas deve-se notar que a restrição referente a operação de um elemento por vez, só se aplica aos pares de elementos que influenciam um ao outro, isto é, aos pares de elementos que possuem uma conexão entre eles. Desta forma, uma solução para o problema de se simular os modelos que seguem estas restrições seria a utilização do **Protocolo de Escalonamento por Reversão de Arestas**.

2.1. Descrição do Protocolo

Podemos representar um sistema complexo na forma de um grafo *G*, onde cada elemento do sistema é representado por um nó do grafo e as conexões entre os elementos pelas arestas do mesmo. Para garantir as restrições apresentadas é necessário que o grafo tenha uma orientação inicial acíclica. O protocolo funciona da seguinte forma:

- todos os nós do grafo *G* que são *sink*, isto é, os nós cujas arestas incidentes são direcionadas a eles, operam;
- a seguir, estes nós revertem as suas arestas incidentes se tornando nós *fontes*;
- retorna-se ao primeiro passo do protocolo.

O protocolo está ilustrado na figura 2.1 e ele garante que os nós vizinhos nunca irão operar concorrentemente e que a freqüência com que operam ao longo do tempo é a mesma para todos os nós.

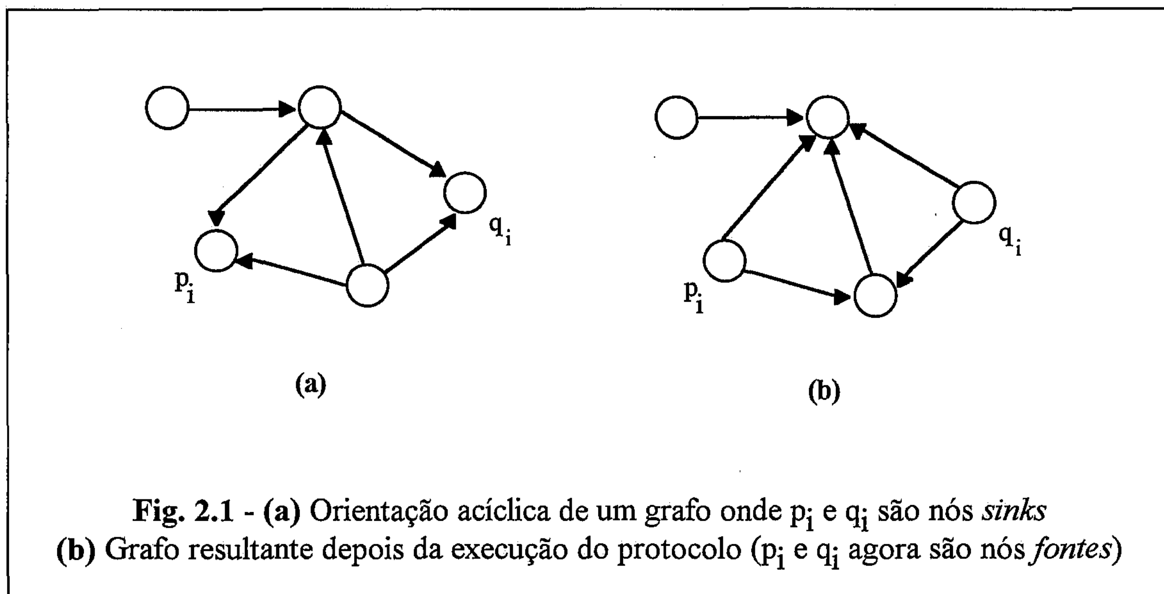


Fig. 2.1 - **(a)** Orientação acíclica de um grafo onde p_i e q_i são nós *sinks*
(b) Grafo resultante depois da execução do protocolo (p_i e q_i agora são nós *fontes*)

A prova de que a primeira restrição é obedecida baseia-se na orientação acíclica do grafo. Inicialmente *G* está orientado aciclicamente de forma que existe pelo menos um nó *sink*. A única forma da restrição ser desobedecida seria se houvesse algum ciclo no grafo durante o procedimento do protocolo. Isto é impossível, visto que a transformação de um nó *sink* em *fonte* não pode formar nenhum ciclo em *G*, pois este ciclo teria que necessariamente incluir o nó em questão já que é o único nó cuja orientação das arestas foi alterada.

A freqüência de operação dos nós também é garantida, visto que depois que um nó *sink* opera e é transformado em um nó *fonte*, ele só operará novamente quando todos os seus nós vizinhos tiverem operado também, isto é, tiverem se transformados em nós *sinks*. Desta forma, todos os nós que são conectados operam com a mesma freqüência ao longo do tempo.

Formalmente, o Escalonamento por Reversão de Aresta tem quatro propriedades que utilizam as definições descritas a seguir. Seja $\Phi(w)$ o conjunto de orientações do grafo *G* que podem ser obtidas a partir de uma orientação inicial w através da operação de vários subconjuntos não-vazios de *sinks*(w). Definimos um escalonamento como uma seqüência infinita de orientações w_k, $k \geq 1$, tal que $w_{k+1} \in \Phi(w_k)$. Denotamos um escalonamento genérico por η, o conjunto de todos os escalonamentos por Ω e o conjunto de todos os escalonamentos iniciados pela orientação w por $\Omega(w)$. Dizemos que um escalonamento é **guloso** (*greedy*) se para $k \geq 1$, w_{k+1} é obtido a partir de w_k

revertendo todas as arestas dos seus *sinks*. Portanto, Ω_g denota o conjunto de todos os escalonamentos gulosos, e $\Phi_g(w)$ a orientação obtida a partir de w através de uma operação gulosa. Definimos também $n_i(\eta, l)$ como o número de vezes que o nó i opera nas l primeiras orientações do escalonamento η, para $l \geq 1$. Em escalonamentos gulosos, nota-se que as orientações geradas são repetidas periodicamente a partir de um certo momento. Seja então a seguinte seqüência de orientações distintas, $\beta_0, ..., \beta_{p-1}$, chamada de um período, tal que $\beta_k = \Phi_g(\beta_{(k-1) \bmod p})$ para $0 \leq k \leq p - 1$, onde p é o seu tamanho. Em vistas destas definições, apresentamos a seguir as quatro propriedades.

i. Sejam w e w' orientações acíclicas de G tal que $w' = \Phi(w)$. Um *sink* em w' tem pelo menos um vizinho que é *sink* em w.

ii. Dado os nós i e j, e dado que o menor percurso entre eles no grafo G contém $r - 1$ arestas, então $| n_i(\eta, l) - n_j(\eta, l) | \leq r - 1$, para todo $\eta \in \Omega$ e $l \geq 1$.

iii. Considerando a orientação w, seja η qualquer escalonamento iniciado por w, e η_g o escalonamento guloso iniciado por w, então $n_i(\eta, l) \leq n_i(\eta_g, l)$ para $l \geq 1$.

iv. Todos os nós operam o mesmo número de vezes em um período.

A propriedade ii traduz a característica de que o escalonamento em questão é livre de *starvation*, podendo ser facilmente demonstrada através de um argumento de indução sobre $r \geq 2$. Ela constata que a freqüência relativa com que os dois nós operam em um escalonamento, não pode diferir de mais do que o número de arestas do menor caminho não-direcionado entre eles.

A propriedade iii pode ser demonstrada por indução em $l \geq 1$ e constata que se seguirmos o escalonamento guloso dada uma orientação inicial, os nós não operam menos freqüentemente a partir desta mesma orientação.

A propriedade iv segue imediatamente da propriedade ii.

2.2. Redes Neuronais

Existem duas abordagens referentes à resolução de problemas. A primeira concepção, chamada **simbolista**, utiliza um processo algorítmico para solucionar um problema, isto é, realiza um desdobramento do problema em tarefas que serão executadas passo a passo.

A segunda abordagem, chamada **conexionista**, sustenta a impossibilidade de se executar algoritmicamente certas tarefas que a mente humana executa com rapidez. Por exemplo: o reconhecimento de imagens, a compreensão de linguagem natural ou a otimização combinatória. A origem desta impossibilidade é devido a complexidade de

programação da solução algorítmica destes problemas ou da sua intensidade computacional, isto é, apesar de um determinado problema ter uma solução clara e concisa, muitos algoritmos não conseguem chegar a uma solução em tempo razoável.

Logo, o objetivo desta abordagem seria desenvolver e projetar um modelo que reproduzisse as capacidades naturais do cérebro humano. Os neurônios vivos conectados por uma complexa rede de sinapses são responsáveis pela riqueza computacional do cérebro. Assim, utiliza-se a estrutura do cérebro para simular a sua capacidade de processamento inteligente de informações.

O sistema nervoso é constituído de dois tipos de células: as células **gliais** e os **neurônios**. As funções dos neurônios são receber informação proveniente do próprio corpo ou do ambiente externo, integrá-la e retransmiti-la a outras células. São os neurônios que nos servem de modelo.

Os neurônios possuem uma rede de prolongamentos formada por um ou mais **dendritos** e um **axônio**. Os dendritos recebem os impulsos nervosos de outros neurônios e os conduz ao corpo celular. Este então processa as informações recebidas gerando novos impulsos que o axônio retransmitirá a outro neurônio. O ponto de contato entre a terminação axônica de um neurônio e o dendrito de outro é chamado de **sinapse**. Esta funciona como uma válvula, controlando a transmissão de impulsos (fluxo de informação) entre os neurônios. Esta capacidade de regulagem é chamada **eficiência sináptica**.

O estudo dos processos computacionais que podem ser realizados por sistemas dinâmicos que obedecem a esse paradigma é chamado **neurocomputação**. E os modelos propostos são chamados **redes neuronais artificiais** ou **modelos conexionistas**.

Os neurônios artificiais são distribuídos pelos espaço e interligados por conexões (sinapses). Através das sinapses os neurônios trocam sinais inibitórios ou excitatórios, competindo ou cooperando entre si. A ação simultânea desta coletividade é que gera um comportamento inteligente. O comportamento de uma rede neuronal não pode ser previsto baseando-se no comportamento individual de cada neurônio. Desta forma, a excitação de alguns neurônios desencadeia uma fase "transitória" cujo término não pode ser previsto com base no comportamento individual das unidades que a integra.

Em uma simulação, as características inteligentes de uma rede têm origem na sua topologia e nas regras de aprendizagem dos neurônios. A topologia de uma rede neuronal descreve fatos como por exemplo: quantas conexões existem para cada neurônio; a quais neurônios este está interligado (aos vizinhos mais distantes, aos mais próximos ou a uma combinação dos vizinhos próximos e distantes); e de maior importância, o que a rede neuronal está tentando representar internamente com uma determinada topologia.

As regras de aprendizagem de um neurônio descreve como este interpreta os sinais recebidos dos neurônios aos quais está conectado, e, baseado nesta interpretação, qual sinal transmitir pela rede.

Logo, um modelo de uma rede neuronal usa uma determinada topologia, um tipo de neurônio, e uma regra de aprendizagem para as interações e interrelações dos seus constituintes fundamentais (os neurônios e suas conexões).

As aplicações de redes neuronais são inúmeras. Existem problemas matemáticos para os quais ainda não se conseguiu obter soluções satisfatórias e cujo tratamento parece poder se beneficiar da capacidade de computação coletiva das redes neuronais. Entre eles estão problemas que envolvem encontrar, dentre um grande número de possibilidades, a melhor opção, segundo um critério pré-estabelecido. O ramo da ciência da computação que trata destas questões é chamado de **otimização combinatória**.

Um outro campo de atuação dos modelos conexionistas é o campo da **Inteligência Artificial**. Os principais problemas de inteligência artificial cujo uso de redes neuronais tem sido estudado são os problemas que envolvem reconhecimento de padrões visuais ou auditivos e processamento de linguagem natural.

Para uma melhor compreensão dos modelos a serem descritos definiremos cada uma das unidades computacionais do modelo, os neurônios, estabelecendo uma nomenclatura coerente e um conjunto de expressões matemáticas que representem as características funcionais dessas unidades. Uma rede neuronal é constituída de n neurônios $N_1, N_2, \ldots, N_n$. O estado do neurônio N_i é seu valor de saída V_i, um número real no intervalo [0, 1]. O valor de V_i é determinado pela entrada P_i do neurônio (seu potencial de entrada), que, por sua vez, depende dos estados dos outros neurônios. V_i e P_i estão interrelacionados pela característica entrada-saída do neurônio, isto é, uma função f_i de forma que $V_i = f_i(P_i)$.

2.2.1. Rede Neuronal Binária de Hopfield

2.2.1.1. Descrição do Modelo

Neste modelo cada neurônio N_i pode assumir dois estados, 0 ou 1, caracterizando assim a rede como binária. A saída dos neurônios são os valores dos seus estados, logo, só se pode atribuir a V_i (saída do neurônio N_i) os valores 0 ou 1. Um neurônio N_i está, a princípio, conectado a todos os outros neurônios N_j com um peso denominado $w_{ij} = w_{ji}$ (onde $w_{ii} = 0$). O elemento w_{ij} pode ser biologicamente visto como a força sináptica de interconexão de N_j para N_i. O potencial de entrada P_i de cada neurônio N_i têm duas fontes: entrada externa I_i e entradas provenientes de outros neurônios. O valor de P_i é determinado pela seguinte equação:

$$P_i = \sum_{j \neq i} w_{ij} V_j + I_i \qquad \textbf{(1)}$$

Um neurônio N_i atualiza seu estado avaliando o seu potencial de entrada P_i e aplicando a seguinte regra (f_i) descrita na figura 2.2:

$$V_i = \begin{cases} 0, & \text{se } P_i < \theta_i \\ 1, & \text{se } P_i > \theta_i \end{cases} \qquad (2)$$

onde θ_i é o limiar de N_i.

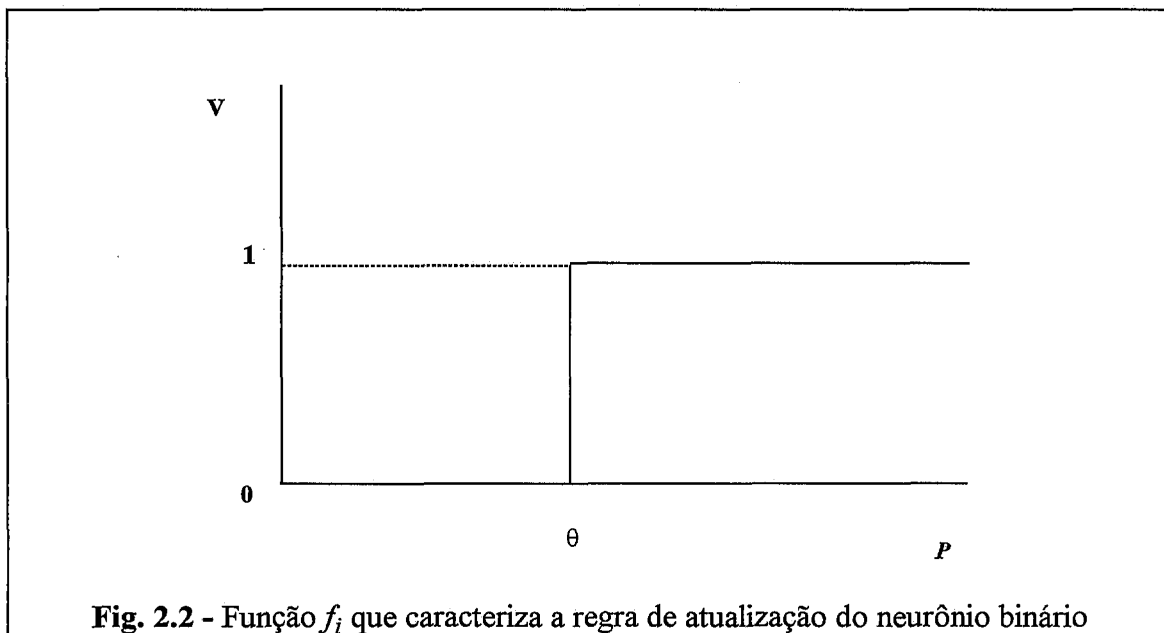


Fig. 2.2 - Função f_i que caracteriza a regra de atualização do neurônio binário

Além desta regra, o sistema tem que obedecer às seguintes restrições: um único neurônio deve ser atualizado por vez; e todos os neurônios têm que ser atualizados com a mesma freqüência ao longo do tempo.

O estado instantâneo do sistema é especificado pela listagem dos n valores de V_i, isto é, o estado é representado por uma palavra binária de n bits. Quando o fluxo espacial de estados gerado pelo algoritmo ao longo do tempo é caracterizado por um conjunto de pontos estáveis e fixos, dizemos que o modelo se comporta como uma **memória associativa**. Se estes pontos estáveis descrevem um espaço no qual pontos próximos tendem a se manter nas suas proximidades durante o fluxo de novos estados, então estados iniciais que estão perto (de acordo com a distância de Hamming) de um determinado estado estável e longe de todos os outros, tenderão a estabilizar neste estado.

Analogamente, se a localização de um determinado ponto estável no espaço de estados é visto como a informação de uma certa memória do sistema, estados próximos a

este ponto estável contêm informação parcial sobre esta memória. A partir de um estado inicial com informação parcial de uma memória, alcança-se um estado estável final com toda informação da memória em questão. Por conseguinte, a memória não é alcançada através de um endereço, mas através de uma subparte da memória fornecida pelo estado inicial. A matriz de sinapses w contém simultaneamente várias memórias, que são individualmente reconstruídas a partir de informações parciais fornecidas pelos estados iniciais.

A garantia de que o sistema sempre irá convergir para um estado estável é a característica essencial deste modelo. A prova desta propriedade se baseia na construção de uma **função de energia** que seja monotonicamente decrescente, isto é, durante qualquer mudança de estado gerada pelo algoritmo, a função decresce. Considere a seguinte função de energia:

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}\, V_i\, V_j - \sum_{i} I_i\, V_i + \sum_{i} \theta_i\, V_i \quad (3)$$

A variação ΔE em E devido a mudança do estado de N_i em ΔV_i é dado por:

$$\Delta E = -\left(\sum_{j\neq i} w_{ij}\, V_j + I_i - \theta_i\right)\Delta V_i \quad (4)$$

De acordo com o algoritmo, ΔV_i é positivo apenas quando a expressão entre parênteses for positiva, e de forma similar para ΔV_i negativo. Assim, qualquer mudança em E ocasionada pelo algoritmo é negativa. A função E é limitada de forma que as iterações do algoritmo devem conduzir a estados estáveis que não se alteram com o tempo.

2.2.1.2. Simulação Seqüencial

A simulação seqüencial da Rede Neuronal Binária de Hopfield é simples e direta, e foi baseada no algoritmo descrito em [6] que apresenta os seguintes passos:

1. Escolha aleatoriamente um neurônio N_i a partir de uma distribuição que garanta a mesma freqüência de atualização para todos os neurônios;

2. Calcule P_i e atualize V_i de acordo com a regra descrita na seção anterior. Se a convergência do sistema for detectada, então termine a simulação. Caso contrário, prossiga para o passo 1.

Desenvolvendo o primeiro passo, escolhemos como ordem de atualização dos neurônios a mais simples possível, que seria enumerar os neurônios e atualizá-los em ordem cíclica, isto é, atualiza-se os neurônios na ordem N_1, N_2, . . ., N_n e depois de se atualizar o n-ésimo neurônio, retorna-se ao neurônio N_1 formando um ciclo. O passo dois

se resume em atualizar o neurônio em questão e testar a estabilidade do sistema. Para se testar a estabilidade, é suficiente verificar se o estado global do sistema, que é formado pelos estados dos neurônios, se mantém constante. Tendo isto em vista, é necessário armazenar o estado anterior de cada neurônio quando este for atualizado. Não é preciso verificar a estabilidade da rede a cada vez que um neurônio for atualizado, basta testar a convergência do sistema quando um novo estado global estiver completo, isto é, quando todos os neurônios tiverem sido atualizados pelo mesmo número de vezes. O algoritmo do simulador seqüencial está descrito na figura 2.3.

Ler a matriz w e o estado inicial do sistema armazenando-o em V
estável ← falso
Enquanto não estável faça
    { atualizar os neurônios }
    Para cada neurônio i faça
        $P_i \leftarrow 0$
        Para cada neurônio N_j tal que $w_{ij} \neq 0$ faça
            Se N_j já foi atualizado então faça
                $P_i \leftarrow P_i + w_{ij} \cdot V_j$
            Senão
                $P_i \leftarrow P_i + w_{ij} \cdot V_j^{\text{antigo}}$
            Fim-se
        Fim-para
        $V_i^{\text{antigo}} \leftarrow V_i$
        Se $P_i \leq \theta_i$ então faça
            $V_i \leftarrow 0$
        Senão
            $V_i \leftarrow 1$
        Fim-se
    Fim-para

    { testar estabilidade }
    Se para cada neurônio i, V_i for igual a V_i^{antigo} então faça
        estável ← verdade
    Fim-se
Fim-enquanto

Fig. 2.3 - Algoritmo do Simulador Seqüencial

2.2.1.3. Simulação Paralela Distribuída

As restrições descritas na seção 2.1 alegando que um único neurônio deve ser atualizado por vez e que as freqüências de atualização dos neurônios ao longo do tempo devem ser iguais, denotam, a primeira vista, uma impossibilidade de paralelismo. Mas é importante perceber que a primeira restrição se aplica apenas em pares de neurônios que se influenciam mutuamente. Desta forma, se dois neurônios N_i e N_j são conectados por uma sinapse cujo $w_{ij} = 0$, eles podem ser atualizados concorrentemente. Representando a rede neuronal por um grafo G onde cada nó representa um neurônio e as arestas representam as conexões sinápticas de peso não-nulo que interligam os neurônios, a condição suficiente e necessária para que dois neurônios possam ser atualizados concorrentemente, é a de que eles não sejam vizinhos dentro do grafo G. Em vista disto, podemos concluir que quanto mais denso for o grafo, menor será o nível de concorrência das atualizações dos neurônios.

Empregaremos o protocolo de escalonamento por reversão de arestas descrito na seção 2.1 como solução do problema de simular o comportamento dos n neurônios assincronamente de forma que as restrições sejam respeitadas. A adaptação do protocolo ao problema em questão é a seguinte: para cada neurônio N_i teremos um processo p_i que irá simular o seu comportamento. E cada aresta do grafo G interligando N_i e N_j, corresponderá a um canal de comunicação conectando p_i e p_j. De acordo com o protocolo, fornece-se uma orientação inicial acíclica ao grafo G, e a vez de operar é dada ao processos que são *sinks* em G. Quando estes processos terminam de atualizar seus respectivos neurônios, eles revertem a orientação das suas arestas incidentes tornando-se *fontes*. Esta reversão de orientação é implementada pelo envio dos novos estados para os processos vizinhos. Este protocolo é repetido continuamente até se alcançar a estabilidade da rede neuronal.

2.2.1.3.1. Algoritmo Original

A seguir descreveremos o algoritmo no qual a construção do simulador foi baseada. Este algoritmo encontra-se em [6]. A "arquitetura" do *software* é constituída de n processos, representando os n neurônios, que se comunicam por canais de acordo com a estrutura do grafo G, e de um processo denominado **processo mestre (p_M)** que se comunica com todos os n processos através de canais bidirecionais. Toda vez que um processo atualiza o seu respectivo neurônio, ele envia este novo estado para o processo mestre de forma que as funções de p_M são basicamente as seguintes:

- controlar a inicialização dos processos no que se refere a distribuição de dados e início de processamento;
- receber os estados dos neurônios e, a cada estado global completado, verificar se a rede neuronal se estabilizou;

- terminar o processamento da rede quando a estabilidade for detectada.

O teste de estabilidade executado por p_M é realizado da seguinte forma: o processo mestre armazena os estados enviados pelos processos de neurônios até formar um estado global (também chamado *snapshot*) completo. Para se verificar a estabilidade é suficiente que p_M compare o novo estado global com o imediatamante anterior. Se os dois *snapshots* forem iguais, a rede então convergiu para um estado estável. O processo mestre, por conseguinte, armazena os estados do sistema em uma matriz na qual cada linha corresponde a uma estado global e cada coluna ao conjunto dos estados de um processo ao longo do tempo. O estado global inicial é formado pelos estados iniciais dos neurônios. Não é necessário que os processos enviem os valores de seus respectivos neurônios a cada vez que são atualizados, é suficiente que este envio seja feito com uma certa freqüência. O p_M tem que guardar espaço de memória para armazenar no máximo n - 1 estados globais incompletos. A prova de que este número de estados globais é suficiente se encontra em [7]. Logo abaixo temos o algoritmo mais detalhado do processo mestre:

1. Enviar cada w_{ij} lido como entrada para p_i e p_j, para todo $i \neq j$ e $1 \leq i, j \leq n$.

2. Enviar cada V_i^{inicial} lido como entrada para p_i, para todo $1 \leq i \leq n$. Estes valores formam o estado global inicial do sistema.

3. Quando um valor 0 ou 1 é recebido de um processo de neurônio, verificar se, com a adição deste novo, um estado global é completado. Se for o caso, testar a estabilidade da rede, comparando este novo estado global com o anterior. Senão, repetir o passo 3. Se o sistema tiver convergido, p_M deve terminar a simulação enviando uma mensagem de STOP para cada processo e esperando uma mensagem de STOP de cada um como retorno. Durante esta espera, p_M ignora quaisquer 0's ou 1's recebidos. Caso contrário, repetir o passo 3.

O processo p_i ao receber os valores de w_{ij} para todo $j \neq i$, tem que estabelecer uma orientação para suas arestas incidentes, tornando a orientação do grafo G acíclica. Utilizamos a regra em que uma aresta é direcionada de p_i para p_j, se e somente se $i < j$. Denominamos então p_i de vizinho *upstream* de p_j, que, por sua vez, é um vizinho *downstream* de p_i.

Depois de estabelecer esta orientação, cada processo p_i se torna **inativo**. O processo p_i só passará para o estado **ativo** quando este tiver recebido seu valor inicial transmitido por p_M, enviando-o para seus vizinhos *downstreams*, e tiver recebido os valores iniciais dos seus vizinhos *upstreams*. A partir do momento que um processo se torna **ativo**, ele pode trocar mensagens com seus vizinhos e atualizar seu neurônio. Simplificaremos a explicação do processamento de p_i com o uso das seguintes variáveis: a variável $n.viz_i$ armazena o número de vizinhos de p_i; $n.up_i$ contém o número de vizinhos *upstreams* de p_i; e o vetor up_i (*j*) indica se p_j é um vizinho *upstream* de p_i. O fato do

processo p_i receber um valor V_j enviado por um vizinho *downstream* p_j, significa que a aresta entre os dois processos passou a ser direcionada de p_j para p_i, tornando p_j um vizinho *upstream* de p_i. Logo, deve-se incrementar $n.up_i$ de 1, tornar $up_i\ (j)$ verdade e verificar se p_i se tornou um *sink* no grafo G, testando se $n.viz_i = n.up_i$. Caso p_i seja um *sink*, este computa o valor de P_i e atualiza V_i, enviando-o para todos os seus vizinhos e para p_M. Além disso, atribui-se o valor 0 para $n.up_i$ e torna-se $up_i\ (j)$ igual a falso para todo $j \neq i$ de forma que p_i e p_j sejam vizinhos.

A seguir apresentamos com maiores detalhes o algoritmo de um processo p_i no qual utilizamos as seguintes variáveis: $n.flushed_i$ que contém o número de canais por onde p_i já recebeu uma mensagem de STOP desde o primeiro STOP recebido; e $n.init_i$ que acumula o número de valores iniciais que p_i recebeu de p_M e dos vizinhos *upstreams* enquanto esteve inativo. Ainda fazemos referência a dois procedimentos: **operar** que envolve os passos desde a verificação se um processo se tornou *sink*, até a atualização do neurônio; e **ativar** que se constitui em verificar se $n.init_i = n.up_i + 1$, e se for o caso, atribuir o valor 0 para $n.init_i$ e tornar p_i **ativo**.

1. Acumular o número de processos p_j tal que $j \neq i$ e $w_{ij} \neq 0$, em $n.viz_i$.
2. Contar o número de processos p_j tal que $1 \leq j \leq i - 1$ e $w_{ij} \neq 0$, armazenando o resultado em $n.up_i$. E para cada um destes processos atribuir verdade para $up_i\ (j)$.
3. Atribuir o valor falso para $up_i\ (j)$ tal que $i + 1 \leq j \leq n$ e $w_{ij} \neq 0$.
4. Ao receber um valor 0 ou 1 de p_M, armazenar este valor em V_i, enviá-lo para todos os vizinhos *downstream*, incrementar $n.init_i$ de 1 e executar **ativar**. Se p_i se tornar ativo então executar **operar**.
5. Ao receber um valor 0 ou 1 de um vizinho *upstream* p_j, armazenar este valor em V_j, incrementar $n.init_i$ de 1 e executar **ativar**. Se p_i se tornar ativo então executar **operar**.
6. Ao receber um valor 0 ou 1 de um vizinho *downstream* p_j, armazenar este valor em V_j, incrementar $n.up_i$ de 1 e atribuir verdade para $up_i\ (j)$. Se p_i estiver ativo, executar **operar**. Caso contrário, se $n.flushed_i = 0$, incrementar $n.init_i$ de 1.
7. Ao receber uma mensagem de STOP enviada por p_M ou por qualquer processo vizinho, incrementar $n.flushed_i$ de 1, e se $n.flushed_i = 1$, enviar STOP para todos os seus vizinhos. Por outro lado, se $n.flushed_i = n.viz_i + 1$, enviar uma mensagem de STOP para p_M e terminar.

Deve-se notar que os três primeiros passos são etapas de inicialização, e os seguintes são passos ativados por recebimento de mensagens.

2.2.1.3.2. Primeira Versão da Implementação

A implementação mais direta deste algoritmo seria alocar cada processo a um processador. Entretanto, como o ambiente de processamento utilizado é constituído por uma rede de 8 *Transputers T-800* interligados por uma topologia de hipercubo, esta alocação se torna inviável, pois o sistema teria que ter no máximo apenas oito processos sendo um deles o processo mestre, equivalendo a uma rede de sete neurônios. Uma outra opção seria a de alocarmos vários processos em cada processador. A desvantagem desta solução é o fato de que processos em um mesmo processador têm que trocar mensagens através de canais lógicos para se comunicarem. Esta troca de mensagens ocasiona um *overhead* desnecessário, visto que os processos, estando em um único processador, têm memória compartilhada. A partir disto, chegamos a nossa solução final, na qual temos um único processo em cada processador, responsável por vários neurônios. Assim, os neurônios são representados por estruturas de dados e a troca de mensagens só é realizada entre processadores através de canais físicos.

O ***processo de comunicação*** descrito em [5] é responsável pelo roteamento das mensagens trocadas pelos processadores conectados por uma topologia de hipercubo. Os processadores serão enumerados de 0 a 7 e denominaremos cada processo alocado a um processador i de p_i tal que $0 \leq i \leq 7$. De acordo com o mencionado anteriormente, o algoritmo descrito serve de base para a construção do simulador, tendo os dois vários pontos em comum, dentre eles a utilização de um *processo mestre* (p_M) que se comunica com todos os outros processos, mantendo basicamente as mesmas funções. Cada processo p_i será responsável por um conjunto de neurônios, e se comunicará com p_M e com os processos p_j cujo conjunto de neurônios contenha pelo menos um vizinho dos neurônios de p_i. O processo p_M e um dos processos p_i (o processo p_0) serão alocados no processador 0 de forma que a sua estrutura, ilustrada na figura 2.4, seja composta por:

- ***processo mestre***: responsável pela distribuição dos dados de entrada aos processos p_i, pela coleta dos valores dos neurônios, pela detecção da estabilidade da rede e terminação do sistema.

- **processo p_0**: responsável pela atualização dos seus respectivos neurônios.

- ***interface de entrada***: processo responsável pelo direcionamento de uma mensagem de origem externa destinada para p_M ou p_0.

- ***interface de saída***: processo responsável pela coleta das mensagens enviadas por p_M e p_0 com destino a outros processadores.

- ***processo de comunicação***: responsável pelo roteamento e entrega das mensagens dentro do hipercubo.

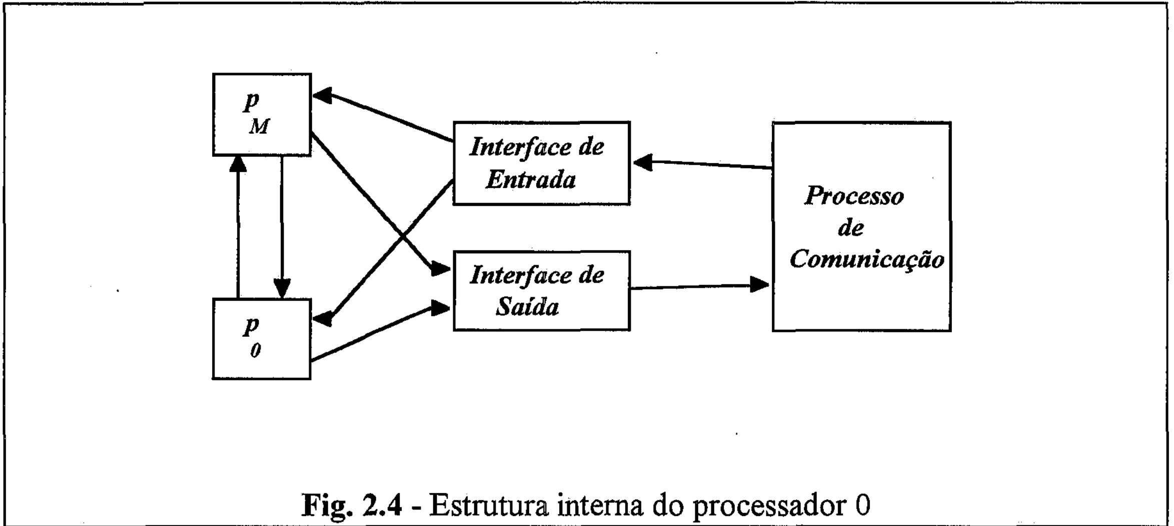


Fig. 2.4 - Estrutura interna do processador 0

Os processos p_i, tal que $1 \leq i \leq 7$, são alocados nos processadores 1 a 7 respectivamente. Na figura 2.5 temos a visualização geral do sistema em uma rede de 4 *Transputers*. Descreveremos cada um dos processos mencionados apresentando seus algoritmos de forma detalhada.

Fig. 2.5 - Estrutura dos processos dentro de uma rede de 4 processadores

Descrição do Processo Mestre

O simulador visualiza a rede neuronal como um grafo G, onde cada processador é responsável por um subgrafo de G. Deve-se considerar que o grafo não precisa ser totalmente interligado, isto é, podem existir nós em G a partir dos quais não se consegue acessar todos os outros nós por meio das arestas. Então, é possível que existam subgrafos separados dentro do grafo G. O sistema trata estes subgrafos separadamente no sentido de que um subgrafo pode estabilizar antes do grafo inteiro ter convergido. Este tratamento é realizado com o intuito de diminuir o número de comunicações, pois ao se detectar que um subgrafo atingiu um estado estável, não é necessário continuar atualizando os neurônios representados por este subgrafo. Ainda pode ocorrer de um subgrafo ser inteiramente alocado em um único processador. Neste caso, a evolução dos estados do subgrafo independe dos outros neurônios e, por conseguinte, dos outros processadores. Ele é, então, tratado internamente e de forma isolada, sendo por isto denominado de **subgrafo isolado**. O processo mestre é o responsável pela verificação da existência de subgrafos gerando um número de identificação para cada um, testando também se são isolados.

Para um melhor entendimento do algoritmo de p_M, definiremos as seguintes variáveis:

- *snapshot (i, j)* - matriz onde se armazena os estados globais do sistema. Cada linha i representa um snapshot e cada coluna j contém os estados do neurônio j ao longo do tempo.
- *estabilizou (j)* - indica se o subgrafo j estabilizou.
- *subgrafo (i)* - armazena a identificação do subgrafo ao qual o neurônio i pertence.
- *isolado (j)* - indica se o subgrafo j é isolado.
- *flushed* - acumula o número de STOP's recebidos.

Primeiramente, apresentaremos genericamente os passos executados por p_M, e a seguir especificaremos algumas etapas.

1. Ler como entrada w e V^{inic}, enviar w_{ij} para o(s) processador(es) que contém os neurônios i e/ou j, e enviar V^{inic} para todos os processadores.
2. Separar os conjuntos de neurônios que formam subgrafos enumerando-os e armazenando suas identificações em *subgrafo*. Identificar aqueles que são isolados e enviar para os processos p_i os valores de *subgrafo (j)* para cada neurônio j alocado em p_i.

3. Ao receber uma mensagem enviada por p_0 ou pela interface de entrada, verificar se é um STOP. Caso afirmativo, incrementar *flushed* de 1 e se este se tornar igual ao número de processadores, então enviar uma mensagem de término para as duas interfaces e finalizar p_M. Se a mensagem não for STOP, executar o procedimento **tratar.mensagem**.

Os passos 1 e 2 são etapas de inicialização, e o passo 3 é ativado pelo recebimento de mensagens. A identificação dos subgrafos dentro do grafo *G*, mencionada no passo 1, é realizada através de um algoritmo de encaminhamento em grafos mostrado na figura 2.6. O algoritmo baseia-se no fato de que dado um subgrafo, é possível "visitar" todos os seus nós a partir de qualquer um deles. Utilizamos então a recursão para, a partir de um nó, visitarmos todos os outros nós do subgrafo através de um encaminhamento pelas suas arestas, alocando-os ao subgrafo em questão. Para descobrir se um subgrafo é isolado, verifica-se se todos os seus nós estão alocados no mesmo processador. Sendo este o caso, o subgrafo é considerado isolado.

```
Procedimento Visitar (neurônio)
    Subgrafo (neurônio) ← ident.subgrafo
    Para cada neurônio N_j tal que (w_neurônio j ≠ 0) faça
        Se j não está alocado a nenhum subgrafo então
            Visitar (j)
        Fim-se
    Fim-para
Fim-proc

{ início do algoritmo }
ident.subgrafo ← 0
Para cada neurônio i faça
    Se i não está alocado a nenhum subgrafo então
        Visitar (i)
        ident.subgrafo ← ident.subgrafo + 1
    Fim-se
Fim-para
```

Fig. 2.6 - Algoritmo de identificação dos subgrafos dentro do grafo *G*.

O procedimento **tratar.mensagem** inicialmente armazena o novo estado do neurônio *j*, recebido pela mensagem, em *snapshot (i, j)*, onde *i* corresponde ao primeiro estado global incompleto que não contém um valor para o estado do neurônio *j*. Seja *k* a identificação do subgrafo do qual o neurônio *j* faz parte. Se, com o valor recebido, o estado do subgrafo *k* se torna completo e se *k* é um subgrafo isolado, atribui-se o valor

verdade para *estabilizou (k)*. Neste caso, é garantido que o subgrafo k está estável, porque sendo ele isolado toda sua evolução é realizada dentro de um processador, portanto é desnecessário enviar os estados intermediários para p_M pois o próprio processador pode identificar quando o subgrafo se estabiliza enviando apenas seu estado final. Caso o subgrafo k não seja isolado, verifica-se a sua estabilidade a partir do novo estado completado. Esta checagem é feita comparando-se o novo estado com o imediatamente anterior, e se forem iguais, atribui-se o valor verdade para *estabilizou (k)* e envia-se uma mensagem de STOP.PARCIAL para todos os processadores que contém algum neurônio i tal que *subgrafo (i)* = k. Sempre que se alterar o valor de *estabilizou*, verifica-se se toda a rede já convergiu, testando se para todos os subgrafos k, *estabilizou (k)* = verdade. Neste caso, p_M envia uma mensagem de STOP para todos os processos p_i.

Descrição do Processo p_i

Os processos p_i, tal que $0 \leq i \leq 7$, são compostos por quatro processos em paralelo de acordo com a figura 2.7. A única diferença entre o processo p_0 e os outros, se refere ao número de canais de comunicação e com quais processos p_0 se comunica. A estrutura interna de p_0 se apresenta na figura 2.8.

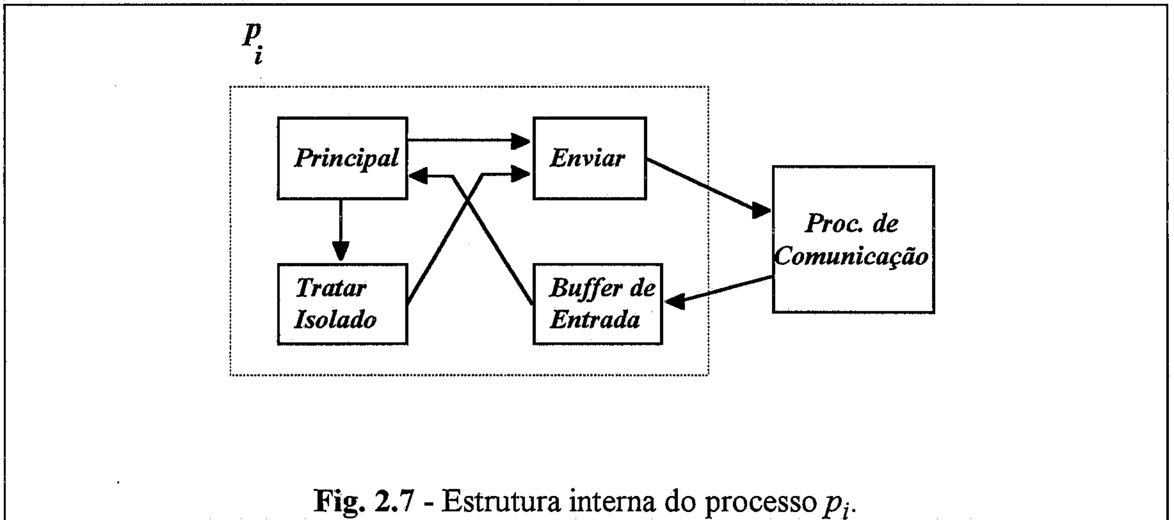


Fig. 2.7 - Estrutura interna do processo p_i.

Discutiremos a seguir o objetivo de cada um dos processos internos de p_i assim como a motivação de sua existência.

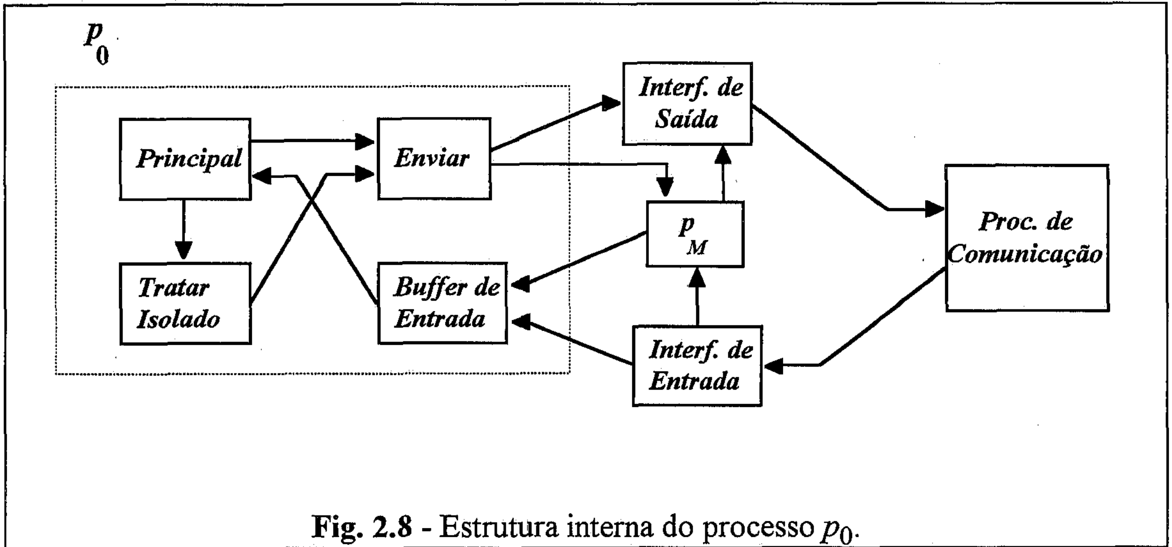


Fig. 2.8 - Estrutura interna do processo p_0.

→ Processo ***Buffer de Entrada***

No algoritmo original não há a possibilidade de ocorrer um *deadlock*, visto que p_i está sempre esperando por mensagens até se tornar *sink*, quando então atualiza o valor do seu neurônio. Neste momento, nenhum outro processo p_j, tal que $w_{ij} \neq 0$, tentará lhe enviar qualquer mensagem, pois todos os vizinhos de p_i estarão esperando, pelo menos, pelo novo estado do neurônio i.

No novo algoritmo, a situação é diferente. Agora, cada processo p_i é responsável pelo estado de vários neurônios de forma que, quando estiver enviando o valor de um de seus neurônios para um processo p_j, este poderá estar tentando enviar também uma mensagem para p_i, ocasionando um *deadlock*.

Um meio de evitar este *deadlock* é criar um *buffer* com a função de armazenar todas as mensagens destinadas a p_i, desbloqueando todos os processadores que desejam lhe enviar qualquer mensagem.

A implementação do *buffer* envolve um problema de **exclusão mútua**. Na maioria dos sistemas computacionais, processos competem entre si pela requisição e posse exclusiva de recursos. Em adição, processos podem cooperar através de interações envolvendo dados compartilhados ou através de troca de mensagens. De qualquer forma, a exclusão mútua é necessária, isto é, deve-se garantir que, em cada momento, apenas um processo possa utilizar um recurso ou alterar informação compartilhada.

O *buffer*, dentro de um ambiente distribuído, é um exemplo do problema **produtor/consumidor** que ocorre freqüentemente entre processos que cooperam entre si. Genericamente, temos um conjunto de processos produtores que fornecem mensagens

para um conjunto de processos consumidores. Todos eles compartilham um conjunto comum de espaços onde as mensagens podem ser armazenadas pelos produtores ou removidas pelos consumidores.

Uma forma usual de um *buffer* é um **buffer circular**, que pode ser representado por um vetor com dois ponteiros. Um ponteiro (**topo**) indicando o próximo espaço vazio no *buffer* e o outro (**base**) apontando para o próximo elemento a ser retirado do *buffer* (ver figura 2.9).

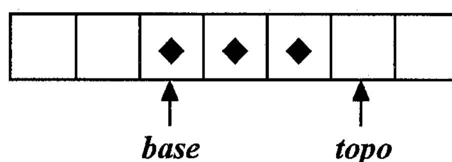


Fig. 2.9 - *Buffer* que segue a ordem FIFO (*first in first out*)

Através das operações de produção e consumo realizadas sobre o *buffer*, os respectivos ponteiros são incrementados usando aritmética modular, de forma que ao atingirem o final do vetor, são desviados para o seu início.

A implementação concorrente do *buffer* circular deve satisfazer duas condições de sincronização: não é possível retirar uma mensagem de um *buffer* vazio, e não se pode incluir uma mensagem em um *buffer* cheio.

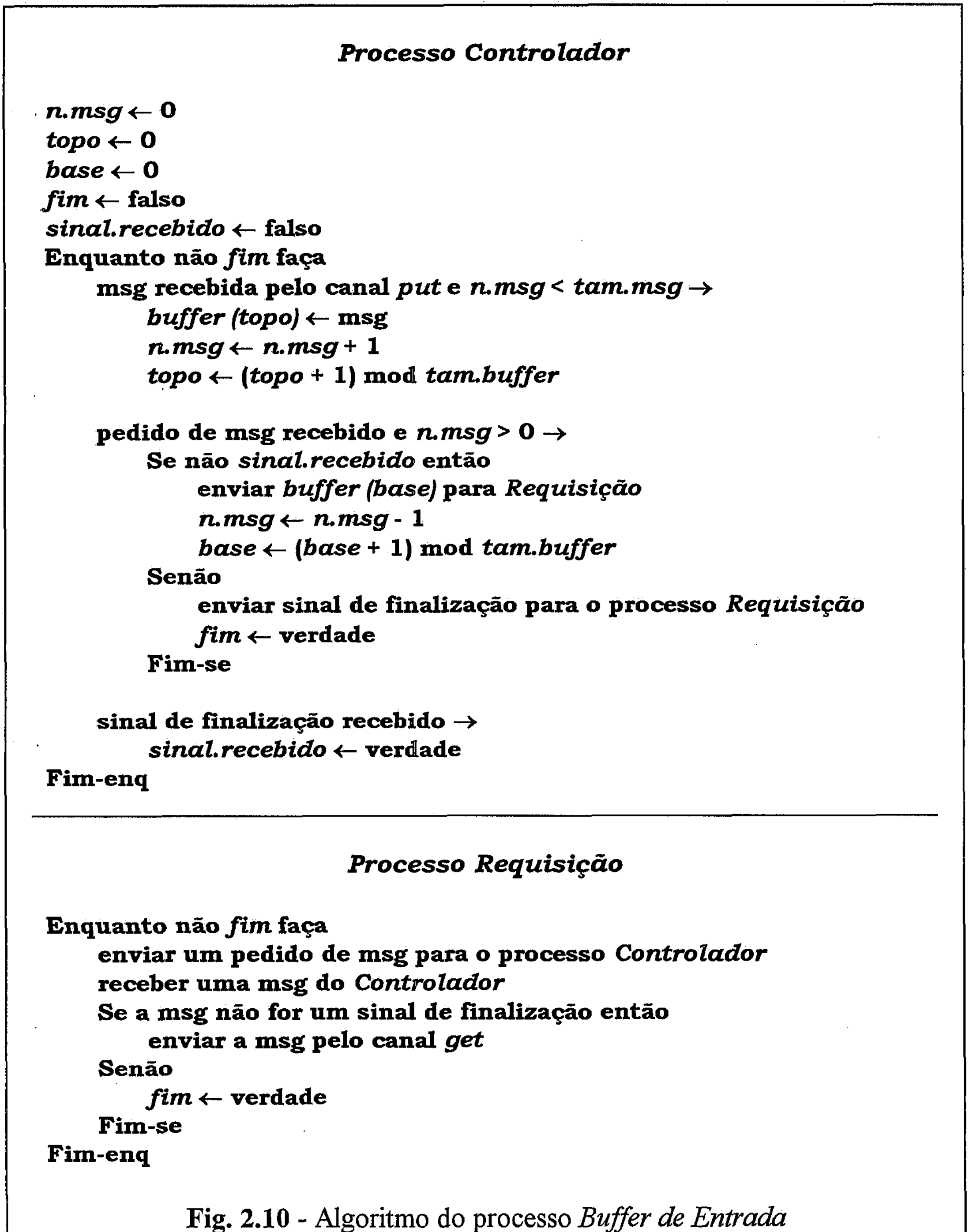
Nomearemos os canais incidentes ao processo *buffer de entrada* para uma melhor compreensão do seu algoritmo:

- *put* - canal de comunicação orientado do *processo de comunicação* para o *buffer*.
- *get* - canal de comunicação orientado do *buffer* para o *processo principal*.

No nosso caso, o *processo de comunicação* é o produtor e o *processo principal* faz o papel de consumidor. Na figura 2.10 temos o algoritmo do *buffer*, que utiliza as seguintes variáveis:

- *tam.buffer* - armazena o número máximo de mensagens que o *buffer* pode conter.
- *n.msg* - acumula o número de mensagens armazenadas no *buffer*.
- *fim* - indica o término do algoritmo.

- *sinal.recebido* - indica se o sinal de finalização foi recebido pelo processo.



Processo Controlador

```
n.msg ← 0
topo ← 0
base ← 0
fim ← falso
sinal.recebido ← falso
Enquanto não fim faça
    msg recebida pelo canal put e n.msg < tam.msg →
        buffer (topo) ← msg
        n.msg ← n.msg + 1
        topo ← (topo + 1) mod tam.buffer

    pedido de msg recebido e n.msg > 0 →
        Se não sinal.recebido então
            enviar buffer (base) para Requisição
            n.msg ← n.msg - 1
            base ← (base + 1) mod tam.buffer
        Senão
            enviar sinal de finalização para o processo Requisição
            fim ← verdade
        Fim-se

    sinal de finalização recebido →
        sinal.recebido ← verdade
Fim-enq
```

Processo Requisição

```
Enquanto não fim faça
    enviar um pedido de msg para o processo Controlador
    receber uma msg do Controlador
    Se a msg não for um sinal de finalização então
        enviar a msg pelo canal get
    Senão
        fim ← verdade
    Fim-se
Fim-enq
```

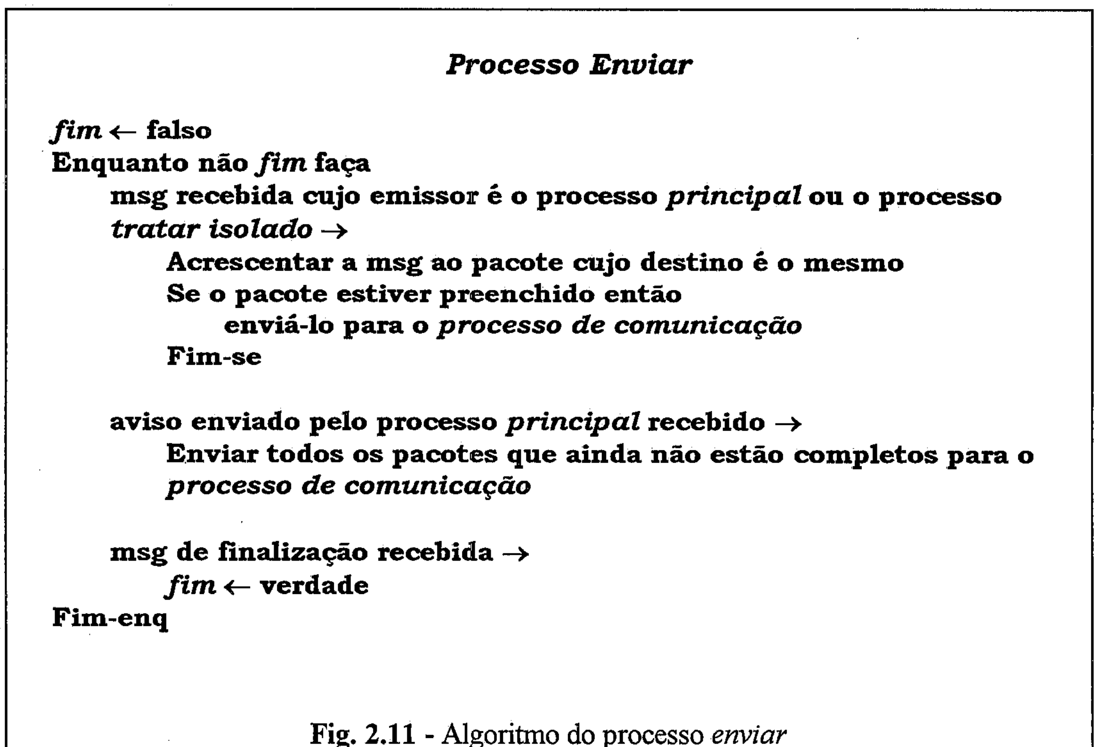
Fig. 2.10 - Algoritmo do processo *Buffer de Entrada*

O algoritmo é constituído por dois processos em paralelo, o processo ***Controlador*** e o processo ***Requisição***. No processo *controlador* (dentro da estrutura "enquanto" mais especificamente), utilizamos uma estrutura de **guardas**, onde escolhe-se, de forma arbitrária, qual procedimento a ser executado correspondente a um dos guardas cuja condição estiver satisfeita. De acordo com o algoritmo, o processo *requisição* está sempre com uma mensagem esperando para ser enviada pelo canal *get*. Quando o processo *principal* precisar de uma mensagem, basta este sincronizar com o processo *requisição* através do canal recebendo assim a mensagem desejada.

O tamanho máximo do *buffer* tem de ser suficiente para garantir que nunca ocorra *deadlock*. No pior caso, teremos o processo p_i responsável por pelo menos um neurônio que esteja conectado a todos os outros neurônios da rede. O processo p_i receberá, então, mensagens com os estados de todos os neurônios e, portanto, será necessário ter espaço para n - 1 estados no *buffer*.

→ Processo ***Enviar***

A única forma de p_i enviar qualquer mensagem para outros processadores, é por intermédio do *processo de comunicação*, que recebe através de um único canal, as mensagens a serem roteadas. Na estrutura de p_i, temos dois processos que enviam mensagens para os outros processadores, sendo necessário criar um processo para coletar estas mensagens e direcioná-las para o *processo de comunicação*.



Processo Enviar

fim **← falso**
Enquanto não *fim* faça
    msg recebida cujo emissor é o processo *principal* ou o processo *tratar isolado* →
        Acrescentar a msg ao pacote cujo destino é o mesmo
        Se o pacote estiver preenchido então
            enviá-lo para o *processo de comunicação*
        Fim-se

    aviso enviado pelo processo *principal* recebido →
        Enviar todos os pacotes que ainda não estão completos para o *processo de comunicação*

    msg de finalização recebida →
        fim **← verdade**
Fim-enq

Fig. 2.11 - Algoritmo do processo *enviar*

Além disto, o processo *enviar* agrupa as mensagens em **pacotes** com o objetivo de diminuir o seu número. Quando dois processadores trocam mensagens, há uma sincronização entre eles. O tempo utilizado para se alcançar esta sincronização, usualmente, é muito maior que o tempo de transmissão da informação. Assim, um número menor de mensagens sendo elas maiores, gera um decréscimo no custo de comunicação. Por outro lado, quando não enviamos o valor de um estado logo que este é atualizado, estamos atrasando a atualização dos neurônios dependentes deste novo valor. Em vista disto, realizamos um balanceamento para definir o número máximo de estados a serem empacotados de forma a não atrasar o processamento global do sistema. Este balanceamento foi realizado experimentalmente levando-se em conta o fator memória. Quanto maior o tamanho da mensagem, mais memória é requisitada pois cada processo que a recebe tem de armazená-la em algum espaço. Tivemos também o cuidado de, depois de p_i atualizar um subgrafo de neurônios, enviar os pacotes armazenados mesmo não estando totalmente preenchidos, com o objetivo de completar a atualização da(s) parte(s) do subgrafo que não esteja(m) em p_i. O algoritmo de *enviar* aparece na figura 2.11.

➡ Processo ***Tratar Isolado***

Definimos um subgrafo isolado como aquele cujos nós estão alocados em um único processador. Sendo assim, com o objetivo de aproveitar esta independência, criamos o processo ***Tratar Isolado***, que não depende de nenhum dado ou mensagem externa e que é responsável pela evolução dos subgrafos isolados até estes se estabilizarem. Deve-se notar que o algoritmo deste processo corresponde basicamente ao algoritmo seqüencial do modelo. A diferença mais marcante entre eles está relacionada com a forma como os neurônios são agrupados. Enquanto o algoritmo seqüencial visualiza o neurônio como parte de um grafo ou rede de neurônios, o processo *tratar isolado* vê o neurônio como parte de um subgrafo e cada subgrafo é tratado separadamente. Assim, se tivermos mais de um subgrafo isolado no mesmo processador, estes podem se estabilizar em momentos distintos, não necessariamente ao mesmo tempo visto que são independentes. O algoritmo de *tratar isolado* se apresenta na figura 2.12.

Processo Tratar Isolado

Esperar por uma msg enviada pelo processo *principal* indicando que todos os dados de entrada já foram recebidos.

```
Se existir algum subgrafo isolado em p_i faça
    fim ← falso
    Enquanto não fim faça
        Para cada subgrafo isolado j não estável faça
            Para cada neurônio k pertencente ao subgrafo j faça
                Calcular o potencial de entrada P_k de acordo com a
                equação 2.1.

                Atualizar o neurônio k de acordo com a equação 2.2.
            Fim-para

            Verificar se o subgrafo j estabilizou comparando seu novo
            estado com o anterior.
        Fim-para

        Se todos os subgrafos isolados estiverem estáveis então
            fim ← verdade
        Fim-se
    Fim-enq

    Para cada neurônio k pertencente a um subgrafo isolado faça
        Enviar o estado final de k para o processo enviar com destino p_M
        no processador 0.
    Fim-para
Fim-se
```

Fig. 2.12 - Algoritmo do processo *tratar isolado*

→ Processo *Principal*

O processo ***Principal*** é o responsável por todos os procedimentos que envolvem a atualização dos neurônios. Assim como no algoritmo original, utilizaremos o protocolo de escalonamento por reversão de arestas, sendo então necessário estabelecer uma orientação inicial acíclica do grafo G formado por todos os neurônios do sistema. Teremos dois tipo de ligação (exemplificadas na figura 2.13): **ligação externa**, que interliga dois neurônios alocados em processadores distintos; e **ligação interna**, onde os dois neurônios das extremidades da ligação pertencem ao mesmo processador.

A orientação acíclica inicial é formada através do seguinte direcionamento das ligações externas: dado $N_j \in p_i$ e $N_k \in p_l$ onde $i \neq l$ e $w_{jk} \neq 0$, a orientação da ligação será

de N_j para N_k se e somente se $i < l$. Neste caso denominaremos N_j vizinho *upstream* de N_k que, por sua vez, será vizinho *downstream* de N_j. As ligações internas não terão nenhum tipo de orientação.

Cada processador só trocará mensagens com os processadores que tiverem alguma ligação externa com este. O processo p_i, então, só receberá os estados referentes aos neurônios com os quais tem ligações externas. As ligações internas serão representadas pela matriz de pesos w e não será necessário haver trocas de mensagens internamente pois utilizaremos estruturas de dados e as trocas de mensagens serão simuladas através de manipulações sobre estas estruturas.

Em cada processador teremos partes de subgrafos sendo tratadas como blocos separados (ilustrado na figura 2.13). O processo *principal* interno a um processo p_i atualiza todos os neurônios de um subgrafo por vez. Um subgrafo j está pronto para atualização, quando o processo tiver recebido os estados de todos os N_k tal que $N_k \notin p_i$ e tenha ligação externa com algum neurônio interno pertencente ao subgrafo j. Depois de realizada a atualização, os novos estados são enviados para todos os processadores que tenham alguma ligação externa com os neurônios atualizados e para o processo p_M. Este envio de mensagens indica a reversão das arestas que correspondem às ligações externas.

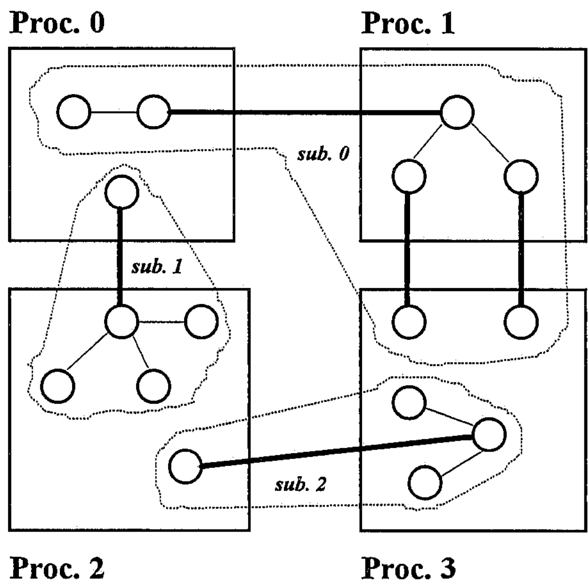


Fig. 2.13 - Exemplo da divisão dos subgrafos por um sistema de 4 processadores. As linhas mais grossas representam as ligações externas e as outras as ligações internas.

Fazendo uma analogia com o algoritmo original, podemos dizer que um subgrafo j só pode ser atualizado quando se tornar um "*sink*". Desta forma, definimos os vizinhos do subgrafo j como sendo os neurônios dos outros processadores que tenham ligação externa com algum neurônio de j. Algumas destas ligações podem ser condensadas em uma única. Se por acaso um neurônio N_k de um processo p_l está conectado a mais de um neurônio do subgrafo j no processo p_i, quando N_k for atualizado, p_l não precisa enviar o seu estado para cada um dos neurônios vizinhos em p_i. É suficiente que p_l envie uma única mensagem destinada ao subgrafo j de p_i.

Na descrição do algoritmo do processo *principal* utilizamos as seguintes variáveis:

- *flushed* - contém o número de mensagens STOP's recebidas.
- $n.viz_j$ - contém o número de ligações externas do subgrafo j.
- $n.up_j$ - contém o número de vizinhos *upstreams* do subgrafo j.
- $estabilizou_j$ - indica se o subgrafo j já estabilizou.

Os passos de inicialização do processo *principal* de p_i são os seguintes:

1. Receber os valores de w_{kl} tal que N_k e/ou N_l estão alocados em p_i e $w_{kl} \neq 0$, os valores iniciais dos estados dos neurônios (V^{inic}), e a separação dos neurônios em subgrafos.
2. Para cada subgrafo j contar o número de ligações externas que seus neurônios possuem, armazenando em $n.viz_j$.
3. Para cada subgrafo j contar dentre as ligações externas quais que representam vizinhos *upstreams*.
4. Enviar um aviso de início para o processo *tratar isolado* indicando que os dados de entrada já foram recebidos.

A seguir temos a descrição do corpo do processo *principal*:

1. Para cada subgrafo j verificar se este é um *sink*. Caso afirmativo, executar o procedimento **operar**.
2. Sincronizar com o processo *buffer de entrada* através do canal de comunicação *get*, para retirar uma mensagem do *buffer*.
3. Caso a mensagem seja o estado de um neurônio N_k, armazenar o valor recebido em V_k e executar o procedimento **operar** (apresentado na figura 2.14). Caso a mensagem seja um STOP.PARCIAL referente a um subgrafo j, atribuir o valor verdade para $estabilizou_j$. Caso a mensagem seja um STOP, incrementar *flushed* de um. Se *flushed*

for igual a um, enviar uma mensagem de STOP para todos os processos p_l tal que $0 \leq l \leq 7$ e $l \neq i$. Se *flushed* for igual ao número de processadores do sistema mais um, mandar uma mensagem de STOP para p_M, um aviso de finalização para os processos *enviar* e *buffer de entrada*, e terminar o processamento. Caso contrário, retornar ao passo 2.

Procedimento Operar

Identificar o subgrafo *j* que tem ligação externa com N_k (neurônio cujo estado foi recebido)
Se não *estabilizou*$_j$ então
    $n.up_j \leftarrow n.up_j + 1$
    Se $n.up_j = n.viz_j$ (significando que o subgrafo *j* se tornou *sink*) então
        $n.up_j \leftarrow 0$
        Para cada neurônio $N_l \in$subgrafo *j* faça
            Calcular o potencial de entrada P_l de acordo com a equação 2.1.

            Atualizar V_l de acordo com a equação 2.2.

            Enviar V_l para o processo *enviar* com destino p_n para todo $0 \leq l \leq 7$ tal que $n \neq i$ e p_n contenha algum neurônio conectado a N_l e enviar para p_M
        Fim-para

        Enviar um sinal para o processo *enviar* com o objetivo de enviar todos os pacotes ainda armazenados, mesmo que não estejam completos.
    Fim-se
Fim-se

Fig. 2.14 - Algoritmo do procedimento operar

Não utilizamos a definição dos estados ativo e inativo introduzidos no algoritmo original, porque inicialmente todos os processos já estão ativos pois tomam conhecimento do estado de todos os seus vizinhos *upstream* ao receber os estados iniciais de todos os neurônios.

2.2.1.3.3. Segunda Versão da Implementação

Durante o desenvolvimento do simulador, tivemos que levar em consideração a densidade de conexão do grafo *G* formado pela rede de neurônios. Sua primeira versão

demonstra um bom desempenho quando temos um grafo de baixa densidade. Entretanto, no caso de um grafo fortemente conectado, o paralelismo não é aproveitado. O grafo seria constituído por um único bloco, não sendo dividido em subgrafos. De acordo com o algoritmo, a cada instante teríamos apenas um processador atualizando seus neurônios enquanto os outros esperam por estes novos valores. O motivo desta seqüencialidade é devido à orientação inicial do grafo que forma apenas um *sink* e durante a evolução do algoritmo sempre teremos apenas um *sink* por vez.

A segunda versão foi desenvolvida com o objetivo de melhorar o desempenho do simulador perante uma rede de neurônios fortemente conectada. A única diferença entre as duas versões corresponde ao processo *principal*.

Em um grafo fortemente conexo, teremos poucos ou apenas um subgrafo, de forma que agora os subgrafos não serão tratados como blocos separados, cada neurônio será tratado individualmente. A orientação acíclica inicial do grafo G é estabelecida pelo conjunto de regras correspondentes aos dois tipos de ligações:

- ligações internas: dado N_j, $N_k \in p_i$ onde $w_{jk} \neq 0$, a orientação da ligação será de N_j para N_k se e somente se $j < k$.
- ligações externas: dado $N_j \in p_i$ e $N_k \in p_l$ onde $i \neq l$ e $w_{jk} \neq 0$, a orientação da ligação será de N_j para N_k se e somente se $i < l$.

Nos dois casos, denominamos N_j de vizinho *upstream* de N_k, que, por sua vez, é vizinho *downstream* de N_j. Empregaremos a definição de neurônio *sink* ao invés de subgrafo *sink*. Cada vez que se forma um *sink*, o valor do respectivo neurônio é atualizado e enviado para todos os processadores que possuam algum neurônio vizinho a este, revertendo assim as arestas correspondentes às ligações externas. Também é necessário reverter as arestas internas possibilitando a geração de novos *sinks*. O procedimento de reversão das arestas internas atualizando-se os *sinks* formados, é projetado com o uso da recursão, visto que é realizado um encaminhamento pelo grafo com o intuito de acessar todos os vizinhos ligados direta e indiretamente ao primeiro neurônio que se tornou *sink*. Denominamos este procedimento de **operar**, mostrado na figura 2.15.

O processo *principal* utiliza as seguintes variáveis:

- *flushed* - número de mensagens STOP's recebidas
- $n.viz_j$ - número de vizinhos N_k do neurônio N_j, tal que $w_{jk} \neq 0$
- $n.up_j$ - número de vizinhos *upstreams* do neurônio N_j
- $estabilizou_k$ - indica se o subgrafo k estabilizou

```
Procedimento Operar (N_k)

  Para cada neurônio N_j∈p_i vizinho de N_k faça
     Identificar o subgrafo l do qual N_j faz parte
     Se não estabilizou_l então
        n.up_j ← n.up_j + 1
        Se n.up_j = n.viz_j então
           n.up_j ← 0
           Calcular o potencial de entrada P_j de acordo com a
           equação 2.1

           Atualizar V_j de acordo com a equação 2.2

           Enviar o novo valor V_j para todos os processadores
           que contenham pelo menos um vizinho de N_j

           Operar (N_j)
        Fim-se
     Fim-se
  Fim-para
```

Fig. 2.15 - Algoritmo do procedimento operar

Inicialmente o processo *principal* estruturado em p_i, executa as seguintes etapas:

1. Receber os valores de w_{jk} tal que N_j e/ou N_k estão alocados em p_i e $w_{jk} \neq 0$, os valores iniciais dos estados dos neurônios (V^{inic}), e a separação dos neurônios em subgrafos.

2. Para cada neurônio $N_j \in p_i$, contar o número de vizinhos N_k tal que $w_{jk} \neq 0$, armazenando-o em $n.viz_j$.

3. Para cada neurônio $N_j \in p_i$, contar o número de vizinhos *upstreams* de N_j, armazenando-o em $n.up_j$.

4. Enviar um aviso de início para o processo *tratar isolado* indicando que os dados de entrada já foram recebidos.

Depois da inicialização, o processo *principal* segue os seguintes passos:

1. Para cada neurônio N_j *sink*, executar o procedimento **operar**.

2. Sincronizar com o processo *buffer de entrada* através do canal de comunicação *get*, para retirar uma mensagem do *buffer*.

3. Caso a mensagem seja o estado de um neurônio N_k, armazenar o valor recebido em V_k e executar o procedimento **operar** (apresentado na figura 2.15). Caso a mensagem seja um STOP.PARCIAL referente a um subgrafo l, atribuir o valor verdade para *estabilizou*$_l$. Caso a mensagem seja um STOP, incrementar *flushed* de um. Se *flushed* for igual a um, enviar uma mensagem de STOP para todos os processos p_r tal que $0 \le r \le 7$ e $r \ne i$. Se *flushed* for igual ao número de processadores do sistema mais um, mandar uma mensagem de STOP para p_M, um aviso de finalização para os processos *enviar* e *buffer de entrada*, e terminar o processamento. Caso contrário, retornar ao passo 2.

2.2.2. Máquina de Boltzmann

Um problema de Otimização Combinatória é comumente apresentado da seguinte forma: seja $\mathbf{X} = \{X_1, ..., X_n\}$ o conjunto de n variáveis que podem assumir os valores de um conjunto domínio finito $\mathbf{D}$, e $\mathbf{F}$ o conjunto de pontos viáveis (*feasible points*) em $\mathbf{D}^n$, o problema se resume em encontrar o mínimo global de uma função f cuja forma é:

$$f: \mathbf{D}^n \rightarrow \Re \tag{5}$$

isto é, deseja-se encontrar um ponto $x^* \in \mathbf{F}$ tal que $f(x^*) \le f(x)$ para todo $x \in \mathbf{F}$.

Os problemas de Otimização Combinatória são usualmente de difícil resolução, porque a maioria dos métodos iterativos, em algum momento, "encalham" em um mínimo local, que muitas vezes se localiza em um ponto bem distante do mínimo global.

Os **problemas de decisão** foram utilizados como base para o estudo da dificuldade de resolução de problemas e eficiência dos algoritmos. Um problema de Otimização pode ser formulado como um de decisão gerando o problema de se verificar a existência de $x \in \mathbf{F}$ tal que $f(x) \le B$ para $B \in \Re$. Definimos então que um problema de decisão pertence à classe **NP** (*nondeterministic polynomial*) se, dado $x \in \mathbf{D}^n$, é possível verificar em tempo polinomial se $x \in \mathbf{F}$ e se $f(x) \le B$. A classe NP é constituída de duas subclasses: a classe **P** e a classe **NP-completo**. Os problemas de decisão da classe P podem ser resolvidos em tempo polinomial. Os problemas que são NP-completos constituem os de maior dificuldade de resolução no sentido de que a prova da existência de um algoritmo que solucione um destes problemas em tempo polinomial, automaticamente prova que todos os problemas em NP podem ser solucionados neste tempo. Até agora esta prova não foi alcançada. Só foram desenvolvidos algoritmos com tempo exponencial como solução para os problemas NP-completos, tornando-os potencialmente intratáveis.

A maioria dos problemas de Otimização Combinatória são NP-completos, motivando o estudo de várias técnicas de aproximação que tentam evitar os mínimos locais de f. A Máquina de Boltzmann é uma destas técnicas e a versão que

apresentaremos é baseada no estudo do método de *Simulated Annealing*, descrito a seguir.

2.2.2.1. Técnica de *Simulated Annealing*

Simulated Annealing pode ser aplicado como uma técnica de aproximação do problema de se encontrar um mínimo global de uma função f em **F**. O método é baseado na observação física de que um gás só consegue atingir seu estado mínimo de energia quando resfriado de forma suficientemente lenta, principalmente nas temperaturas mais baixas, a partir de uma temperatura inicial relativamente alta. O material congelado através deste método é constituído por uma estrutura perfeita que não é alcançada sob outras condições que ocasionam a cristalização do material em um estado de energia mínima "local". Este processo é identificado na Física como *annealing*.

Utilizamos também a técnica de simulação Metropolis [8] que se constitui de uma seqüência de passos, onde em cada um desloca-se uma partícula calculando a respectiva variação de energia ΔE do sistema. Este deslocamento é aceito com uma probabilidade 1 se $\Delta E \leq 0$ e com probabilidade

$$e^{-\Delta E / k_B T} \qquad (6)$$

caso contrário, onde T é uma temperatura fixa e k_B é a constante de Boltzmann, caracterizando uma distribuição de Boltzmann.

A técnica de *Simulated Annealing* se constitui em visualizar cada variável em **X** como uma partícula de acordo com a observação física mencionada, e utilizar os cálculos de Metropolis em uma seqüência decrescente de pequenas variações de temperaturas. A função de energia E corresponde à função objetivo f e para cada temperatura desta seqüência, executa-se os passos da simulação Metropolis até se atingir um equilíbrio. Esta condição de equilíbrio é determinada pelo problema a ser solucionado. Se a seqüência de temperaturas é iniciada por um valor relativamente alto sendo decrementado gradualmente, é esperado que no final as variáveis em **X** atinjam valores caracterizando um estado de energia globalmente mínimo. Nesta aplicação, ignoraremos a constante de Boltzmann, tendo então, como probabilidade de aceitação de uma configuração de energia mais alta

$$e^{-\Delta E / T} \qquad (7)$$

Pela descrição do método, nota-se que a granularidade da busca pelo ponto mínimo varia a medida que T se aproxima do seu valor final. Inicialmente T assume valores altos, caracterizando a busca feita sobre **F** como de granularidade grossa (*coarse-grained*), pois praticamente todo deslocamento é aceito uma vez que a probabilidade da equação (7) se torna próxima de 1. Por outro lado, a medida que T diminui, o deslocamento para pontos

de maior energia se torna menos provável, caracterizando uma busca de granularidade fina (*fine-grained*).

A forma com que o método de *Simulated Annealing* foi descrito demonstra uma impossibilidade de implementação paralela em um sistema distribuído. Devido à necessidade de se ter um conhecimento global da configuração das variáveis para a escolha de uma nova configuração, e à existência da restrição de que a temperatura só pode ser decrescida quando o sistema atingir um estado de equilíbrio, nos deparamos com a necessidade de alterar o método com o objetivo de sanar estes dois problemas.

Inicialmente, definimos que duas variáveis em **X** são vizinhas ou não de acordo com algum critério declarado pelo problema a ser resolvido. No contexto de uma busca estocástica sobre $\mathbf{D}^n$, o conjunto de variáveis X forma um **Campo Aleatório de Markov** (*Markov Random Field*) se puder ser visto como um conjunto de variáveis aleatórias e satisfizer as seguintes propriedades:

$$\begin{aligned}&\text{Prob}(x) > 0, \text{ para todo } x \in \mathbf{D}^n;\\&\text{Prob}(X_i = x_i \mid X_j = x_j, j \neq i) = \text{Prob}(X_i = x_i \mid X_j = x_j, X_j \text{ é vizinho de } X_i),\end{aligned} \quad (8)$$

para todo $X_i \in \mathbf{X}$. Estas propriedades expressam o requerimento de que a probabilidade com a qual X_i assume um valor, depende apenas dos valores das variáveis X_j vizinhas de X_i.

Um Campo Aleatório de Markov possui a propriedade de que seja **C** o conjunto de todos os **cliques** em **X**, onde um clique é definido como um subconjunto $\mathbf{W} \subseteq \mathbf{X}$ tal que todas as variáveis em **W** são vizinhas entre si, as propriedades em (8) valem para o conjunto **X** se e somente se existir uma função $u : \mathbf{D}^n \rightarrow \Re$ da forma

$$u(x) = \sum_{\mathrm{W} \in \mathrm{C}} v_{\mathrm{W}}(x) \quad \mathbf{(9)}$$

onde $v_{\mathrm{W}}(x)$ depende apenas das variáveis em **W** e $\text{Prob}(x) = \pi(x)$ para todo $x \in \mathbf{D}^n$. $\pi(x)$ representa a distribuição de Gibbs dada por

$$\pi(x) = \frac{1}{Z} e^{-u(x)/T} \quad \mathbf{(10)}$$

onde Z é uma constante de normalização e T é um parâmetro de controle semelhante à temperatura na distribuição de Boltzmann.

Tendo em vista estas definições, geramos uma variação do método de *Simulated Annealing* que minimiza funções f com formas semelhantes à de u. Teremos então que formular a função f da seguinte forma

$$f(x) = \sum_{Y \subseteq X} g_Y(x) \tag{11}$$

onde $g_Y(x)$ depende apenas das variáveis em **Y**. Definindo que duas variáveis são vizinhas se e somente se pertencem a um subconjunto $\mathbf{Y} \subseteq \mathbf{X}$ tal que $g_Y(x)$ não é constante, o conjunto **X** pode ser caracterizado como um Campo Aleatório de Markov dado que cada um dos subconjuntos **Y** é um clique. O conjunto **X** segue então a seguinte distribuição

$$\mathbf{Prob}(x) = \frac{1}{Z} e^{-f(x)/T} \tag{12}$$

A simulação do Campo Aleatório de Markov para um valor fixo de T, é realizada através da atualização das variáveis em **X** de acordo com as probabilidades condicionais em (8), utilizando a distribuição em (12). Ao iniciarmos T com um valor relativamente alto, decrementando-o gradativamente ao longo da simulação, alcança-se um ponto $x \in \mathbf{D}^n$ em que f é globalmente mínimo.

2.2.2.2. O Modelo da Máquina de Boltzmann

A Máquina de Boltzmann surgiu a partir de uma combinação da variação do método de *Simulated Annealing*, descrito na seção anterior, com o modelo de Rede Neuronal Binária. Associamos cada variável $X_1, \ldots, X_n$ a cada estado do neurônio $V_1, \ldots, V_n$. Dado $\mathbf{D} = \{0, 1\}$, devemos notar que a função de energia E, mostrada em (3) na seção 2.2.1.1, tem o formato de $f(x)$ em (11). Em adição, definimos que X_i e X_j são vizinhas se $w_{ij} \neq 0$ (w representa a matriz de pesos das sinapses da rede neuronal).

A partir destas associações, utilizamos a técnica de *Simulated Annealing* para encontrar um mínimo global de E. Logo, os estados dos neurônios são atualizados por meio das probabilidades condicionais em (8) decrescendo-se o parâmetro T. Neste caso em particular, as probabilidades se tornam

$$\begin{aligned} \text{Prob}(V_i = 1 \mid V_j = v_j, j \neq i) &= \frac{e^{-\Delta E/T}}{1 + e^{-\Delta E/T}} \\ \text{Prob}(V_i = 0 \mid V_j = v_j, j \neq i) &= \frac{1}{1 + e^{-\Delta E/T}} \end{aligned} \tag{13}$$

onde $v_j \in \{0, 1\}$. Podemos definir ΔE tendo em vista os valores de E quando $V_i = 0$ e quando $V_i = 1$. Denominaremos estes valores de E_0 e E_1 respectivamente. Temos então que

$$\Delta E = E_1 - E_0$$
$$\Delta E = -\sum_{j \neq i} w_{ij} \, V_j - I_i + \theta_i \qquad (14)$$

Uma das vantagens da Máquina de Boltzmann sobre a Rede Neuronal é que a primeira busca encontrar um mínimo global de E, enquanto que a Rede Neuronal se satisfaz com um mínimo local.

O algoritmo seqüencial deste modelo referente a um problema genérico, se encontra na figura 2.16. Deve-se notar que a escolha dos seguintes parâmetros dependem do problema em questão: as temperaturas inicial e final (T_0 e T_{final} respectivamente), os valores iniciais V^{inic} dos neurônios, e a escolha da função r responsável pela redução da temperatura.

```
Seja T0 a temperatura inicial e Tfinal a temperatura mínima final
Seja Vinic ∈ Dn os valores iniciais dos neurônios do sistema
T ← T0
V ← Vinic
Enquanto T ≥ Tfinal faça
    Para cada neurônio Ni faça
        Calcular as probabilidades condicionais em (13)
        Atualizar Vi de acordo com estas probabilidades
    Fim-para
    T ← r (T)
Fim-enq
```

Fig. 2.16 - Algoritmo do simulador seqüencial da Máquina de Boltzmann.

2.2.2.3. Simulação Paralela Distribuída

Primeiramente, apresentaremos o algoritmo distribuído em que se baseia a implementação do simulador. Associaremos a cada neurônio N_i um processo p_i, cuja tarefa é a de atualizar o estado do neurônio durante a simulação. Cada par de processos p_i e p_j são interconectados por um canal de comunicação bidirecional se e somente se N_i e N_j forem vizinhos entre si. Assim como na Rede Neuronal Binária de Hopfield, dois neurônios só podem ser atualizados concorrentemente se não forem vizinhos. Em outras palavras, dado os neurônios N_i e N_j tal que $1 \leq i, j \leq n$ e $i \neq j$, a condição para que eles possam ser atualizados concorrentemente é a de que $w_{ij} = 0$.

Este sistema de processos é assíncrono pois cada processo possui a sua própria base de tempo e as mensagens enviadas pelos canais podem sofrer atrasos que, apesar de durarem um tempo finito, são imprevisíveis. Representaremos o sistema através de um grafo não-direcionado $G = (\mathbf{P}, \mathbf{C})$ onde o conjunto de nós $\mathbf{P}$ é o conjunto de processos e

o conjunto de arestas **C** representa o conjunto de canais de comunicação. A restrição de que dois neurônios vizinhos não podem ser atualizados ao mesmo tempo, pode ser vista da seguinte forma: dado um estado global do sistema, o conjunto de processos que estão atualizando seus neurônios deve constituir um conjunto independente em *G*, isto é, um subconjunto de nós onde nenhum par de nós pode estar conectado por arestas. Para que esta condição seja garantida utilizamos o protocolo de escalonamento por reversão de arestas descrito na seção 2.1.

Inicialmente estabelecemos uma orientação acíclica para o grafo *G* de forma que dado p_i e p_j tal que $i \neq j$ e $w_{ij} \neq 0$, direcionaremos a aresta de p_i para p_j se e somente se $i < j$. Em seguida, cada processo envia o estado inicial de seu neurônio através das arestas direcionadas para "fora". Cada nó *sink*, depois de receber os valores iniciais de seus vizinhos, atualiza seus neurônios de acordo com as probabilidades condicionais em (13), e envia seu novo valor para seus vizinhos revertendo assim suas arestas incidentes.

Seja V_i^{inic} o valor inicial do neurônio N_i.
Seja T_0 a temperatura inicial e T_{final} a temperatura final.
$V_i \leftarrow V_i^{inic}$
$T \leftarrow T_0$
Marcar os canais orientados na direção de p_i.
Enviar o valor de V_i através dos canais de comunicação orientados para "fora".
Esperar mensagens de cada processo vizinho p_j tal que os canais estejam orientados de p_j para p_i, armazenando-as em V_j.
Enquanto $T \geq T_{final}$ faça
    Enquanto p_i não for *sink* faça
        Receber mensagem enviada por qualquer vizinho p_j armazenando-a em V_j.
        Direcionar o canal por onde a mensagem foi transmitida, em direção a p_i.
    Fim-enq
    Calcular as probabilidades condicionais em (13).
    Atualizar V_i de acordo com estas probabilidades.
    Reverter as arestas incidentes enviando V_i para todos os processos vizinhos.
    $T \leftarrow r(T)$
Fim-enq
Esperar por mensagens transmitidas pelos canais marcados ignorando-as.

Fig. 2.17 - Algoritmo original de um processo p_i do simulador da Máquina de Boltzmann

Na figura 2.17 temos o algoritmo executado pelo processo p_i depois de estabelecida a orientação inicial de G. É importante notar que a seqüência de temperaturas $T_0, \ldots, T_{final}$ é a mesma para todos os processos, demonstrando que todos os neurônios são atualizados o mesmo número de vezes não havendo necessidade de teste de estabilidade do sistema. Quando todos os neurônios tiverem alcançado uma temperatura menor ou igual à final, o sistema pode ser terminado. O objetivo de se marcar os canais que inicialmente são direcionados para "dentro" e esperar o recebimento de mensagens por estes canais antes de terminar a execução de p_i, é o de garantir a não existência de mensagens em trânsito. Isto está detalhado no algoritmo.

Na implementação distribuída deste modelo não associamos um processo a cada neurônio como descrito acima, pois temos disponíveis apenas 8 processadores e a criação de vários processos em um mesmo processador onera muito o custo do sistema referente ao tempo, devido à troca desnecessária de mensagens internamente. Tendo isto em mente, alocamos vários neurônios em cada processo.

De agora em diante, ao nos referirmos ao grafo G estamos referenciando o grafo formado pela rede de neurônios e não àquele formado pelo sistema de processos. Inicialmente G é orientado aciclicamente através do direcionamento das ligações internas e externas (definidas na seção 2.2.1.3.3). A direção de uma ligação externa que interliga os processadores i e j são orientados de i para j se e somente se $i < j$. E o direcionamento de uma ligação interna entre os neurônios N_i e N_j é de N_i para N_j se e somente se $i < j$. A partir destes direcionamentos temos a denominação de vizinho *upstream* e *downstream*, definidos anteriormente. Um neurônio é dito *sink* quando todas as suas ligações estiverem orientadas na sua direção. A referência a vizinho externo diz respeito a um vizinho de um neurônio que não esteja alocado no mesmo processador. Um processo ao identificar que contém um neurônio *sink*, atualiza-o e reverte as ligações internas verificando se não foram formados novos *sinks*. Esta verificação é feita em cadeia, pois para cada novo *sink* atualizado, é necessário verificar os seus vizinhos. Por este motivo, utilizamos um procedimento recursivo de encaminhamento em grafo. Com relação às ligações externas, o novo estado é enviado para os processadores com os quais o neurônio em questão tem ligação externa, revertendo assim a direção destas.

Devido ao assincronismo do sistema, cada neurônio possui a sua temperatura corrente. O sistema então se estabiliza quando todos os neurônios atingem a temperatura final. Isto só é possível porque a seqüência de temperaturas gerada durante a simulação, é igual para todos os neurônios.

Apesar de não haver necessidade de se verificar a convergência do sistema, precisamos ter um *processo mestre* (p_M) cuja tarefa é distribuir os dados iniciais. O processador 0 é composto pelos processo p_M, p_0, *interface de entrada*, *interface de saída* e *processo de comunicação*, de acordo com a estrutura da figura 2.4 (na seção 2.2.1.3.2). E na figura 2.5 temos a estrutura dos outros processadores.

O algoritmo de p_M se constitui simplesmente de enviar os valores iniciais dos neurônios para os processadores onde foram alocados. Assim como, enviar a submatriz de pesos w referente às vizinhanças dos neurônios de cada processador. O processo mestre também é responsável pela detecção da existência de subgrafos isolados, utilizando o procedimento descrito na figura 2.6 que identifica os subgrafos que formam o grafo G. Além disso, p_M coleta os valores finais dos neurônios ao término da simulação.

Receber os valores de w_{kl} tal que N_k e/ou N_l estão alocados em p_i e $w_{kl} \neq 0$, e os valores iniciais dos estados dos neurônios (V^{inic}).
Contar o número de neurônios vizinhos que cada neurônio N_j tem armazenando-o em *n.viz*$_j$.
Contar o número de vizinhos *upstreams* de cada neurônio N_j marcando o respectivo canal, e armazenando este número em *n.up*$_j$ e em *n.marca*$_j$.
Enviar um aviso de início para o processo *tratar isolado* indicando que os dados de entrada já foram recebidos.
***fim* ← falso**
Para cada neurônio N_j alocado em p_i e que não pertença a nenhum subgrafo isolado faça
    $T_j \leftarrow T_0$
    Se N_j é um *sink* então
        Executar o procedimento operar (*j*, *fim*)
    Fim-se
Fim-para
Enquanto não *fim* faça
    Retirar uma msg do *buffer* através do canal *get*.
    Seja N_k o neurônio cujo estado está armazenado na msg em questão.
    Armazenar o estado de N_k em V_k.
    Para cada neurônio $N_j \in p_i$ e vizinho de N_k faça
        Operar (*j*, *fim*)
    Fim-para
Fim-enq
Enviar os estados finais de todos os neurônios que não pertencem a subgrafos isolados para o processo *enviar* com destino p_M.
Enviar um aviso de finalização para os processos *buffer de entrada* e *enviar*.

Fig. 2.18 - Algoritmo do processo *principal*

A estrutura de p_i é a mesma do simulador de Redes Neuronais Binária de Hopfield descrita na seção 2.2.1.3.2. Esta estrutura é ilustrada na figura 2.7 sendo composta pelos processos *principal*, *tratar isolado*, *buffer de entrada* e *enviar*. Os únicos processos com algoritmos diferentes são os processos *principal* e *tratar isolado*. O primeiro é

responsável pela detecção de *sinks* e atualização destes e o outro atualiza os neurônios componentes de subgrafos isolados. Seus respectivos algoritmos se encontram nas figuras 2.18 e 2.20. O processo *principal* faz referência ao procedimento **operar** que está representado na figura 2.19.

```
Procedimento Operar (k, fim)

  Para cada neurônio Nj∈pi vizinho de Nk faça
      Se Tj > Tfinal então
          n.upj ← n.upj + 1
          Se n.upj = n.vizj então
              n.upj ← 0
              Calcular as probabilidades condicionais de acordo com a
              equação 2.14.
              Atualizar Vj a partir destas probabilidades.
              Enviar o novo valor Vj para o processo enviar com destino
              a todos os processadores que contenham pelo menos um
              vizinho de Nj.
              Tj ← r (Tj)
              fim ← verdade
              Para cada neurônio Nl∈pi faça
                  Se (Tl > Tfinal) ou (n.marcal > 0) então
                      fim ← falso
                  Fim-se
              Fim-para
              Se não fim então
                  Operar (Nj)
              Fim-se
          Fim-se

      Senão
          Se o canal por onde a msg foi transmitida está marcado então
              n.marcaj ← n.marcaj - 1
              fim ← verdade
              Para cada neurônio Nl ∈ pi faça
                  Se (Tl > Tfinal) ou (n.marcal > 0) então
                      fim ← falso
                  Fim-se
              Fim-para
          Fim-se
      Fim-se
  Fim-para
```

Fig. 2.19 - Algoritmo do procedimento Operar

Esperar pelo aviso mandado pelo processo *principal* indicando que os dados de entrada já foram recebidos.

Se existir algum subgrafo isolado então
    $T \leftarrow T_0$
    Enquanto $T > T_{final}$ **faça**
        Para cada neurônio N_j **pertencente a um subgrafo isolado faça**
            Calcular as probabilidades condicionais da equação 2.14
            Atualizar V_j **de acordo com estas probabilidades**
        Fim-para
        $T \leftarrow r(T)$
    Fim-enq
    Enviar os estados finais de todos os neurônios pertencentes a qualquer subgrafo isolado para o processo *enviar* com destino p_M
Fim-se

Fig. 2.20 - Algoritmo do processo *tratar isolado*

2.3. Rede Bayesiana

2.3.1. Descrição do Modelo

As teorias de probabilidades tradicionais sempre utilizam a definição de uma função de distribuição de união $P(x_1, \ldots, x_n)$ de todas as proposições e suas combinações, como forma de representação de conhecimento probabilístico. Esta função é utilizada como base primária para todas as inferências. A representação numérica apresenta vários problemas. Por exemplo, considere o problema de se codificar uma distribuição de união arbitrária $P(x_1, \ldots, x_n)$ para um conjunto de n variáveis proposicionais $\mathbf{X} = \{X_1, \ldots, X_n\}$. A armazenagem explícita desta função requere uma tabela com 2^n entradas, que é um número significativamente grande para qualquer padrão.

Outro problema gerado pelas representações puramente matemáticas de informação probabilística, é sua falta de significado psicológico. A representação numérica pode gerar medidas probabilísticas coerentes para todas as sentenças proposicionais, mas isto é feito através de computações que a razão humana não utilizaria. Desta forma, o processo que nos conduz das premissas até as conclusões, não podem ser seguidas, testadas ou justificadas pelos usuários.

Uma das mais fortes inadequações das teorias probabilísticas tradicionais, está na forma de determinação da noção de independência. Esta é definida através do uso da igualdade de quantidades numéricas como em $P(x, y) = P(x)\, P(y)$, sugerindo que deve-se testar se a distribuição de união de X e Y é igual ao produto das suas marginais com o objetivo de determinar se X e Y são independentes. Em contraste, as pessoas podem

facilmente detectar dependências mesmo sem o conhecimento de como avaliar estimativas numéricas precisas das probabilidades. As pessoas tendem a julgar a relação de dependência condicional da relação entre três elementos (isto é, *X* influencia *Y*, dado *Z*) com clareza, convicção e consistência, utilizando o "bom senso". Logo, um sistema racional com bom senso deve empregar uma linguagem para representar informação probabilística que permita expressar qualitativa, direta e explicitamente asserções referentes às relações de dependência.

Uma lógica de dependência pode ser útil na verificação da consistência de um conjunto de dependências sustentado por um agente, e da geração de uma nova dependência a partir do conjunto inicial. Entretanto, não é garantido que esta verificação seja tratável ou que qualquer seqüência de inferências correspondem aos passos mentais dados por humanos. Deve-se assegurar que a maioria das derivações na lógica correspondam a operações locais simples em estruturas. Estas estruturas denominaremos de **grafos de dependências**.

Os nós deste grafo representam as variáveis proposicionais e os arcos representam as dependências locais entre proposições conceitualmente relacionadas. As ligações do grafo nos permite expressar as relações de dependência direta e qualitativamente, e a topologia do grafo ilustra estas relações explicitamente preservando-as de qualquer atribuição de parâmetros numéricos.

Não há nenhum problema em se configurar um grafo que represente fenômenos com noções de vizinhança ou adjacência. Por outro lado, ao se modelar relações conceituais como causalidade, associação e relevância, é geralmente difícil se distingüir vizinhos diretos de indiretos, tornando esta tarefa bem delicada. Um exemplo de um problema de difícil modelagem, seria a noção de independência condicional da teoria de probabilidade. Para uma dada distribuição de probabilidade *P* e três variáveis *X*, *Y* e *Z*, pode-se facilmente checar se dado *Z* podemos dizer que *X* independe de *Y*, mas *P* não nos determina quais variáveis são definidas como vizinhas diretas. Por este motivo, pode-se utilizar várias topologias distintas para representar as dependências de *P*.

Redes Bayesianas utilizam a linguagem de **grafos direcionados**, onde as direções das setas nos permitem distingüir dependências genuínas daquelas induzidas por observações hipotéticas. A rede pode ser usada como um instrumento de inferência das dependências lógicas e funcionais. Uma das maiores vantagens deste tipo de representação é que este facilita a quantificação das ligações por meio de parâmetros locais com significados conceituais, tornando a rede, como um todo, uma base de conhecimento globalmente consistente.

Redes Bayesianas são grafos direcionados acíclicos em que os nós representam variáveis, os arcos indicam a existência de influências causais diretas entre as variáveis interligadas, e a intensidade (também chamada peso) destas influências são expressas através de probabilidades condicionais.

Utilizamos como modelo uma rede Bayesiana cujas variáveis são binárias, isto é, assumem valores 0 ou 1. Neste caso, existirá uma aresta direcionada de X_i para X_j se o evento $X_i = 1$ puder causar o evento $X_j = 1$ com alguma probabilidade não-nula. Definimos formalmente uma rede Bayesian como um grafo direcionado acíclico mínimo em que cada nó de separação estrutural (definido mais detalhadamente em [9]) tem uma independência condicional de eventos na distribuição P correspondente.

Classificamos a relação que um nó X_i pode ter com outros nós, em três tipos:

- nós **pais** - X_j tal que existe no grafo uma aresta direcionada de X_j para X_i.
- nós **filhos** - X_j tal que existe no grafo uma aresta direcionada de X_i para X_j.
- nós **companheiros** - X_j tal que existe no grafo as seguintes arestas: uma direcionada de X_i para X_k e outra de X_j para X_k, onde X_k é um nó qualquer.

Denominamos estes três conjuntos de nós de $\mathbf{P}(X_i)$, $\mathbf{F}(X_i)$ e $\mathbf{C}(X_i)$ respectivamente. Ainda temos um conjunto de evidências **E** composto pelas variáveis que se constituem entrada para a rede, tendo seus valores constantes durante a simulação.

O conjunto $\mathbf{P}(X_i)$ é definido da seguinte forma: dado uma distribuição P e uma ordem total $\prec$ de $\mathbf{X} = \{X_1, \ldots, X_n\}$, $\mathbf{P}(X_i)$ é um subconjunto $\mathbf{W} \subset \mathbf{X}$ de forma que $X_j \prec X_i$ para todo $X_j \in \mathbf{W}$ sendo $\mathbf{W}$ mínimo de acordo com a propriedade

$$P(x_i \mid x_j; X_j \prec X_i) = P(x_i \mid x_j; X_j \in \mathbf{W}) \qquad \textbf{(15)}$$

Temos então

$$P(x_1, \ldots, x_n) = \prod_i P(x_i \mid x_j; X_j \in \mathbf{P}(X_i)) \qquad \textbf{(16)}$$

Em princípio, dado qualquer distribuição $P(x_1, \ldots, x_n)$, temos um procedimento recursivo simples para a construção de uma rede Bayesiana. Dado que o conjunto **X** está ordenado da forma $\mathbf{X} = \{X_1, ..., X_n\}$, iniciamos com a escolha de X_1 como uma raiz e atribuímos a probabilidade marginal $P(x_1)$ a ela. Em seguida, geramos um nó para representar X_2. Se X_2 é dependente de X_1, cria-se uma ligação de X_1 para X_2 quantificada por $P(x_2 \mid x_1)$, definindo X_1 como nó **pai** de X_2. Caso contrário, X_1 e X_2 ficam desconectados atribuindo-se a probabilidade $P(x_2)$ ao nó X_2. Na i-ésima etapa, formamos o nó X_i, geramos um grupo de ligações orientadas de um conjunto de nós pais $\mathbf{P}(X_i)$ para X_i e quantificamos este grupo de ligações com a probabilidade condicional

$$P(x_i \mid \mathbf{P}_{X_i}) = P(x_i \mid x_1, \ldots, x_{i-1}),\ \mathbf{P}(X_i) \subset \{X_1, X_2, \ldots, X_{i-1}\} \qquad \textbf{(17)}$$

Como resultado teremos um grafo direcionado acíclico representando a maioria das independências embutidas em $P(x_1, \ldots, x_n)$.

Inversamente, as probabilidades condicionais $P(x_i \mid \mathbf{P}(X_i))$ das ligações do grafo devem conter toda informação necessária para se reconstruir a função de distribuição original. Teremos então

$$\begin{aligned} P(x_1, x_2, \ldots, x_n) &= P(x_n \mid x_{n-1}, \ldots, x_1)\, P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \ldots P(x_3 \mid x_2, x_1) \\ & \quad P(x_2 \mid x_1)\, P(x_1) \\ &= \prod_i P(x_i \mid \mathbf{P}(X_i)) \end{aligned} \tag{18}$$

Na figura 2.21 apresentamos um exemplo de um grafo direcionado acíclico cuja distribuição pode ser escrita como

$$\begin{aligned} P(x_1, x_2, x_3, x_4, x_5, x_6) &= P(x_6 \mid x_3, x_5)\, P(x_5 \mid x_2, x_3)\, P(x_4 \mid x_1)\, P(x_3 \mid x_1) \\ & \quad P(x_2 \mid x_1)\, P(x_1) \end{aligned} \tag{19}$$

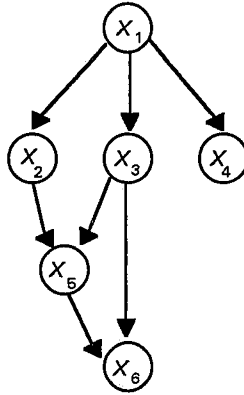


Fig. 2.21 - Exemplo de grafo direcionado acíclico

Entretanto, a representação numérica para $P(x_1, \ldots, x_n)$ apresenta-se raramente disponível na prática. Por esta razão, o grafo pode continuar a ser configurado como antes, mas os conjuntos de pais $\mathbf{P}(X_i)$ devem ser identificados através de julgamento humano.

Os pais de X_i são aquelas variáveis julgadas como causas diretas de X_i ou por terem influências diretas em X_i. As noções formais de causalidade e influência substituem a noção formal de independência condicional. Uma característica importante da representação da rede é que esta permite se expressar diretamente a importância e a qualidade das relações de influência direta.

2.3.2. Simulação do Modelo

Em uma simulação estocástica, cada variável $X_i \in \mathbf{X}$ é atualizada um número de vezes de acordo com a probabilidade condicional

$$P(x_i \mid x_j; j \neq i) \quad \textbf{(20)}$$

e, ao longo da simulação, é contabilizado o número de vezes que o evento $X_i = 1$ acontece, sendo esta freqüência dita como uma aproximação de

$$P(x_i \mid x_j; X_j \in \mathbf{E}) \quad \textbf{(21)}$$

Dado que $\mathbf{V}(X_i) = \mathbf{P}(X_i) \cup \mathbf{F}(X_i) \cup \mathbf{C}(X_i)$ é o conjunto das variáveis vizinhas de X_i, definimos que para uma rede Bayesiana a probabilidade em (20) é dada por

$$P(x_i \mid x_j; j \neq i) = P(x_i \mid x_j; X_j \in \mathbf{V}(X_i)) \quad \textbf{(22)}$$

Pode-se mostrar que esta probabilidade é calculada pela equação

$$P(x_i \mid x_j; j \neq i) = \alpha P(x_i \mid x_j; X_j \in \mathbf{P}(X_i)) \prod_{X_j \in \mathbf{F}(X_i)} P(x_j \mid x_k; X_k \in \mathbf{P}(X_j)) \quad \textbf{(23)}$$

onde α é uma constante de normalização. É importante notar que os elementos de $\mathbf{P}(X_j)$, interno ao produtório, também pertencem a $\mathbf{C}(X_i)$. Considerando uma simulação paralela distribuída, a equação (23) nos permite associar um processo a cada variável em $\mathbf{X}$, comunicando-se através de canais que interligam as correspondentes variáveis de acordo com a relação de vizinhança definida pelos conjuntos $\mathbf{V}(X_i)$ para todo $1 \leq i \leq n$.

Podemos garantir que a simulação converge para um estado de equilíbrio, se a distribuição de probabilidade P for estritamente positiva. Portanto, as probabilidades condicionais estão restritas a

$$0 < P(X_i = 1 \mid x_j; X_j \in P(X_i)) < 1 \text{ e}$$
$$P(X_i = 1 \mid x_j; X_j \in \mathbf{P}(X_i)) + P(X_i = 0 \mid x_j; X_j \in \mathbf{P}(X_i)) = 1$$

para todo $X_i \in \mathbf{X}$ e toda combinação dos x_j's.

A seguir apresentamos um algoritmo paralelo do simulador no qual a implementação foi baseada. Como mencionado anteriormente, para cada variável X_i teremos um processo p_i responsável por sua atualização. Cada p_i armazena localmente as probabilidades $P(x_i \mid x_j; X_j \in \mathbf{P}(X_i))$ e atualiza X_i de acordo com a probabilidade da equação (23). Esta equação demonstra a dependência pelas variáveis pertencentes a $\mathbf{V}(X_i)$ implicando na não atualização concorrente de X_i e X_j caso $X_j \in \mathbf{V}(X_i)$. Por este motivo, utilizamos o protocolo de escalonamento por reversão de arestas descrito na

seção 2.1. O direcionamento atribuído ao grafo formado pelos processos, não tem relação nenhuma com a orientação da rede Bayesiana. O primeiro é utilizado apenas para a execução do protocolo.

Depois que uma variável X_i é atualizada por ser um *sink*, o processo p_i envia mensagens para os processos cujas variáveis estejam em $\mathbf{V}(X_i)$ indicando a reversão das arestas incidentes a p_i. O processo p_i envia probabilidades condicionais para aqueles em $\mathbf{P}(X_i)$, o novo valor de X_i para aqueles em $\mathbf{F}(X_i)$ e aqueles em $\mathbf{C}(X_i)$ recebem sinais que apenas informam a reversão da aresta em questão.

Na figura 2.22 temos o algoritmo do processo p_i, onde utilizamos as definições de vizinhos *upstreams* e *downstreams* descritos na seção 2.2.1.3.1 e denominamos para cada $X_j \in \mathbf{P}(X_i)$

$$\begin{aligned} \pi_i^1(j) &= P(X_i = 1 \mid X_j = 1, x_k; k \neq j, X_k \in \mathbf{P}(X_i)) \text{ e} \\ \pi_i^0(j) &= P(X_i = 1 \mid X_j = 0, x_k; k \neq j, X_k \in \mathbf{P}(X_i)) \end{aligned} \tag{24}$$

O último passo do algoritmo da figura 2.22 garante a não existência de mensagens em trânsito ao término da simulação.

Assim como nos outros modelos descritos anteriormente, na implementação do simulador, alocamos um grupo de variáveis em cada processo p_i visando uma melhor eficiência. A estrutura de p_i é a mesma introduzida na seção 2.2.1.3.2 e ilustrada na figura 2.7. O processo p_i é formado então pelos processos *principal*, *tratar isolado*, *enviar* e *buffer de entrada*. As estruturas dos processadores estão na figura 2.5.

Na forma como a rede Bayesiana foi apresentada, não é necessário a detecção de convergência, uma vez que é estabelecido o número de iterações, isto é, o número de vezes que cada variável é atualizada. Portanto, utilizamos o processo p_M apenas para a distribuição dos dados de entrada, identificação dos subgrafos isolados e coleta dos valores finais das variáveis.

Cada processo p_i inicialmente estabelece uma orientação acíclica do grafo formado pelas variáveis e suas dependências. Esta orientação é composta pelo direcionamento das ligações internas e externas definidas na seção 2.2.1.3.3. Todos os processos recebem como dado de entrada os valores iniciais $\mathbf{X}^{\mathbf{inic}}$ de todas as variáveis, sendo necessário receber as probabilidades condicionais definidas em (24) dos vizinhos externos *upstream* que são filhos das variáveis correspondentes. Quando p_i recebe todas as probabilidades referentes a uma variável, esta se torna ativa, indicando que se for um *sink*, já pode ser atualizada. Dada uma variável X_j alocada em p_i, caso exista uma variável $X_k \in \mathbf{F}(X_j)$, tal que X_k é *upstream* a X_j e X_k também está alocada em p_i, deve-se calcular estas probabilidades condicionais armazenando-as em uma estrutura de dados com a finalidade de ser utilizada na atualização de X_j. Além disto, durante a inicialização, marcam-se as variáveis *upstream* de todas aquelas alocadas em p_i.

Marcar os vizinhos *upstreams* de X_i.
Enviar o valor de X_i para todos os processos p_j tal que $X_j \in F(X_i)$.
Receber de todo p_j tal que $X_j \in P(X_i)$, o valor de X_j.
Se X_i = 1 então

  Enviar $\pi_i^1(j)$ e $\pi_i^0(j)$ para todo p_j tal que $X_j \in P(X_i)$ e seja vizinho *downstream* de X_i.

Senão

  Enviar $1-\pi_i^1(j)$ e $1-\pi_i^0(j)$ para todo p_j tal que $X_j \in P(X_i)$ e seja vizinho downstream de X_i.

Fim-se
Receber de cada processo p_j tal que $X_j \in F(X_i)$ e X_j é vizinho *upstream* de X_i, duas probabilidades a serem armazenadas em ρ_j^1 e ρ_j^0.
Repetir K vezes

  Receber de todos os p_j tal que $X_j \in V(X_i)$ e X_j é vizinho *downstream* de X_i, o valor de X_j, se $X_j \in P(X_i)$; as probabilidades ρ_j^1 e ρ_j^0, se $X_j \in F(X_i)$; e uma mensagem a ser ignorada, se $X_j \in C(X_i)$.

  Dado que $a = P(X_i = 1 \mid x_j; X_j \in P(X_i)) \prod_{X_j \in F(X_i)} \rho_j^1$ e $b = [1 - P(X_i = 1 \mid x_j; X_j \in P(X_i))] \prod_{X_j \in F(X_i)} \rho_j^0$, caso $X_i \notin E$, atualizar X_i tendo X_i = 1 com probabilidade a/(a + b) e X_i = 0 caso contrário.

  Se X_i = 1 então

    Enviar $\pi_i^1(j)$ e $\pi_i^0(j)$ para todo p_j tal que $X_j \in P(X_i)$

  Senão

    Enviar $1-\pi_i^1(j)$ e $1-\pi_i^0(j)$ para todo p_j tal que $X_j \in P(X_i)$

  Fim-se

Fim-rep

Receber uma mensagem (ignorando-a) de cada p_j tal que $X_j \in V(X_i)$ e X_j está marcado.

Fig. 2.22 - Algoritmo do processo p_i do simulador da rede Bayesiana.

Quando uma variável *sink* X_j se torna ativa, ela é atualizada e enviam-se mensagens para seus vizinhos externos onde o conteúdo das mensagens dependem do tipo de relação existente entre as variáveis em questão. Se X_j tiver vizinhos internos, isto é, se existe X_k tal que $X_k \in V(X_j)$ e X_k está alocada no mesmo processo que X_j, simula-se o envio destas mensagens atualizando estruturas de dados. Um processo p_i termina quando todas as

suas variáveis tiverem sido atualizadas um determinado número de vezes e p_i tiver recebido mensagens de todos os vizinhos marcados depois deste número de atualizações tiver sido alcançado.

Logo abaixo temos os passos seguidos pelo processo *principal* que compõe o processo p_i de modo mais detalhado.

1. Inicializar *fim* com falso, receber os valores iniciais das variáveis pertencentes a $\mathbf{X}$, receber as probabilidades condicionais das variáveis X_j tal que $X_j \in p_i$, receber o número de iterações K a serem executadas e enviar um sinal para o processo *tratar isolado* indicando que todos os dados de entrada foram recebidos.

2. Para cada variável $X_j \in p_i$ contar o número de vizinhos de X_j armazenando-o em $n.viz_j$; contar o número de vizinhos *upstream* armazenando-o em $n.up_j$ e em $n.marca_j$; marcar estes vizinhos *upstreams*; inicializar a_j e b_j com o valor 1, $iteração_j$ e $n.init_j$ com o valor 0; e tornar X_j inativo.

3. Para cada variável $X_j \in p_i$ e para cada $X_k \in \mathbf{V}(X_j)$ e tal que $X_k \in p_i$, se X_k for *upstream* a X_j e $X_k \notin \mathbf{F}(X_j)$, incrementar $n.init_j$ de 1; se X_k for *downstream* a X_j e $X_k \in \mathbf{P}(X_j)$, calcular $\pi_j^1(k)$ e $\pi_j^0(k)$ de acordo com (24), incrementar $n.init_k$ de 1, e se $X_j = 1$ calcular $a_k \leftarrow a_k \pi_j^1(k)$ e $b_k \leftarrow b_k \pi_j^0(k)$, senão calcular $a_k \leftarrow a_k\,(1-\pi_j^1(k))$ e $b_k \leftarrow b_k\,(1-\pi_j^0(k))$.

4. Para cada $X_j \in p_i$ se $n.init_j = n.up_j$, tornar X_j ativo e executar **operar** (j).

5. Retirar uma mensagem do *buffer* de entrada. Seja o emissor da mensagem X_k, temos três possibilidades, X_k pode ser pai, filho ou companheiro de uma ou mais variáveis em p_i. Se a mensagem se destinar às variáveis $X_j \in \mathbf{P}(X_k)$ então para cada X_j, enquanto *fim* = falso, faça:

    Se $iteração_j = K$ então
        Se X_k estiver marcado então
            $n.marca_j \leftarrow n.marca_j - 1$
            Executar **testar.fim**
        Fim-se
    Senão
        Seja ρ_k^1 e ρ_k^0 os valores armazenados na mensagem
        $a_j \leftarrow a_j\,\rho_k^1$
        $b_j \leftarrow b_j\,\rho_k^0$
        Se X_j estiver inativo então
            Se X_k for *upstream* a X_j então
                $n.init_j \leftarrow n.init_j + 1$
                Se $n.init_j = n.up_j$ então tornar X_j ativo e executar **operar** (j)
            Senão

$n.init_j \leftarrow n.init_j + 1$
$n.up_j \leftarrow n.up_j + 1$
Tornar X_k *upstream* de X_j
Fim-se
Senão
$n.up_j \leftarrow n.up_j + 1$
Tornar X_k *upstream* de X_j e executar **operar** (j)
Fim-se
Fim-se

Se a mensagem se destinar a $X_j \notin \mathbf{P}(X_k)$ então para cada X_j, enquanto *fim* = falso faça:

Se $iteração_j = K$ então
Se X_k estiver marcado então
$n.marca_j \leftarrow n.marca_j - 1$
Executar **testar.fim**
Fim-se
Senão
Se $X_k \in \mathbf{P}(X_j)$ então armazenar o valor da mensagem em x_k
$n.up_j \leftarrow n.up_j + 1$
Tornar X_k *upstream* de X_j
Se X_j estiver ativo então executar **operar** (j) senão incrementar $n.init_j$ de 1
Fim-se

6. Se *fim* = verdade, então repetir o passo 5. Caso contrário, enviar sinal de finalização para os processos *enviar* e *buffer de entrada*, e terminar o processamento.

O algoritmo dos procedimentos **testar.fim** e **operar** se encontram nas figuras 2.23 e 2.24 respectivamente.

Procedimento Testar.fim

fim **←** **verdade**
Para cada $X_j \in p_i$ **e** X_j **não pertence a um subgrafo isolado faça**
Se $iteração_j \neq K$ **ou** $n.marca_j \neq 0$ **então**
fim **←** **falso**
Fim-se
Fim-para

Fig. 2.23 - Algoritmo do procedimento **testar.fim**.

```
Procedimento Operar (j)

    Se X_j é um sink, isto é, se n.up_j = n.viz_j então
        n.up_j ← 0
        iteração_j ← iteração_j + 1
        Se X_j ∉ E então
            a_j ← a_j P(X_j = 1 | x_k; X_k ∈ P(X_j))
            b_j ← b_j (1 − P(X_j = 1 | x_k; X_k ∈ P(X_j)))
            Atribuir 1 a x_j com probabilidade a_j / (a_j + b_j), senão atribuir 0
        Fim-se
        a_j ← 1; b_j ← 1
        Executar testar.fim
        Para cada X_k ∈ V(X_j) tal que X_k ∉ p_i faça
            Se X_k ∈ P(X_j) então
                Calcular π_j^1(k) e π_j^0(k) de acordo com (24)
                Se x_j = 1 então
                    Enviar π_j^1(k) e π_j^0(k) para o processo enviar com
                    destino ao processador que contém X_k
                Senão
                    Enviar (1 − π_j^1(k)) e (1 − π_j^0(k)) para o processo enviar
                    com destino ao processador que contém X_k
                Fim-se
            Se X_k ∈ F(X_j) então
                Enviar x_j para o processo enviar com destino ao
                processador que contém X_k
            Se X_k ∈ C(X_j) então
                Enviar uma mensagem de flag para o processo enviar com
                destino ao processador que contém X_k
            Fim-se
            Tornar X_k vizinho downstream de X_j
        Fim-para
        Para cada X_k ∈ V(X_j) tal que X_k ∈ p_i e não fim faça
            Tornar X_k vizinho downstream de X_j
            Se iteração_k = K então
                Se X_j estiver marcado em relação a X_k então
                    n.marca_k ← n.marca_k - 1
                    Executar testar.fim
                Fim-se
            Senão
                Se X_j ∈ F(X_k) então
                    Calcular π_j^1(k) e π_j^0(k) de acordo com (24)
                    Se x_j = 1 então
                        a_k ← a_k π_j^1(k)
                        b_k ← b_k π_j^0(k)
                    Senão
```

Fig. 2.24 - Algoritmo do procedimento **operar** (cont. na próxima página).

$a_k \leftarrow a_k\ (1-\pi_j^1(k))$

$b_k \leftarrow b_k\ (1-\pi_j^0(k))$

Fim-se

Fim-se

Se X_k estiver inativo então

$n.init_k \leftarrow n.init_k + 1$

Se X_j for *downstream* a X_k então

$n.up_k \leftarrow n.up_k + 1$

Tornar X_j *upstream* a X_k

Senão

Se $n.up_k = n.init_k$ então tornar X_k ativo

Fim-se

Fim-se

Se X_k estiver ativo e não *fim* então executar operar (k)

Fim-se

Fim-para

Fim-se

Fig. 2.24 - Algoritmo do procedimento **operar** (continuação).

Receber um sinal do processo *principal* indicando que os dados de entrada já foram recebidos.

Se existir algum subgrafo isolado em p_i então

Repetir K iterações

Para cada X_j pertencente a um subgrafo isolado e $X_j \notin E$ faça

$a \leftarrow 1;\ b \leftarrow 1$

Para cada $X_k \in F(X_j)$ faça

Calcular $\pi_k^1(j)$ e $\pi_k^0(j)$ de acordo com (24)

Se $x_k = 1$ então

$a \leftarrow a\ \pi_k^1(j)$ **e** $b \leftarrow b\ \pi_k^0(j)$

Senão

$a \leftarrow a\ (1-\pi_k^1(j))$ **e** $b \leftarrow b\ (1-\pi_k^0(j))$

Fim-se

Fim-para

$a \leftarrow a\ P(X_j = 1 \mid x_k; X_k \in P(X_j))$

$b \leftarrow b\ [1 - P(X_j = 1 \mid x_k; X_k \in P(X_j))]$

Atribuir 1 a x_j com probabilidade $a\ /\ (a + b)$, senão atribuir 0

Fim-para

Fim-rep

Enviar os valores de x_j, tal que X_j pertence a algum subgrafo isolado, para o processo *enviar* com destino p_M.

Fim-se

Fig. 2.25 - Algoritmo do processo *tratar isolado*.

O algoritmo do simulador seqüencial é muito semelhante ao algoritmo do processo *tratar isolado*, por este motivo apresentamos apenas o algoritmo do último na figura 2.25.

Capítulo III

3. Modelos Baseados no Protocolo Distribuído de Sincronização

O capítulo três tem como objetivo apresentar o desenvolvimento dos simuladores distribuídos baseados nos modelos de sistemas complexos que têm em comum algoritmos síncronos. Primeiramente, descreveremos um sincronizador para transformar os algoritmos síncronos em assíncronos possibilitando assim a implementação dos modelos, e em seguida mostraremos o desenvolvimento dos modelos em questão.

Um sistema distribuído **síncrono** é aquele em que existe uma base de tempo comum a todos os elementos do sistema, isto é, existe um relógio global de forma que todos os elementos do sistema têm acesso a ele e executam cada passo do seu processamento em um *clock* deste relógio. Este tipo de sistema distribuído na realidade não existe, pois todos os ambientes de processamento são assíncronos, logo os sistemas são **assíncronos**, isto é, não possuem uma base de tempo global. Cada elemento tem o seu relógio de tempo local e são independentes entre si.

Os modelos estudados foram desenvolvidos inicialmente de forma síncrona apresentando assim um problema que pode ser resolvido com a utilização de um **sincronizador**. Um sincronizador é um protocolo distribuído que permite a execução assíncrona de algoritmos distribuídos projetados para sistemas síncronos. O sincronizador que utilizaremos é o chamado **sincronizador α** que será descrito a seguir.

3.1. Descrição do Sincronizador Alfa (α)

Em um sistema síncrono, no início de uma unidade de tempo do relógio global no nó p_i, todas as mensagens enviadas para p_i no início da unidade anterior já foram recebidas. O sincronizador tem então que simular este comportamento. Para tanto utilizaremos as seguintes regras: cada mensagem do algoritmo síncrono terá uma mensagem de confirmação correspondente; diremos que um nó está seguro em relação a um ciclo do relógio síncrono se todas as mensagens enviadas por ele naquele ciclo tiverem chegado aos seus destinos. Baseado nestas regras definimos que um nó p_i só pode iniciar um novo ciclo do relógio síncrono quando todos os seus vizinhos estiverem seguros em relação ao ciclo anterior.

Tendo em vista estas regras e definições, podemos descrever o algoritmo do sincronizador α para o nó p_i no ciclo $j \geq 1$ da seguinte forma:

- Quando p_i tiver recebido as confirmações de todas as mensagens enviadas por ele no ciclo j, p_i estará seguro e deve informar isto aos seus vizinhos;

- Quando p_i souber que todos os seus vizinhos estão seguros, ele pode começar o ciclo $j + 1$.

O custo de comunicação do sincronizador α é muito alto, mas ele pode ser simplificado pois nos modelos implementados, em cada ciclo do relógio, cada elemento envia apenas uma mensagem para cada um dos seus vizinhos. Desta forma, torna-se desnecessário as mensagens de confirmação e o algoritmo do sincronizador fica da seguinte forma: quando p_i tiver recebido uma mensagem de cada um de seus vizinhos, ele inicia o próximo ciclo que, neste caso, se compõe de executar os procedimentos internos de p_i e enviar uma mensagem para cada vizinho. Como as mensagens de controle foram eliminadas, pode ocorrer de um elemento receber duas mensagens de um mesmo vizinho antes de ter recebido todas as mensagens necessárias para iniciar o próximo ciclo. Entretanto, é garantido que não mais de duas mensagens podem ser recebidas de um mesmo vizinho neste caso. Isto é facilmente mostrado da seguinte forma: seja p_i e p_j dois elementos vizinhos do sistema. Inicialmente, cada elemento executa o seu processamento interno e envia uma mensagem para cada vizinho. Vamos supor então que p_i recebe a mensagem enviada por p_j mas fica esperando as mensagens dos seus outros vizinhos. Por outro lado, p_j recebe a mensagem enviada por p_i e por todos os seus vizinhos, de forma que já pode iniciar um novo ciclo e enviar novas mensagens. Assim, p_i receberá um novo valor de p_j antes de começar um novo ciclo. Mas é impossível que p_i receba uma terceira mensagem de p_j pois este só pode gerar esta terceira mensagem depois que receber uma mensagem de p_i indicando que este começou um novo ciclo. Devido a isto, basta reservamos um espaço para armazenarmos esta segunda mensagem.

Esta simplificação diminui o custo de comunicação, visto que o número de mensagens é reduzido devida a eliminação das mensagens de confirmação e das mensagens indicando que um determinado elemento está seguro.

3.2. Redes Neuronais

A seguir estudaremos alguns modelos de redes neuronais empregando a mesma nomenclatura apresentada na seção 2.2. Os modelos a serem descritos têm em comum algoritmos síncronos e, por este motivo, utilizaremos o sincronizador α descrito na seção anterior.

3.2.1. Rede Neuronal Contínua de Hopfield

Este modelo surgiu a partir de críticas e especulações a respeito do modelo de rede neuronal binária. O modelo matemático deste último, foi baseado em "neurônios" que diferem tanto dos neurônios biológicos reais quanto do funcionamento real de circuitos eletrônicos simples. Algumas destas diferenças são tão gritantes que neurobiologistas e

engenheiros eletrônicos têm questionado se redes neuronais e circuitos elétricos reais realmente se comportam como o sistema do modelo.

As duas maiores divergências entre o modelo binário e os sistemas biológico e físico, são que os neurônios reais possuem relações de entrada-saída contínuas, enquanto que o modelo original tem saídas com valores 0 ou 1. A segunda divergência se refere ao fato de que neurônios e circuitos físicos reais sofrem atrasos no tempo devido a capacitância, e o tempo de evolução destes sistemas devem ser representados por uma equação diferencial. Por outro lado, o modelo original usa um algoritmo estocástico envolvendo mudanças súbitas dos estados dos neurônios de 0 para 1 ou de 1 para 0 em momentos aleatórios.

O modelo de rede neuronal contínua elimina estas divergências garantindo que as propriedades do modelo original são mantidas intactas. Apesar da incerteza de que as propriedades dos novos "neurônios" contínuos sejam suficientemente análogas às propriedades essenciais dos neurônios reais para ser aplicado à neurobiologia, o maior obstáculo conceitual foi eliminado.

3.2.1.1. Descrição do Modelo

O novo modelo é baseado em variáveis e respostas contínuas, retendo todos os comportamentos significativos do modelo binário. Seja a saída V_i do neurônio N_i restrita ao intervalo $0 < V_i < 1$ e seja f_i uma função contínua e monotonicamente crescente da entrada P_i do neurônio N_i. Teremos então $V_i = f_i(P_i)$. Uma relação típica de f_i é a sigmoide assintótica a 0 e 1, ilustrada na figura 3.1

$$V_i = \frac{1}{1+e^{-(P_i-\theta_i)/T}} \qquad (1)$$

onde θ_i pode ser interpretado como um limiar, da mesma forma que no caso de neurônios binários, e T controla o grau de curvatura no ponto $P_i = \theta_i$. Neste ponto particularmente, valores pequenos de T determinam um ganho bem alto.

Em um sistema biológico, P_i se atrasa em relação às saídas instantâneas V_j das outras células por causa da capacitância de entrada C das membranas das células, da resistência R ao transpassar a membrana e da impedância finita T^{-1} entre a saída V_j e o corpo da célula i. Logo temos uma equação que considera o ônus resistência-capacitância na determinação da taxa de deslocamento de P_i

$$C_i \frac{dP_i}{dt} = \sum_{j \neq i} w_{ij} V_j + I_i - \frac{P_i}{R_i} \qquad (2)$$

onde $w_{ij}\ V_j$ representa a corrente elétrica de entrada da célula devido ao potencial presente na célula j, e w_{ij} é, portanto, a eficácia da sinapse em questão. I_i é qualquer outra corrente de entrada fixa do neurônio N_i.

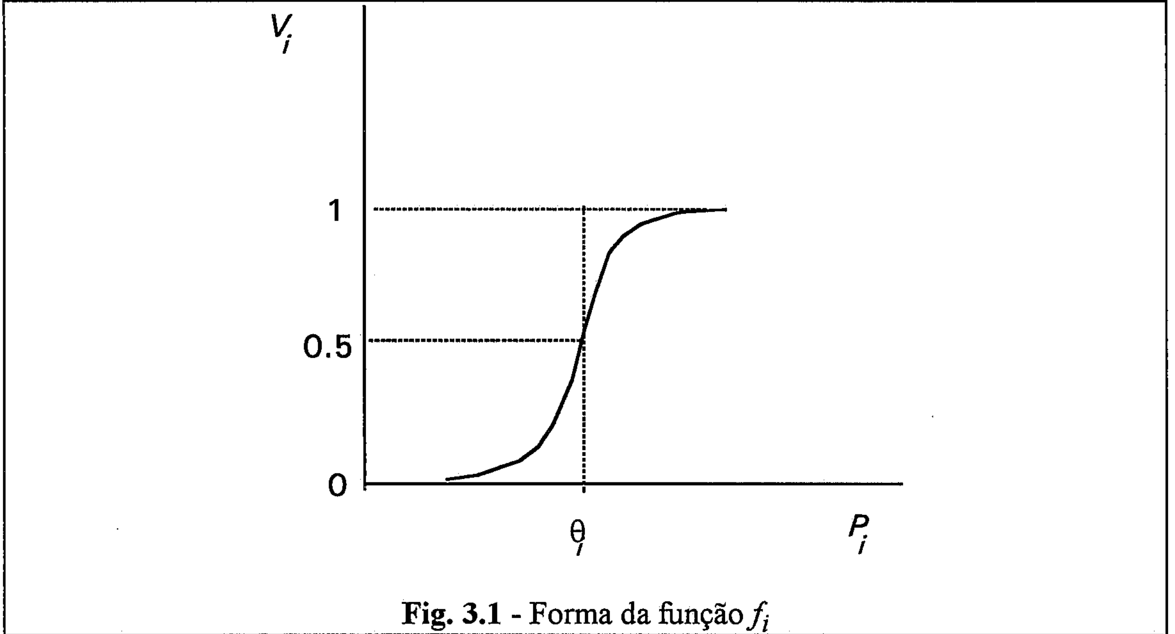


Fig. 3.1 - Forma da função f_i

Considere a seguinte função de energia

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}\ V_j + \sum_{i}\frac{1}{R_i}\int_{0}^{V_i} f_i^{-1}(V)\mathrm{d}V + \sum_{i} I_i\ V_i \tag{3}$$

Ao derivarmos E em função do tempo, considerando w simétrico, teremos

$$\frac{\mathrm{d}E}{\mathrm{d}t} = -\sum_{i}\frac{\mathrm{d}V_i}{\mathrm{d}t}\left(\sum_{j\neq i} w_{ij}\ V_j - \frac{P_i}{R_i} + I_i\right) \tag{4}$$

O que está entre parênteses corresponde ao lado direito da equação (2), logo

$$\begin{aligned}\frac{\mathrm{d}E}{\mathrm{d}t} &= -\sum C_i\left(\frac{\mathrm{d}V_i}{\mathrm{d}t}\right)\left(\frac{\mathrm{d}P_i}{\mathrm{d}t}\right) \\ &= -\sum C_i f_i^{-1}(V_i)\left(\frac{\mathrm{d}V_i}{\mathrm{d}t}\right)^2\end{aligned} \tag{5}$$

Visto que $f_i^{-1}(V_i)$ é uma função monotonicamente crescente e C_i é positivo, cada termo deste somatório é não-negativo. Portanto

$$\frac{\mathrm{d}E}{\mathrm{d}t} \leq 0 \text{ e se } \frac{\mathrm{d}E}{\mathrm{d}t} = 0 \Rightarrow \frac{\mathrm{d}V_i}{\mathrm{d}t} = 0 \text{ para todo } i \qquad \textbf{(6)}$$

A conclusão em (6) demonstra que a evolução do sistema no tempo representa um deslocamento no espaço de estados em que se procura por um mínimo de E, parando nestes pontos. Este modelo determinístico possui as mesmas propriedades de evolução no seu espaço contínuo que o modelo estocástico possui no espaço discreto. Desta forma ele pode ser usado em qualquer problema computacional em que uma função de energia é essencial.

3.2.1.2. Simulação Paralela Distribuída

Da mesma forma que nos outros modelos, primeiro apresentaremos uma versão do simulador mais simplificada descrita em [6] e em seguida descrevemos a implementação do simulador. Análogo aos outros modelos, associamos cada neurônio N_i a um processo p_i. Ao realizarmos esta correspondência, nos deparamos com o parâmetro de tempo real t criando uma base de tempo comum, o que define a rede neuronal como um sistema distribuído síncrono. Este parâmetro t pode ser quantificado de modo que cada *quantum* de tempo represente um passo do relógio global do sistema síncrono. Em cada *quantum* de tempo, todos os processos executam uma nova iteração na integração das equações diferenciais em (2), gerando uma aproximação das equações diferenciais por equações de diferenças. O novo valor de V_i é então calculado através da aplicação de f_i ao novo valor de P_i e em seguida enviado para todos os processos vizinhos de p_i.

Como não existe um sistema distribuído síncrono real, utilizamos o sincronizador α descrito na seção 3.1 como solução do problema de implementação desta simulação. Adaptando o sincronizador ao modelo, teremos que em todos os processos cada passo do relógio global corresponde à atualização dos respectivos neurônios. O funcionamento do sincronizador se resume em, inicialmente, todos os processos enviam os estados de seus neurônios para os seus vizinhos. Um processo então quando receber os valores de seus vizinhos, calcula o novo estado do seu neurônio e o envia para todos os processos vizinhos. A próxima vez que este processo irá atualizar seu neurônio, será quando receber novos valores de todos os seus vizinhos. Este procedimento garante que um processo só inicia um novo passo do relógio global quando todas as mensagens enviadas a ele no passo corrente tiverem sido recebidas. É possível que um processo receba dois valores consecutivos de um processo antes de ter recebido os valores dos outros processos vizinhos referentes ao *quantum* de tempo em questão. Entretanto, pode-se assegurar que não mais de duas mensagens consecutivas sejam recebidas nestes termos. Portanto é suficiente guardar espaço de armazenamento para estas possíveis mensagens.

Utilizamos um processo *mestre* (p_M) para a distribuição dos dados de entrada para os outros processos, a detecção da estabilidade do sistema e a terminação da simulação. Seu processamento se resume em

1. Enviar para os processos p_i e p_j o valor w_{ij} para todo $i \neq j$ tal que $w_{ij} \neq 0$ e $1 \leq i, j \leq n$.

2. Enviar para todo processo p_i o valor de V_i^{inic} que representa o valor inicial do neurônio N_i.

3. Quando um valor real pertencente ao intervalo [0, 1] é recebido de um processo p_i, verificar se um novo estado global é completado. Caso afirmativo, comparar o novo estado global com o imediatamente anterior. Esta comparação é realizada calculando-se a diferença entre os dois estados. Se esta diferença for menor ou igual a um valor pré-estabelecido ε, então conclui-se que o sistema estabilizou, envia-se uma mensagem de STOP para cada processo e espera-se por uma mensagem resposta de STOP de cada um deles terminando o processamento em seguida. Se o sistema não estabilizou, repetir o passo 3.

Assim como na Rede Neuronal Binária, um processo pode estar ativo ou inativo. Todos os p_i's iniciam a simulação inativos, se tornando ativos quando recebem $n.viz_i + 1$ valores iniciais onde um é enviado por p_M e os outros pelos seus vizinhos. Como já mencionado, um processo pode receber dois valores consecutivos de um vizinho antes de atualizar seu neurônio. Para controlar esta situação utilizamos as seguintes variáveis:

- $n.rec_i$ - número de mensagens recebidas devido ao *quantum* de tempo corrente.

- $rec_i(j)$ - indica se p_i recebeu uma mensagem referente ao *quantum* de tempo corrente enviada por p_j.

- $prox\text{-}rec_i(j)$ - indica se p_i recebeu uma mensagem referente ao próximo *quantum* de tempo enviada por p_j.

- $prox\text{-}V_j$ - o valor do neurônio N_j enviado por p_j referente ao próximo *quantum* de tempo.

Na descrição dos passos executado por p_i, empregamos o procedimento **operar**, apresentado na figura 3.2, e o procedimento **ativar**. Este último simplesmente verifica se $n.init_i = n.viz_i + 1$, e sendo assim , torna p_i ativo.

1. Receber os dados de entrada enviados por p_M.

2. Calcular $P_i = f_i^{-1}(V_i)$.

3. Inicializar $rec_i(j)$ e $prox\text{-}rec_i(j)$ com falso para todo j tal que $w_{ij} \neq 0$. E inicializar $n.rec_i$ e $flushed_i$ com 0.

4. Enviar o valor de V_i^{inic} para todos os vizinhos e incrementar $n.init_i$ de 1. Executar o procedimento **ativar** e se p_i se tornar ativo, executar **operar**.

5. Ao receber um valor no intervalo [0, 1] enviado por um processo p_j, e se $rec_i(j)$ for falso, então armazenar este valor em V_j, tornar $rec_i(j)$ verdadeiro e incrementar $n.rec_i$ de 1. Se p_i estiver inativo, em adição, incrementar $n.init_i$ de 1. Se p_i está ou se tornou ativo, executar **operar**.

6. Ao receber um valor no intervalo [0, 1] enviado por p_j e se $rec_i(j)$ for verdadeiro, armazenar este valor em $prox\text{-}V_j$ e tornar $prox\text{-}rec_i(j)$ verdadeiro.

7. Ao receber uma mensagem de STOP, incrementar $flushed_i$ de 1. Se $flushed_i = 1$ então enviar uma mensagem de STOP para cada processo vizinho. Se $flushed_i = n.viz_i + 1$, enviar uma mensagem de STOP para p_M e terminar o processamento.

Os passos de 1 à 4 são de inicialização, enquanto que os passos de 5 à 7 são ativados pelo recebimento de mensagens.

Procedimento Operar

Se $n.rec_i = n.viz_i$ **então**

$\Delta P_i \leftarrow \frac{\Delta t}{C_i}\left(\sum_{j \neq i} w_{ij} V_j + I_i - \frac{P_i}{R_i}\right)$ **onde** Δt **tem um valor muito menor que a constante** $R_i C_i$.

$P_i \leftarrow P_i + \Delta P_i$

$V_i \leftarrow \frac{1}{1+e^{-(P_i-\theta_i)/T}}$

Enviar V_i **para** p_M

$n.rec_i \leftarrow 0$

Para cada vizinho p_j **de** p_i **faça**

- **Enviar** V_i **para** p_j
- $rec_i(j) \leftarrow prox\text{-}rec_i(j)$
- $prox\text{-}rec_i(j) \leftarrow$ **falso**
- **Se** $rec_i(j)$ = **verdade então**
  - $n.rec_i \leftarrow n.rec_i + 1$
  - $V_j \leftarrow prox\text{-}V_j$
- **Fim-se**

Fim-para

Fim-se

Fig. 3.2 - Algoritmo do procedimento **operar**

O simulador implementado foi baseado no algoritmo descrito pelos passos apresentados. O motivo por que não implementamos o modelo utilizando diretamente este algoritmo, é o mesmo do apresentado na descrição do modelo da Rede Neuronal Binária, isto é, devido ao *overhead* causado pela criação de vários processos em paralelo em um mesmo processador. Portanto, neste modelo, também alocaremos um conjunto de neurônios a cada processo p_i, sendo representados e controlados por meio de estruturas de dados locais a cada processo.

Receber aviso enviado pelo processo *principal* indicando que os dados de entrada já foram recebidos.
Se houver algum grafo isolado alocado a p_i então
  Para cada neurônio N_j pertencente a um subgrafo isolado faça
    $P_j \leftarrow f_j^{-1}(V_j)$
  Fim-para
  ***aproximado* ← falso**
  Enquanto não *aproximado* faça
    ***aproximado* ← verdade**
    Para cada N_j pertencente a um subgrafo isolado faça
      $$\Delta P_j \leftarrow \frac{\Delta t}{C_j}\left(\sum_{k \neq j} w_{jk} V_k + I_j - \frac{P_j}{R_j}\right)$$
      $P_j \leftarrow P_j + \Delta P_j$
      $old\text{-}V_j \leftarrow V_j$
      $$V_j \leftarrow \frac{1}{1+e^{-(P_j-\theta_j)/T}}$$
      Se (*aproximado* = verdade) e (|*old-*V_j - V_j| > ε) então
        ***aproximado* ← falso**
      Fim-se
    Fim-para
  Fim-enq
  Enviar os valores finais de todos os neurônios pertencentes a subgrafos isolados para o processo *enviar* com destino p_M
Fim-se

Fig. 3.3 - Algoritmo do processo *tratar isolado.*

Baseado em um grafo *G* formado pelos neurônios e pelas sinapses que os interligam, geramos duas versões do simulador. Uma voltada para grafos bem esparços, e outra para grafos densos. Nos dois casos, as estruturas dos processadores são as mesmas ilustradas na figura 2.5, onde cada processo p_i é composto pelos processos *principal, tratar isolado, enviar* e *buffer de entrada.* Além disso, nas duas versões o processo

mestre tem as mesmas funções já descritas, em adição com a função de identificar a separação do grafo G em subgrafos verificando os que são isolados. Esta identificação é realizada através do procedimento descrito na figura 2.6. O processo *tratar isolado*, responsável pela atualização do(s) subgrafo(s) isolado(s) alocado(s) a um processador, é igual para as duas versões e bem semelhante ao algoritmo do simulador seqüencial. Por este motivo só apresentamos o processo *tratar isolado* na figura 3.3.

Versão Para Grafos Esparsos

Nesta versão aproveitamos o esparsamento do grafo identificando os subgrafos que o constituem. Então, cada processo se torna responsável por "partes" de subgrafos tratando-as como unidades compactas e, sendo assim, os neurônios que compõem cada subgrafo são atualizados no mesmo instante. Um processo só pode atualizar sua parte de um subgrafo, quando tiver recebido os valores de todos os neurônios vizinhos daqueles que formam esta parte do subgrafo. Logo depois, os valores dos neurônios atualizados são enviados para seus vizinhos externos e para p_M.

Procedimento Operar (j)

Se $n.rec_j = n.viz_j$ **então**
- **Para cada neurônio** N_l **pertencente à parte do subgrafo** j **do processo** p_i **faça**
  - $\Delta P_l \leftarrow \frac{\Delta t}{C_l}\left(\sum_{k \neq l} w_{lk} V_l + I_l - \frac{P_l}{R_l}\right)$ **onde** Δt **tem um valor muito menor que a constante** $R_l C_l$.
  - $P_l \leftarrow P_l + \Delta P_l$
  - $V_l \leftarrow \frac{1}{1+e^{-(P_l-\theta_l)/T}}$
  - **Enviar** V_l **para** p_M
- **Fim-para**
- $n.rec_j \leftarrow 0$
- **Para cada vizinho externo** N_k **do subgrafo** j **faça**
  - **Enviar os novos valores dos neurônios atualizados vizinhos de** N_k **para o processo** ***enviar*** **com destino** N_k.
  - $rec_j(k) \leftarrow prox\text{-}rec_j(k)$
  - $prox\text{-}rec_j(k) \leftarrow$ **falso**
  - **Se** $rec_j(k)$ = **verdade então**
    - $n.rec_j \leftarrow n.rec_j + 1$
    - $V_k \leftarrow prox\text{-}V_k$
  - **Fim-se**
- **Fim-para**

Fim-se

Fig. 3.4 - Algoritmo do procedimento **operar**.

Dado que a maioria das variáveis se referem a um subgrafo, o processo *principal* de p_i se traduz nos seguintes passos:

1. Receber os seguintes dados enviados por p_M: V^{inic} de todos os neurônios, $w_{kl} \neq 0$ tal que N_k e/ou N_l estão alocados em p_i, a identificação dos subgrafos.

2. Ao receber um valor no intervalo [0, 1] referente ao neurônio N_k pertencente ao subgrafo j e se $rec_j(k)$ for falso, então armazenar este valor em V_k, tornar $rec_j(k)$ verdadeiro, incrementar $n.rec_j$ de 1 e executar **operar**.

3. Ao receber um valor no intervalo [0, 1] referente a N_k pertencente ao subgrafo j e se $rec_j(k)$ for verdade, então armazenar este valor em $prox\text{-}V_k$ e tornar $prox\text{-}rec_j(k)$ verdade.

4. Ao receber uma mensagem de STOP, incrementar $flushed_i$ de 1. Se $flushed_i = 1$ então enviar uma mensagem de STOP para cada processo vizinho. Se $flushed_i = n.proc$ (onde $n.proc$ representa o número de processadores do sistema), enviar uma mensagem de STOP para p_M e terminar o processamento.

O procedimento operar se encontra na figura 3.4.

Versão Para Grafos Densos

No caso de grafos densos, o número de mensagens trocadas é muito grande e uma forma de remediar isto seria compensando o custo de comunicação através de um maior paralelismo.

Nesta versão, cada processo também será responsável pela atualização de um grupo de neurônios, porém, cada neurônio é tratado individualmente. O algoritmo é basicamente o mesmo, sendo que agora um processo p_i controla as mensagens que cada neurônio interno recebe separadamente, podendo haver o caso de uma mensagem se destinar a vários neurônios dentro de um mesmo processador.

Um processo p_i só poderá atualizar um neurônio interno N_j, quando tiver recebido mensagens correspondentes a todos os vizinhos externos de N_j e quando seus vizinhos internos tiverem alcançado o mesmo *quantum* de tempo que N_j. Depois de N_j ter sido atualizado, envia-se seu novo valor para seus vizinhos externos e verifica-se a possibilidade de, com este novo valor, atualizar algum vizinho interno. Para esta verificação e atualização em cadeia, utilizamos um algoritmo recursivo.

Nesta versão do simulador, os passos do processo *principal* são basicamente os mesmos da outra versão, sendo que todas as variáveis referentes a subgrafos, agora se

referem a neurônios e o procedimento **operar** correspondente está apresentado na figura 3.5.

Procedimento Operar (j)

Se $n.rec_j = n.viz_j$ **então**
  $n.rec_j \leftarrow 0$
  $soma \leftarrow 0$
  Para cada vizinho N_k **de** N_j **faça**
    Se $prox\text{-}rec_j(k)$ **= verdade então**
      $soma \leftarrow soma + w_{jk}\ old\text{-}V_k$
    Senão
      $soma \leftarrow soma + w_{jk}\ V_k$
    Fim-se
  Fim-para

$$\Delta P_j \leftarrow \frac{\Delta t}{C_j}\left(soma + I_j - \frac{P_j}{R_j}\right)$$

  $P_j \leftarrow P_j + \Delta P_j$
  $old\text{-}V_j \leftarrow V_j$

$$V_j \leftarrow \frac{1}{1+e^{-(P_j-\theta_j)/T}}$$

  Enviar V_j **para o processo** ***enviar*** **com destino** p_M
  Para cada vizinho N_k **de** N_j **faça**
    $rec_j(k) \leftarrow prox\text{-}rec_j(k)$
    $prox\text{-}rec_j(k) \leftarrow$ **falso**
    Se $rec_j(k)$ **= verdade então**
      $n.rec_j \leftarrow n.rec_j + 1$
    Fim-se
  Fim-para
  Para cada vizinho externo N_k **de** N_j **faça**
    Enviar o novo valor V_j **para o processo** ***enviar*** **com destino** N_k**.**
  Fim-para
  Para cada vizinho interno N_k **de** N_j **faça**
    Se $rec_k(j)$ **= verdade então**
      $prox\text{-}rec_k(j) \leftarrow$ **verdade**
    Senão
      $rec_k(j) \leftarrow$ **verdade**
      $n.rec_k \leftarrow n.rec_k + 1$
      Executar operar(*k*)
    Fim-se
  Fim-para
Fim-se

Fig. 3.5 - Algoritmo do procedimento **operar**.

A diferença mais marcante diz respeito aos passos 2 e 3, onde na nova versão são compactados em um único passo descrito logo abaixo:

2. Ao receber um valor no intervalo [0, 1] referente ao neurônio externo N_k, atribuir $old\text{-}V_k \leftarrow V_k$ e armazenar o valor recebido em V_k. Para cada neurônio $N_j \in p_i$ tal que $w_{jk} \neq 0$, se $rec_j(k)$ for falso então incrementar $n.rec_j$ de 1, tornar $rec_j(k)$ verdadeiro e executar **operar**(j). Caso contrário, tornar $prox\text{-}rec_j(k)$ verdadeiro.

3.2.2. Resolução de Sistemas Lineares

Nesta seção apresentaremos um modelo de rede neuronal formado por neurônios contínuos, cujo objetivo é solucionar sistemas lineares de equações algébricas $\boldsymbol{Ax} = \boldsymbol{b}$, através da minimização de uma função de erro quadrática que mede o desvio entre os vetores $\boldsymbol{Ax}$ e $\boldsymbol{b}$.

3.2.2.1. Descrição do Problema

Desejamos solucionar o problema de encontrar um vetor $\boldsymbol{x} \in \Re^{m_2}$ tal que $\boldsymbol{Ax} = \boldsymbol{b}$, onde $\boldsymbol{A} \in \Re^{m_1 \times m_2}$ e $\boldsymbol{b} \in \Re^{m_1}$, ilustrado a seguir

$$\begin{bmatrix} a_{11} & \cdots & a_{1m_2} \\ \vdots & \vdots & \vdots \\ a_{m_1 1} & \cdots & a_{m_1 m_2} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{m_2} \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_{m_1} \end{bmatrix} \quad \textbf{(7)}$$

Resolver este problema é análogo à resolução do problema de minimização do erro residual $\varepsilon = \frac{1}{2}\|\boldsymbol{Ax} - \boldsymbol{b}\|^2$, visto que um sistema linear pode ter nenhuma, uma ou uma infinidade de soluções. A notação $\|\ \|$ representa qualquer norma desejada e utilizando a norma Euclideana teremos o problema de minimizar

$$\varepsilon = \frac{1}{2}(\boldsymbol{Ax} - \boldsymbol{b})^T(\boldsymbol{Ax} - \boldsymbol{b}), \quad \textbf{(8)}$$

cujas soluções se localizam em pontos onde $\nabla\varepsilon = 0$ sendo $\nabla\varepsilon = \mathbf{A}^T(\mathbf{Ax} - \mathbf{b})$.

Em sistemas consistentes com uma ou uma infinidade de soluções, este problema de minimização possui respectivamente um ou uma infinidade de pontos onde $\varepsilon = 0$. Por outro lado, em sistemas inconsistentes, o problema apresenta uma ou uma infinidade de soluções aproximadas em que $\varepsilon > 0$.

Uma forma de resolver o problema de minimização é utilizando um algoritmo descrito em [10]. Através deste procedimento, geramos uma seqüência de aproximações $x^1, x^2 \ldots$, a partir de um ponto arbitrário x^0. Esta seqüência é gerada de forma que, para $k \geq 0$, $x^{k+1} = x^k + \alpha(-\nabla\varepsilon)$, onde $-\nabla\varepsilon$ representa a direção descendente e $\alpha > 0$ o passo. O algoritmo termina quando $d^k = -\alpha\ \nabla\varepsilon$ é aproximadamente igual a zero, indicando a proximidade a um determinado ponto. Teremos então

$$d^k = -\alpha\ A^{\mathrm{T}}\left(Ax^k - b\right) \text{ ou, mais especificamente}$$

$$d_j^k = -\alpha \sum_{i=1}^{m_1} a_{ij}\left(\sum_{l=1}^{m_2} a_{il}x_l^k - b_i\right), \text{ para } 1 \leq j \leq m_2 \tag{9}$$

3.2.2.2. Descrição do Modelo

Objetivando a resolução de um sistema linear de equações, utilizaremos uma rede neuronal constituída de dois conjuntos de neurônios $\mathbf{N_1}$ e $\mathbf{N_2}$ de tamanhos m_1 e m_2 respectivamente, sendo $m_1 + m_2 = n$. Adotaremos a convenção de que os neurônios $N_1, \ldots, N_{m_1}$ pertencem a $\mathbf{N_1}$, enquanto que $N_{m_1+1}, \ldots, N_{m_1+m_2}$ pertencem a $\mathbf{N_2}$. A topologia da rede de neurônios está ilustrada na figura 3.6 e os neurônios se interligam de acordo com as seguintes regras (dado que w_{ij} representa o peso sináptico interligando N_j a N_i)

$$w_{ij} = \begin{cases} a_{i,j-m_1}, & \text{se } N_i \in \mathbf{N_1} \text{ e } N_j \in \mathbf{N_2}; \\ -a_{j,i-m_1}, & \text{se } N_i \in \mathbf{N_2} \text{ e } N_j \in \mathbf{N_1}; \\ 0, & \text{em qualquer outro caso.} \end{cases} \tag{10}$$

e os neurônios $N_i \in \mathbf{N_1}$ têm como entrada externa $I_i = -b_i$.

O estado V_j de um neurônio $N_j \in \mathbf{N_2}$ é dado por

$$C\frac{\mathrm{d}V_j}{\mathrm{d}t} = \sum_{i=1}^{m_1} w_{ji}\ V_i - \frac{V_j}{R} \tag{11}$$

onde C e R são a capacitância e resistência do neurônio respectivamente. Para um neurônio $N_i \in \mathbf{N_1}$ teremos

$$V_i = \sum_{j=1}^{m_2} w_{ij}\ V_j - b_i \tag{12}$$

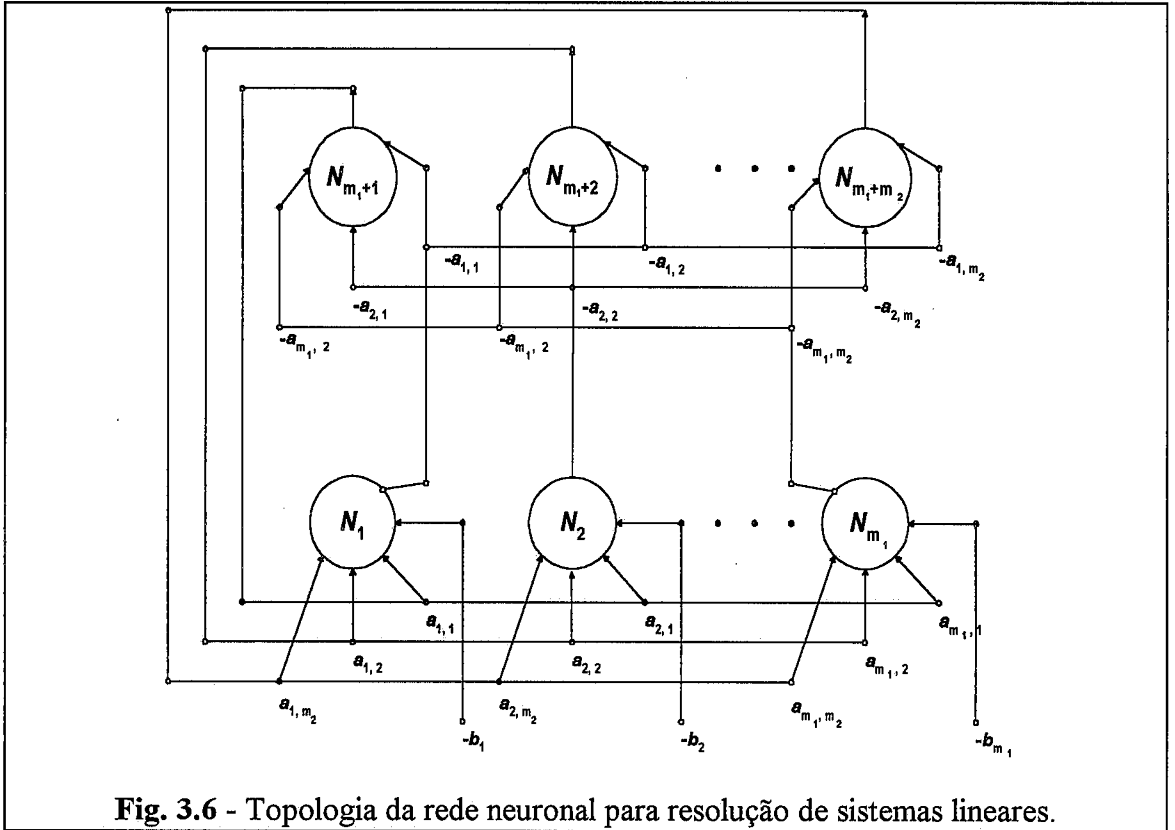


Fig. 3.6 - Topologia da rede neuronal para resolução de sistemas lineares.

Para $1 \leq k \leq m_1$ e $1 \leq l \leq m_2$, seja a_{kl} um elemento de uma matriz em $\Re^{m_1 \times m_2}$ e b_k um elemento de um vetor em $\Re^{m_1}$. Considerando o sistema linear de equações

$$\sum_{l=1}^{m_2} a_{kl}\, x_l = b_k \quad \text{para } 1 \leq k \leq m_1, \tag{13}$$

a equação (11) toma a forma

$$C \frac{\mathrm{d}V_j}{\mathrm{d}t} = \sum_{i=1}^{m_1} a_{i,j-m_1}\, V_i - \frac{V_j}{R} \tag{14}$$

e a equação (12) se torna

$$V_i = \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1}\, V_j - b_i \tag{15}$$

Se $R >> 0$ e substituindo (15) em (14) teremos

$$C\frac{\mathrm{d}V_j}{\mathrm{d}t} = \sum_{i=1}^{m_1} a_{i,j-m_1} \left(\sum_{l=m_1+1}^{m_1+m_2} a_{i,l-m_1} V_l - b_i \right) \tag{16}$$

Esta é a equação (9) com $\alpha = 1 / C$, ao realizarmos a correspondência entre os estados de ativação V_j e os componentes x_j^k do vetor $\boldsymbol{x}^k$. Desta forma, podemos concluir que este modelo de rede neuronal minimiza o erro residual ε e que os estados V_i dos neurônios $N_i \in \mathbf{N}_2$ em todos os momentos $t \geq 0$, nos oferecem uma aproximação da solução do sistema linear $\boldsymbol{Ax} = \boldsymbol{b}$.

3.2.2.3. Simulação Paralela Distribuída

Tendo em vista o uso do parâmetro de tempo t nas equações de atualização dos neurônios, concluímos que este modelo representa um sistema distribuído síncrono. Portanto, utilizaremos o sincronizador α objetivando a implementação do sistema de forma assíncrona, cujo comportamento simula um sistema síncrono.

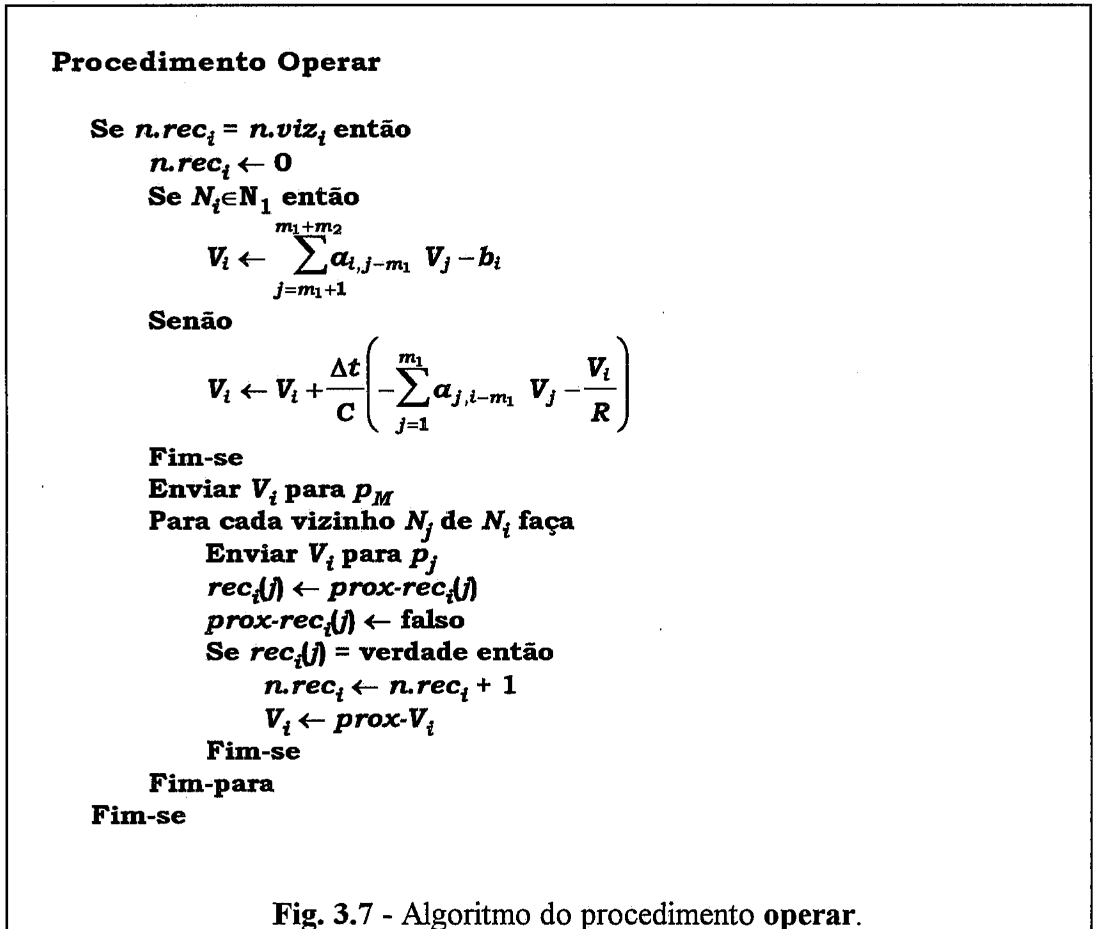
Observando as equações (14) e (15), notamos que a condição de atualização de um neurônio N_i pertencente a um grupo, é a de que este tenha conhecimento dos valores de todos os neurônios do outro grupo. Associando cada neurônio N_i a um processo p_i, o fato mencionado acima se reflete na topologia formada pelos processos. Logo, cada processo p_i responsável pelo neurônio $N_i \in \mathbf{N}_j$ onde $j \in \{1, 2\}$, é interligado através de canais de comunicação, a todos os processos p_k tal que $N_k \in \mathbf{N}_l$ onde $l \neq j$ e $l \in \{1, 2\}$.

O sincronizador funciona da seguinte maneira: inicialmente todos os processos enviam os valores dos seus neurônios para seus processos vizinhos. Quando um processo recebe os valores de todos os seus vizinhos, ele atualiza seu neurônio e envia este novo valor para os seus vizinhos. Este procedimento é repetido até o sistema convergir para um estado de equilíbrio. Assim como no simulador da Rede Neuronal Contínua de Hopfield, é possível que um processo receba dois valores consecutivos de um mesmo neurônio antes de atualizar o seu respectivo neurônio. Portanto, também é necessário reservar um espaço para armazenar estes possíveis valores.

O processo responsável pela detecção da convergência do sistema é o *processo mestre* (p_M). Para tanto, um processo precisa enviar o valor de seu neurônio para p_M de tempos em tempos. O teste de convergência é realizado através da comparação entre os dois últimos estados globais consecutivos do sistema. Se o módulo da diferença entre os dois for menor ou igual a uma tolerância ξ, então diz-se que o sistema estabilizou. Ao detectar a estabilidade do sistema, p_M envia um sinal para todos os processos indicando o término da simulação.

Os algoritmos de p_M e dos processo p_i, são os mesmos dos utilizados nos algoritmos originais da simulação da Rede Neuronal Contínua de Hopfield descritos na

seção 3.2.1.2. A única diferença se encontra no algoritmo do procedimento **operar** ilustrado na figura 3.7.



Procedimento Operar

Se $n.rec_i = n.viz_i$ **então**
    $n.rec_i \leftarrow 0$
    Se $N_i \in \mathbf{N}_1$ **então**

$$V_i \leftarrow \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} V_j - b_i$$

    Senão

$$V_i \leftarrow V_i + \frac{\Delta t}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1} V_j - \frac{V_i}{R}\right)$$

    Fim-se
    Enviar V_i **para** p_M
    Para cada vizinho N_j **de** N_i **faça**
        Enviar V_i **para** p_j
        $rec_i(j) \leftarrow prox\text{-}rec_i(j)$
        $prox\text{-}rec_i(j) \leftarrow$ **falso**
        Se $rec_i(j) =$ **verdade então**
            $n.rec_i \leftarrow n.rec_i + 1$
            $V_i \leftarrow prox\text{-}V_i$
        Fim-se
    Fim-para
Fim-se

Fig. 3.7 - Algoritmo do procedimento **operar**.

A partir destes algoritmos, implementamos o simulador alocando vários neurônios a cada processo p_i. Neste modelo, a estrutura do processo p_i difere dos outros modelos apresentados. Não utilizamos o processo *tratar isolado*, uma vez que é impossível haver um subgrafo que não se comunique com nenhum outro processador, pois isto só acontece quando todos os neurônios dos dois grupos estão no mesmo processador. A estrutura de p_i fica então reduzida à estrutura ilustrada na figura 3.8, onde os processos *enviar* e *buffer de entrada* são iguais aos descritos nas figuras 2.11 e 2.10 respectivamente.

A divisão dos neurônios em dois grupos, onde os neurônios de um grupo só se comunicam com os do outro grupo, simplifica a implementação, pois todos os neurônios de um mesmo grupo precisam dos mesmos valores para serem atualizados. Portanto, cada processo atualiza todos os neurônios pertencentes a um grupo por vez.

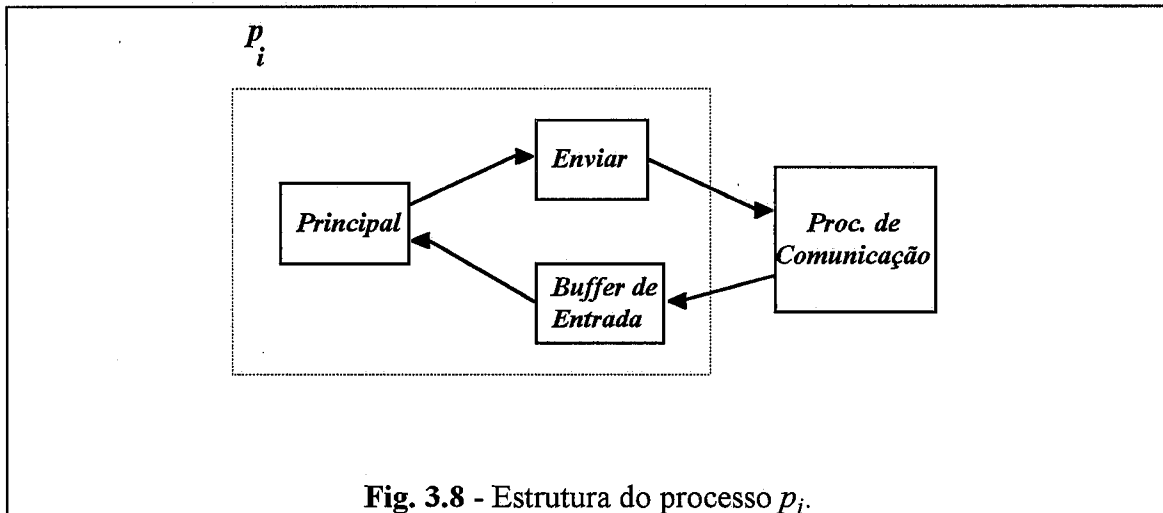


Fig. 3.8 - Estrutura do processo p_i.

É importante notar também que esta topologia nos possibilita alocar os neurônios nos processadores de forma a minimizar o número de mensagens trocadas. Os processadores estão interligados fisicamente através de uma topologia de hipercubo, logo se cada processador enviar mensagens para todos os outros, é necessário fazer um roteamento das mensagens trocadas por processadores que não são diretamente interligados por canais de comunicação. Este roteamento gera trocas de mensagens entre processadores que não estavam incluídos originalmente na dupla emissor e receptor da mensagem. O nosso objetivo então é não gerar mensagens que precisam ser roteadas. Como os neurônios só se comunicam com aqueles de grupo diferente dos seus, para cada par de processadores diretamente interligados, alocamos neurônios de grupos diferentes, isto é, se os processadores i e j são vizinhos, alocamos em i neurônios $N_k \in \mathbf{N_1}$ e em j neurônios $N_l \in \mathbf{N_2}$. Esta forma de alocação está ilustrada na figura 3.9.

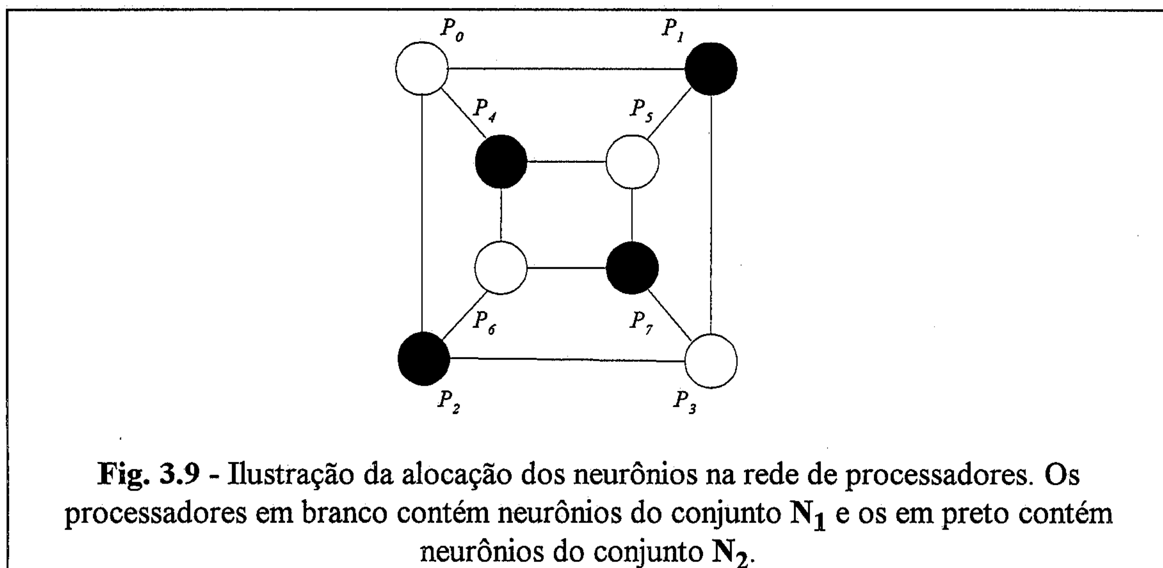


Fig. 3.9 - Ilustração da alocação dos neurônios na rede de processadores. Os processadores em branco contém neurônios do conjunto $\mathbf{N_1}$ e os em preto contém neurônios do conjunto $\mathbf{N_2}$.

Observando a figura, nota-se que cada processador só se comunica indiretamente com um processador, mantendo apenas 1 / 3 das possíveis mensagens originalmente roteadas. No algoritmo seguido pelo processo p_i utilizamos os procedimentos **atualizar** e **atualizar.parcial** descritos nas figuras 3.10 e 3.11 respectivamente.

Os passos seguidos pelo processo p_i estão descritos logo abaixo:

1. Receber os valores a_{jk} tal que N_j e/ou N_{k+m_1} estão alocados em p_i. Receber os valores b_j se N_j estiver alocado em p_i. Receber os valores iniciais V_j de todos os neurônios e se $grupo_j = 1$ então armazenar V_j em $old\text{-}V_j$.

2. Inicializar *n.rec*(0), *n.rec*(1) e *flushed* com 0, $soma_j$ com 0 para todo $N_j \in p_i$, *rec*(*j*) com verdade e *prox-rec*(*j*) com falso para $1 \leq j \leq m_1 + m_2$. Para cada neurônio N_j, onde $1 \leq j \leq m_1 + m_2$, executar **atualizar.parcial(***grupo*, V_j, *j***)** onde $grupo \neq grupo_j$.

3. Executar **atualizar(0)** e **atualizar(1)**, onde os argumentos do procedimento representam a identificação do grupo de neurônios a ser atualizado.

4. Ao receber um valor real referente a um neurônio N_k pertencente ao conjunto $grupo_k$, se *rec*(*k*) for verdadeiro então armazenar este valor em V_k e tornar *prox-rec*(*k*) verdadeiro. Caso contrário, tornar *rec*(*k*) verdadeiro, incrementar $n.rec(grupo_k)$ de 1 e executar **atualizar.parcial(***grupo*, valor recebido, *k***)** onde $grupo \neq grupo_k$. Se $n.rec(grupo_k) = m_{grupo_k+1}$ então executar **atualizar(***grupo***)** tal que $grupo \neq grupo_k$. Enquanto $n.rec(grupo) = m_{grupo+1}$, atribuir $grupo \leftarrow (grupo + 1) \bmod 2$ e executar **atualizar(***grupo***)**.

5. Ao receber uma mensagem de STOP, incrementar *flushed* de 1. Se *flushed* = 1 então enviar uma mensagem de STOP para todos os outros processadores. Se *flushed* = *n.proc* então enviar uma mensagem de STOP para p_M e terminar o processamento enviando um sinal de finalização para os processos *enviar* e *buffer de entrada*.

Os três primeiros passos são de inicialização, enquanto que os dois últimos são ativados pelo recebimento de mensagens.

Procedimento Atualizar(*grupo*)

    outro.gr ← (***grupo*** + 1) **mod 2**
    ***n.rec*(*outro.gr*)** ← **0**
    Para cada neurônio $N_j \in p_i$ **tal que** $grupo_j = grupo$ **faça**
        Se ***grupo*** **= 0 então**
            $valor \leftarrow soma_j - b_j$
        Senão
            $valor \leftarrow old-V_j + \left(\frac{\Delta t}{C}\left(-soma_j\right)\right)$
            $old\text{-}V_j \leftarrow valor$
        Fim-se
        $soma_j \leftarrow 0$
        Se ***rec*(*j*)** **= verdade então**
            $V_j \leftarrow valor$
            ***prox-rec*(*j*)** ← **verdade**
        Senão
            ***rec*(*j*)** ← **verdade**
            ***n.rec*(*grupo*)** ← ***n.rec*(*grupo*)** + **1**
            Executar atualizar.parcial(*outro.gr*, *valor*, *j*)
        Fim-se
        Para cada processador *k* que possui neurônios do conjunto *outro.gr* faça
            Enviar *valor* para o processo *enviar* destinado ao processador *k*
        Fim-para
        Enviar *valor* para o processo *enviar* com destino p_M
    Fim-para
    Se ***grupo*** **= 0 então**
        $início \leftarrow m_1 + 1$
        $fim \leftarrow m_1 + m_2$
    Senão
        $início \leftarrow 1$
        $fim \leftarrow m_1$
    Fim-se
    Para *j* de *início* até *fim* faça
        ***rec*(*j*)** ← ***prox-rec*(*j*)**
        Se ***rec*(*j*)** **= verdade então**
            ***prox-rec*(*j*)** ← **falso**
            ***n.rec*(*outro.gr*)** ← ***n.rec*(*outro.gr*)** + **1**
            Executar atualizar.parcial(*grupo*, V_j**, *j*)**
        Fim-se
    Fim-para

Fig. 3.10 - Algoritmo do procedimento **atualizar**.

```
Procedimento Atualizar.Parcial(grupo, valor, j)

    Se grupo = 0 então
        Para cada neurônio N_k ∈ p_i tal que grupo_k = 0 faça
            soma_k ← soma_k + (a_{k,j-m_1} * valor)
        Fim-para
    Senão
        Para cada neurônio N_k ∈ p_i tal que grupo_k = 1 faça
            soma_k ← soma_k + (a_{j,k-m_1} * valor)
        Fim-para
    Fim-se
```

Fig. 3.11 - Algoritmo do procedimento **atualizar.parcial**.

Capítulo IV

4. Resultados Obtidos

Uma importante medida de desempenho de um sistema distribuído é o fator *speedup S* associado a um problema em particular. O *speedup* é definido como a razão do tempo requerido para a conclusão de um determinado processamento em um único nó computacional, pelo processamento equivalente executado em um processador concorrente. Esta medida nem sempre é facilmente calculada, visto que, por exemplo, pode não ser possível, na prática, acomodar o problema inteiro na memória de um único nó de processamento. Tecnicamente, é mais fácil prover as máquinas concorrentes com memória total grande do que prover as máquinas seqüenciais, de modo que problemas que exigem muita memória, geralmente não podem ser solucionados por máquinas seqüenciais. O interesse de se definir *speedup* não é meramente acadêmico, mas sim uma manifestação de uma característica interessante das máquinas concorrentes, especialmente aquelas com memória distribuída.

Dado que $T_{conc}(N)$ é definido como o tempo despendido por um processador concorrente com N nós, denotamos o tempo de um processador seqüencial como sendo

$$T_{seq} = T_{conc}(1) \qquad \textbf{(1)}$$

Segue-se que o *speedup S* depende de N, o número de nós, e é dado por

$$S(N) = \frac{T_{seq}}{T_{conc}(N)} \qquad \textbf{(2)}$$

Se T_{seq} não pode ser diretamente medido, então, geralmente, é calculado através da execução de um problema pequeno na máquina seqüencial e em seguida extrapolando ao problema real através da utilização da dependência de predição ou medida no tamanho do problema.

Algumas vezes, é de grande utilidade a introdução da eficiência concorrente ε definida por

$$\varepsilon = \frac{S}{N} \qquad \textbf{(3)}$$

Os possíveis motivos para se reduzir o *speedup S* de seu valor ideal N são os seguintes:

- Algoritmo não-ótimo ou *overhead* algorítmico: pode não ser possível encontrar um algoritmo para a máquina concorrente que seja tão eficiente quanto o algoritmo seqüencial.

- *Overhead* de *software*: mesmo com um algoritmo completamente equivalente, existe um *overhead* de *software* devido a implementação. Por exemplo, pode ser necessário o uso de cálculos de índices adicionais devido a forma com que os dados são distribuídos pelos processadores.

- Balanceamento de carga: o *speedup* geralmente é limitado pela velocidade do nó mais lento. Assim, uma consideração importante é assegurar que cada nó execute a mesma quantidade de trabalho.

- *Overhead* de comunicação: qualquer tempo despendido com comunicação, constitui um prejuízo no desempenho geral em comparação com o caso seqüencial.

No nosso caso, o custo de comunicação constitui o problema dominante em relação a eficiência, pois o número de mensagens trocadas durante a simulação é muito grande. O NCP I permite nenhuma sobreposição de processamento e comunicação, isto é, se um processo deseja enviar uma mensagem para um processo localizado em outro processador, o primeiro só poderá continuar o seu processamento quando a comunicação estiver concluída. Portanto, qualquer tempo despendido em comunicação degrada diretamente o *speedup*.

Quando cálculo e comunicação não podem ser sobrepostos, o efeito do *overhead* de comunicação no *speedup* pode ser escrito como

$$T_{conc} = \frac{T_{seq}}{N}\left(1+f_C\right) \quad \text{ou} \tag{4}$$

$$S = \frac{N}{1+f_C} \quad \text{e} \tag{5}$$

$$\varepsilon = \frac{1}{1+f_C} \approx 1-f_C \tag{6}$$

onde f_C representa o *overhead* de comunicação fracionado podendo ser definido como

$$f_C = \frac{t_{com}}{t_{calc}} \tag{7}$$

onde t_{calc} é o tempo padrão requerido para executar um cálculo genérico e t_{com} é o tempo padrão para comunicar uma única palavra entre dois nós conectados. Estas duas medidas são parâmetros do *hardware*. O resultado final em (6) só é válido quando o *overhead* f_C for pequeno comparado à unidade.

Nas seções seguintes apresentaremos alguns problemas que foram utilizados para testar e avaliar o desempenho dos simuladores.

4.1. Memória Associativa

Memória associativa é uma memória cujo endereçamento é feito através do seu conteúdo ao invés da sua posição. Suponha que "Anderson, J. A. (1977) *Psych. Rev.* 84, 413 - 451" é um item armazenado na memória. Uma memória associativa seria capaz de recuperar inteiramente este item de memória baseando-se em suficiente informação parcial, que poderia ser, por exemplo, "Anderson, (1977)". Uma memória ideal poderia lidar com erros e recuperar esta referência mesmo com a entrada "Andyrson, (1977)". Apenas formas relativamente simples de memória associativa têm sido feitas em *hardware*, e sofisticadas idéias, como correção de erro no acesso a informação, são usualmente introduzidas como *software*.

Considere um sistema físico descrito pelas coordenadas $X_1, ..., X_n$ que compõem um vetor de estados X. Dado que o sistema possui pontos locais estáveis $X_a, X_b, ...$, então se iniciamos o sistema em um ponto suficientemente perto de qualquer X_a, como em $X = X_a + \Delta$, ele prosseguirá no tempo até $X \approx X_a$. Podemos considerar a informação armazenada no sistema como sendo os vetores $X_a, X_b, ...$. O ponto $X = X_a + \Delta$ representa o conhecimento parcial do item X_a, e o sistema então gera a informação total X_a.

Qualquer sistema físico cuja dinâmica no aspecto espacial é dominada por um número substancial de estados locais estáveis aos quais é atraído, pode ser considerado como uma memória endereçável por conteúdo.

As rede neuronais binária e contínua de Hopfield são sistema considerados memórias associativas, pois eles convergem para pontos locais estáveis. Os padrões considerados pontos estáveis são armazenados nas redes neuronais através dos pesos das sinapses. Suponha que desejamos armazenar o conjunto de estados $\mathbf{V}^s$, $s = 1, ..., m$. Utilizamos a fórmula de armazenamento

$$w_{ij} = \sum_s (2V_i^s - 1)(2V_j^s - 1) \qquad \textbf{(8)}$$

onde $w_{ii} = 0$. Tanto na rede neuronal binária quanto na contínua, geramos os padrões aleatoriamente, sendo que na primeira os padrões são compostos por 0's e 1's e na última são compostos por números reais no intervalo [0, 1].

4.1.1. Rede Neuronal Binária de Hopfield

Dado que n é o número de neurônios da rede, constatamos experimentalmente que por volta de $0.15n$ estados podem ser simultaneamente lembrados sem causar erros de recuperação muito severos.

A probabilidade de que o estado final seja o mais aproximado do estado inicial, foi estudada como uma função da distância entre o estado inicial e o estado de memória mais próximo. Em uma rede em que $n = 30$ e $m = 5$, para distâncias ≤ 5, alcança-se o estado mais próximo em mais de 90% das vezes. Acima desta distância, a probabilidade cai uniformemente, abaixando para um nível de 0.2 para uma distância de 12.

O fluxo espacial é dominado, aparentemente, por "atrativos" que são os padrões de memória, onde cada um domina uma região substancial a sua volta. O fluxo não é inteiramente determinístico e o sistema responde a um estado inicial ambíguo através de uma escolha estatística entre os estados de memória que mais se assemelham.

O estado 0000... é sempre estável. Para um limiar igual a 0, este estado estável tem muito mais energia do que os estados armazenados e ocorre muito raramente. Adicionar um limiar uniforme ao algoritmo, equivale a aumentar a energia efetiva das memórias armazenadas, comparado ao estado 0000, e este último se torna um estado provável. O estado 0000 então é gerado a partir de qualquer estado inicial que não se assemelha o bastante de nenhum dos padrões, representando o reconhecimento de que o estado inicial não é familiar ao sistema.

A equação (8) para o armazenamento dos padrões na rede, gerou grafos praticamente totalmente conexos, e, como já mencionamos, nestas condições, a eficiência do sistema distribuído é quase nenhuma. Isto ocorre porque em um grafo direcionado deste tipo , temos apenas um nó *sink*, e a cada reversão das arestas do nó *sink*, forma-se um único *sink*, tornando o algoritmo basicamente seqüencial. Logo, o sistema apresenta, para este problema, um *speedup* menor que 1, pois em adição ao algoritmo se tornar seqüencial, temos todos os encargos gerados pelo paralelismo.

4.1.2. Rede Neuronal Contínua de Hopfield

Ao utilizarmos a rede neuronal contínua de Hopfield como memória associativa, constatamos que o armazenamento dos padrões aleatórios no sistema através da equação (8), gera um grafo totalmente conexo. Por este motivo, quando um neurônio é atualizado, seu novo valor deve ser enviado para todos os processadores. O processo de comunicação oferece uma ferramenta de *broadcast*, que forma uma árvore cuja raiz é o nó de processamento gerador do *broadcast* e os nós são os outros processadores. Esta árvore indica o caminho por onde a mensagem passa tentando minimizar o número de cópias criadas da mensagem a ser distribuída.

Mesmo com esta facilidade, o número de mensagens trocadas é muito grande e para um processador atualizar seus neurônios é necessário receber os valores de todos os outros neurônios da rede. Portanto o *speedup* do sistema é bem baixo, como demonstra o gráfico da figura 4.1. Isto ocasionou, também, um tempo de processamento muito grande, e alguns testes demoraram cerca de 4 horas ou mais em uma rede de 8

processadores, dificultando a execução dos testes em redes de apenas 2 ou 4 processadores, pois o tempo seria ainda maior. Na figura 4.1, apresentamos então o *speedup* produzido por um sistema de 8 processadores com exemplos de redes compostas por 100 a 1000 neurônios.

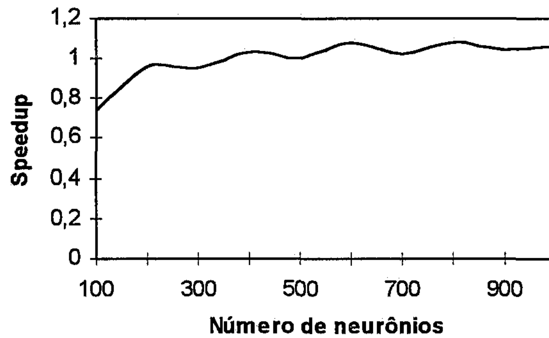


Fig. 4.1 - *Speedup* produzido pela execução do simulador em um sistema de 8 processadores.

4.2. Problema de Cobrimento Mínimo de Vértices (CMV)

Considerando-se um grafo não-direcionado G = (**V**, **A**) onde **V** é o conjunto de vértices e **A** representa o conjunto de arestas, o problema de Cobrimento Mínimo de Vértices (CMV) envolve a procura pelo menor subconjunto $\mathbf{S} \subseteq \mathbf{V}$ tal que pelo menos um dos dois vértices de cada aresta em **A** esteja em **S**. Com o objetivo de formular o problema CMV como um problema de Otimização, definimos $n = |\mathbf{V}|$ associando cada variável em **X** a um vértice em **V**, o domínio $\mathbf{D} = \{0, 1\}$ e f como sendo:

$$f(x) = \sum_{v_i \in \mathbf{V}} X_i + \sum_{(v_i, v_j) \in \mathbf{A}} \begin{cases} 2, & \text{se } X_i = X_j = 0; \\ 0, & \text{em qualquer outro caso} \end{cases} \quad (9)$$

Deve-se notar que um ponto $x \in \mathbf{D}^n$ minimiza a função f se e somente se neste ponto todas as variáveis que tem valor 1 correspondem ao conjunto de nós que forma o cobrimento mínimo de vértices de G.

4.2.1. Máquina de Boltzmann

O problema CMV quando formulado como um problema de decisão, pertence a classe NP-completo. Além disso, ele pertence à classe de problemas que podem ser

solucionados através da utilização de uma Máquina de Boltzmann, visto que **X** forma um Campo Aleatório de Markov de acordo com as definições descritas na seção 2.2.2.1.

Sendo assim, o problema de cobrimento mínimo de vértices pode ter sua solução aproximada através do algoritmo descrito na seção 2.2.2.3, onde as probabilidades condicionais da equação (13) do capítulo II, tomam a seguinte forma

$$\begin{aligned} \text{Prob}(X_i = 0 | X_j = x_j, j \neq i) &= \frac{1}{1+e^{-(1-2z_i)/T_k}} \\ \text{Prob}(X_i = 1 | X_j = x_j, j \neq i) &= \frac{e^{-(1-2z_i)/T_k}}{1+e^{-(1-2z_i)/T_k}} \end{aligned} \qquad \textbf{(10)}$$

onde $x_j \in \{0, 1\}$, z_i corresponde ao número de vizinhos de X_i cujos valores são iguais a 0 e T_k é a temperatura na k-ésima vez em que X_i é atualizado.

Portanto, para se resolver este problema através da aplicação de uma Máquina de Boltzmann, basta utilizar o algoritmo descrito na seção 2.2.2.3 onde as probabilidades de atualização dos neurônios são as apresentadas na equação (10).

A diminuição gradativa do parâmetro de tempo T de $T_{inicial}$ até T_{final} é dado pela equação

$$T = \alpha T \qquad \textbf{(11)}$$

Como resultado tivemos os *speedups* apresentados nas figuras 4.2 e 4.3 que utilizam respectivamente $\alpha = 0.5$ e $\alpha = 0.9$. Deve-se notar que para valores maiores de α o número de iterações executados é maior, correspondendo a um maior *speedup*. Além disso, o *speedup* só se torna realmente significativo quando utilizamos 8 processadores.

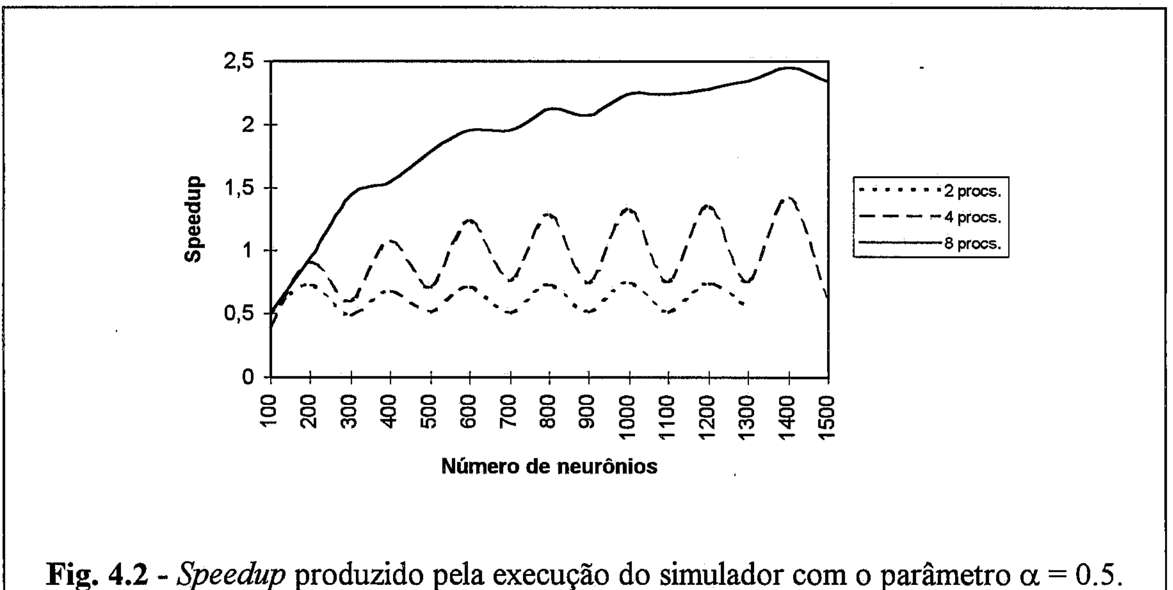


Fig. 4.2 - *Speedup* produzido pela execução do simulador com o parâmetro $\alpha = 0.5$.

Os exemplos de grafos utilizados como teste eram de baixa densidade (menor que 0.5) e foi constatado que quanto maior o número de subgrafos isolados, melhor é o desempenho do simulador. A explicação desta observação se baseia no fato de que, tendo um maior número de subgrafos isolados, o processo tem mais processamento interno a executar enquanto espera pelas mensagens necessárias.

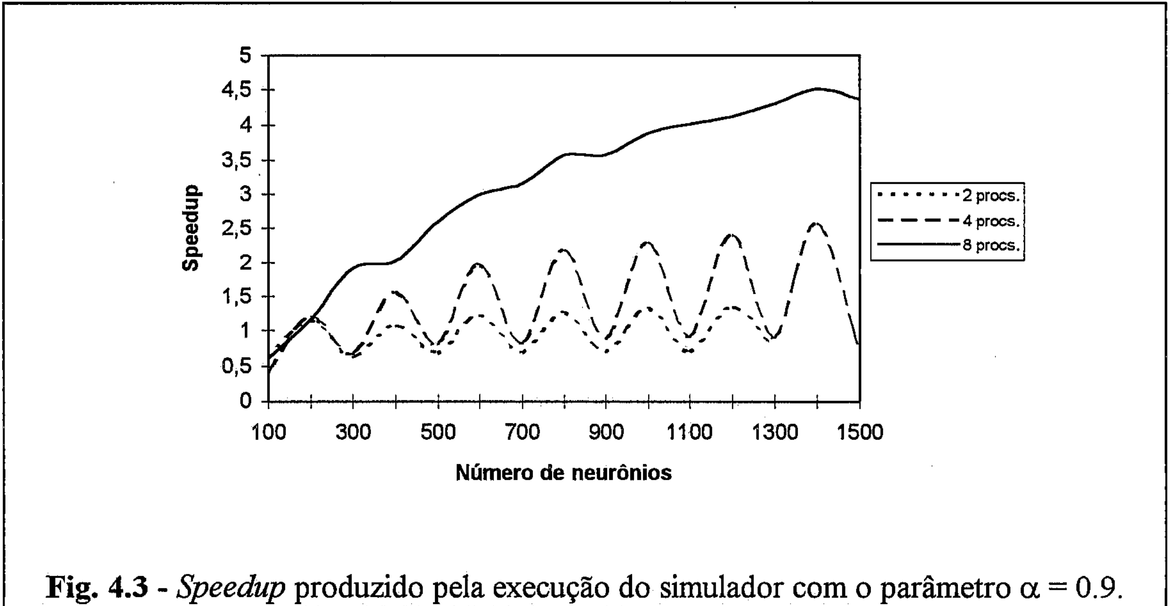


Fig. 4.3 - *Speedup* produzido pela execução do simulador com o parâmetro $\alpha = 0.9$.

Observando as figuras 4.2 e 4.3, verifica-se uma certa regularidade na variação do *speedup* quando utilizamos 4 e 2 processadores. Esta variação ocorreu devido à particularidade dos grafos de entrada e à alocação dos nós nos processadores. Para obter *speedups* razoáveis, foi necessário utilizar grafos bem esparsos, e para tanto desenvolvemos um gerador que constroi grafos a partir do número de nós e da densidade desejada. O gerador interliga os nós de forma que quando são alocados em uma rede de 8 processadores, cada processador contenha pelo menos um subgrafo isolado. Temos então dois casos: quando alocamos o mesmo número de nós em cada processador (o número de nós é divisível por 8), e quando alocamos diferentes números de nós em cada processador (o número de nós não é divisível por 8). No segundo caso, alguns processadores têm apenas um nó a mais do que os outros, pois a alocação tenta alocar o mesmo número de nós para cada um, mas quando isto não é possível, tenta-se diminuir ao máximo esta diferença.

Quando todos os 8 processadores são responsáveis pelo mesmo número de nós do grafo, ao alocarmos o mesmo grafo em uma rede de 4 processadores, continuamos tendo pelo menos um subgrafo isolado em cada processador e o mesmo número de nós. Isto é garantido porque as partes do grafo armazenadas em cada 2 processadores na rede de 8 processadores, são agrupadas em 1 processador na rede de 4 processadores (ver figura 4.4).

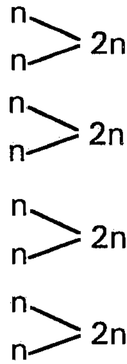


Fig. 4.4 - Cada processador na rede de 8 processadores possui n nós do grafo, e na rede de 4 processadores cada um possui 2n nós.

No caso em que o número total de nós do grafo não é divisível por 8, ao alocarmos o grafo na rede de 4 processadores, tenta-se sempre alocar o mesmo número de nós em cada um, de forma que é possível alocar um nó que fazia parte de um subgrafo isolado (na rede de 8 processadores), em um processador diferente de seus vizinhos. Como conseqüência, há um aumento do número de comunicações entre processadores em relação ao outro caso. Na figura 4.5 temos um exemplo desta situação.

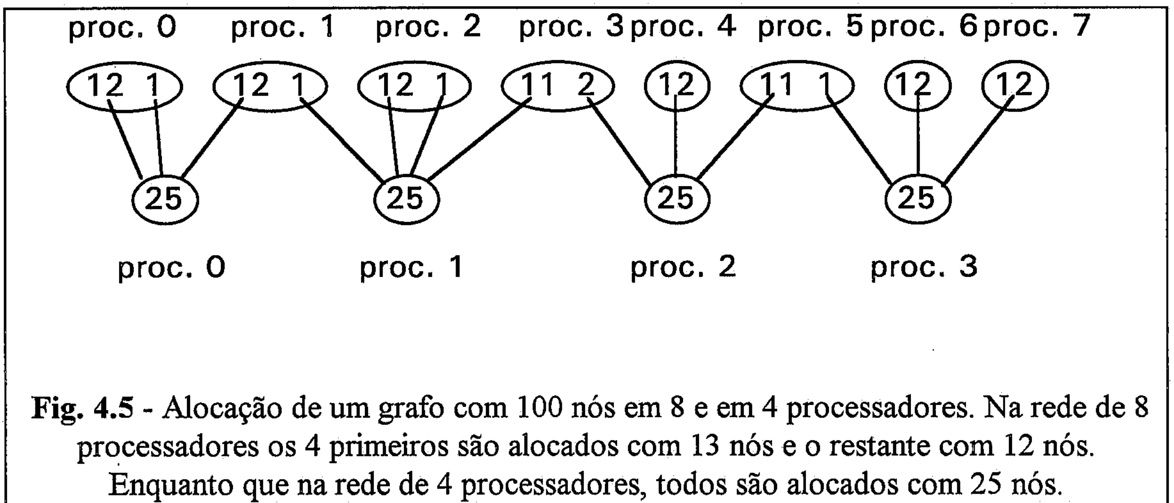


Fig. 4.5 - Alocação de um grafo com 100 nós em 8 e em 4 processadores. Na rede de 8 processadores os 4 primeiros são alocados com 13 nós e o restante com 12 nós. Enquanto que na rede de 4 processadores, todos são alocados com 25 nós.

Neste exemplo notamos que um nó alocado no processador 1 é separado dos outros 12 nós ao ser alocado na rede de 4 processadores. Assim como 2 nós são separados dos outros 11 nós do processador 3. Estas separações geram dependências entre processadores que não existiriam se o número de nós fosse divisível por 8.

Portanto, agora não se garante que sempre existirá pelo menos um subgrafo isolado em cada processador.

Estas duas possibilidades são representadas nos gráficos pela oscilação do *speedup* ao utilizarmos grafos com número de nós divisíveis por 8 (200, 400, 600, 800, ...) obtendo melhores *speedups*, e ao empregarmos grafos com número de nós não divisíveis por 8 (100, 300, 500, 700, ...) obtendo *speedups* menores.

4.2.2. Rede Neuronal Binária de Hopfield

Na discussão a seguir denominaremos P_0 e P_1 como as respectivas probabilidades condicionais apresentadas em (10). A medida que T_k se aproxima de 0 temos duas possibilidades: se z_i for insignificantemente pequeno, P_0 se aproxima de 1 e P_1 se aproxima de 0; ou se z_i é significativo, P_0 se aproxima de 0 e P_1 se aproxima de 1. Tendo em vista a aplicação de um modelo semelhante à Rede Neuronal de Hopfield Binária, usaremos esta simplificação das probabilidades para gerar a seguinte regra de atualização das variáveis:

$$X_i = \begin{cases} 0, \text{se } X_i = 1 \text{ e } z_i = 0 \\ 1, \text{se } X_i = 0 \text{ e } z_i > 0 \\ \text{constante, em qualquer outro caso} \end{cases} \quad \textbf{(12)}$$

Esta variação do *Simulated Annealing* pode ser implementada distribuidamente associando um processo p_i para cada variável $X_i \in \mathbf{X}$. Particularmente para o problema CMV, associamos cada vértice em **V** a um processo, e cada aresta em **A** a um canal de comunicação. Trataremos cada variável X_i como um neurônio binário N_i que é atualizado pelo processo p_i de acordo com a regra da equação (12). Os neurônios que não são vizinhos, isto é, os vértices que não são conectados por arestas, podem ser atualizados concorrentemente. O procedimento executado por cada processo consiste em atualizar o valor de seu neurônio baseado nos valores dos neurônios vizinhos.

A adaptação deste problema a Rede Neuronal Binária de Hopfield é bem simples. As únicas diferenças entre este modelo e o da rede são que as sinapses aqui não têm pesos, elas apenas existem ou não; e quanto a regra de atualização dos neurônios, que obedecem as regras em (12). Assim como na Rede de Hopfield, só teremos um ganho com a paralelização deste método, se o grafo G for esparso.

Na figura 4.6 demostramos o comportamento do simulador frente ao problema em questão. Comparando-se esta figura às figuras 4.2 e 4.3, verificamos a semelhança no formato dos gráficos, demostrando que para este problema a utilização de 8 ou mais processadores é essencial. Os mesmos testes que foram usados na máquina de Boltzmann

são empregados na geração dos resultados da figura 4.6. e, devido a semelhança dos gráficos, as observações levantadas na seção anterior valem para este caso também.

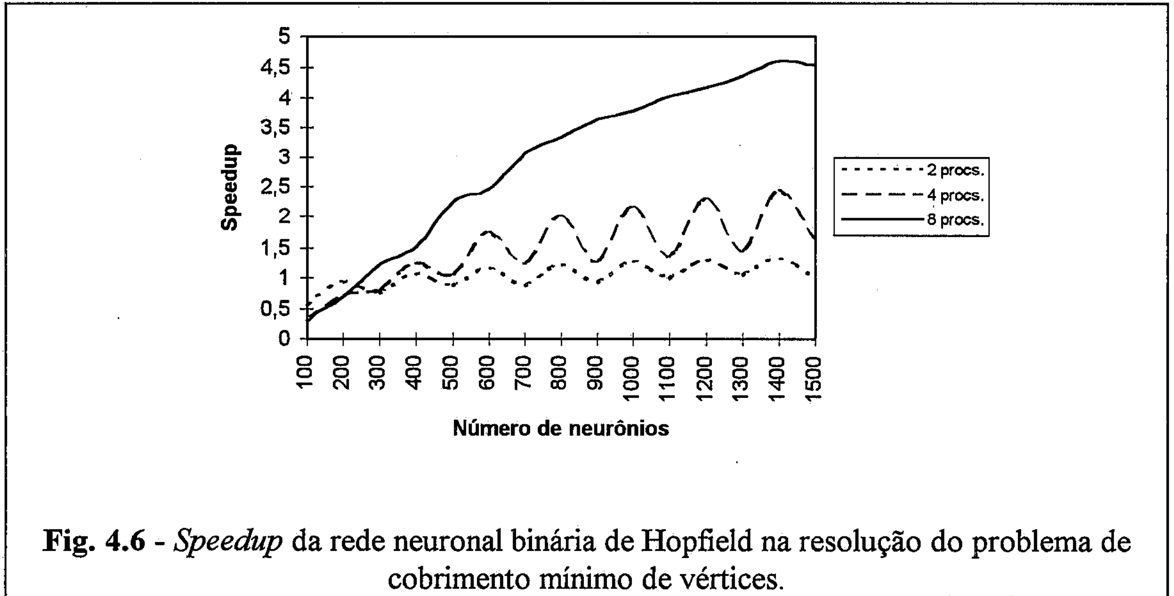


Fig. 4.6 - *Speedup* da rede neuronal binária de Hopfield na resolução do problema de cobrimento mínimo de vértices.

4.3. Resolução de Ambigüidade Léxica

A resolução de ambigüidade léxica tem um papel de grande importância no processamento de linguagem natural, podendo ser classificada em **ambigüidade sintática e ambigüidade semântica**. A primeira se refere a ambigüidade gramatical ocasionada pelo fato de uma sentença poder ser gramaticalmente analisada em diferentes maneiras. A ambigüidade semântica ocorre devido aos diferentes significados que uma mesma palavra pode ter.

Com o objetivo de resolver este problema, partimos do princípio que um processo de desambiguação se resume em resolver incertezas nos domínios sintático e semântico, e, por sua vez, estas incertezas são melhor representadas por um modelo probabilístico. Esta adaptação do problema ao modelo é descrita detalhadamente em [11] e aqui apresentaremos apenas um resumo.

O modelo probabilístico utilizado é o modelo de uma rede Bayesiana descrita na seção 2.3. O modelo proposto permite executar paralelamente as duas funções:

- análise sintática, envolvendo uma ou mais árvores sintáticas (*parsed trees*) igualmente prováveis;
- análise semântica, baseada em uma representação de conhecimento que tenta cobrir os casos requeridos para cada significado do *verbo*.

Desta forma, quando a análise sintática resulta em uma decisão ambígua, a análise semântica ajuda na decisão por uma das árvores analíticas em questão dependendo dos casos que podem ser preenchidos.

A rede Bayesiana para a desambiguação léxica é composta por duas sub-redes interconectadas: uma sub-rede para análise sintática, gerada a partir de uma gramática livre de contexto, e uma sub-rede para análise semântica, que pode ser construída a partir de um conjunto de palavras e suas possíveis categorias gramaticais. Estas duas sub-redes são então interconectadas através de um procedimento automático.

A construção da sub-rede sintática para uma gramática livre de contexto G pode ser automaticamente realizada através dos passos descritos logo abaixo. Assumimos que a gramática tem como símbolo inicial S, não contém recursão em S e que l representa o tamanho máximo que uma sentença pode ter. As produções da gramática têm o formato $N \rightarrow w$, onde N é um símbolo não-terminal e w é uma string composta de um ou mais símbolos terminais e/ou não-terminais. Além disso, utilizamos as definições de **nó-OR** e **nó-AND**. Um nó X_i em que $P(X_i = 1 \mid x_j, X_j \in \mathbf{P}(X_i))$ tem um valor baixo exceto nas seguintes configurações de $|\mathbf{P}(X_i)|$ (1, 0, ..., 0), (0, 1, ..., 0), ..., (0, 0, ..., 1), é um nó-OR. Analogamente, um nó X_i em que $P(X_i = 1 \mid x_j, X_j \in \mathbf{P}(X_i))$ tem um valor baixo exceto na configuração (1, 1, ..., 1) de seus pais, é um nó-AND. Idealmente estes "valores baixos" são iguais a zero, mas devido a problemas de convergência, utilizamos valores próximos de zero.

1. Eliminar todas as produções-ϵ (produções em que temos um símbolo não-terminal no lado esquerdo e uma string vazia no lado direito).

2. Criar l grupos de nós, onde cada um contém um nó para cada símbolo terminal da gramática.

3. Para $2 \leq k \leq l$, criar um nó-AND para cada produção que corresponde a uma seqüência de k símbolos terminais, e fazer com que os seus pais sejam os nós criados anteriormente que correspondem a símbolos do lado direito da produção. Se mais de uma produção reescreve o mesmo não-terminal e este não-terminal não é S, criar um nó-OR para englobar isto, tornando seus pais os nós que correspondem a estas produções.

4. Repetidamente, eliminar todos os nós sem filhos, com exceção daqueles relacionados a S. Este passo eventualmente, elimina nós de entrada que correspondem a categorias que não podem ocorrer em um determinado grupo.

A complexidade deste algoritmo é uma função do número de símbolos em G, do número de produções, do número de símbolos da maior produção e l. Este procedimento é impraticável para valores arbitrários de l. Entretanto, ele pode ser aplicável se, dado G, l for um número modestamente pequeno.

A geração da sub-rede destinada à análise semântica (descrita em [11]) é mais complexa e por não estar no escopo deste trabalho, não a apresentamos aqui. O exemplo de rede Bayesiana utilizado para medidas de *speedup*, foi retirado de [12] e é constituída de 150 nós. Na tabela da figura 4.7 mostramos os *speedups* obtidos.

n. de iterações	2 procs.	4 procs.	8 procs.
300	0.830	0.744	0.914
600	0.849	0.747	0.917
1000	0.851	0.746	0.921

Fig. 4.7 - *Speedups* de uma rede Bayesiana utilizada no problema de disambiguação léxica

De acordo com a tabela, verificamos que o *speedup* produzido por 2 processadores é maior do que o gerado por 4 processadores. Isto é devido a topologia da rede em questão. Este exemplo é formado por dois subgrafos desconectados e quando eles são alocados em 2 processadores, um dos subgrafos é inteiramente alocado em um único processador gerando um subgrafo isolado. Logo, parte do processamento interno de um processador independe do outro, não necessitando esperar por mensagens para atualizar parte de seus nós. Ao dividirmos a rede em 4 e 8 processadores, esta característica desaparece.

Todos os *speedups* encontrados são abaixo de 1, devido principalmente ao pequeno tamanho da rede, pois ao distribuirmos a rede pelos 8 processadores, cada um se torna responsável apenas por 18 ou 19 nós e, portanto, com um processamento interno muito baixo em comparação ao número de mensagens trocadas.

4.4. Problema do Caixeiro Viajante

Dado m cidades e as distâncias d_{AB} entre quaisquer duas cidades A e B, o problema do caixeiro viajante consiste em encontrar o menor percurso que passe por todas as cidades uma única vez. Utilizamos uma rede neuronal contínua de Hopfield com o objetivo de encontrar uma solução aproximada do problema.

A adaptação do problema à rede neuronal usa uma rede de m^2 neurônios organizados no formato de uma matriz, onde cada linha corresponde a uma cidade e cada coluna corresponde a uma das possíveis posições que uma cidade pode ocupar na rota das cidades. Um possível estado estável desta rede é caracterizado por exatamente m neurônios no estado ligado, isto é, aproximadamente 1, espalhados pela matriz de forma que exista apenas um deles por linha e um deles por coluna.

Esta definição de um estado estável é embutida na rede através da determinação dos pesos das sinapses. Sejam N_i e N_j dois neurônios da rede. Se eles ocupam a mesma linha ou coluna da matriz, então eles são interligados por uma sinapse inibitória $w_{ij} = W < 0$. Se eles estão nas linhas correspondentes às cidades A e B respectivamente, e eles ainda ocupam colunas adjacentes, onde, por definição, a coluna 1 é adjacente à coluna m, então N_i e N_j são interconectados pela sinapse inibitória $w_{ij} = -d_{AB}$. Todas as outras sinapses da rede têm peso 0, isto é, quaisquer outros pares de neurônios não são interligados.

Para garantirmos que todo estado estável do sistema é possível utilizamos

$$W < \frac{\theta}{R} < \min_{A}\left\{-2\sum_{B \neq A} d_{AB}\right\} \tag{13}$$

A prova desta proposição está descrita em [13].

A coleta de dados foi realizada através da utilização de exemplos ondes as distâncias entre as cidades foram escolhidas aleatoriamente e limitadas por um quadrado de dimensão 1. A alocação dos neurônios foi realizada por linha da matriz que eles compõem. Sendo assim, em um exemplo com 10 cidades, teremos uma matriz 10x10, onde os neurônios da primeira linha são N_1, ..., N_{10}, na segunda linha teremos N_{11}, ..., N_{20}, ..., e a última linha é formado por N_{91}, ..., N_{100}.

Tanto na alocação por linha quanto por coluna, cada processador contém neurônios que, no conjunto, precisam dos valores de todos os outros neurônios. Isto é constatado observando a topologia na qual são conectados.

O gráfico resultante é mostrado na figura 4.8. O *speedup* referente a 35 cidades não é realístico, pois foi necessário desenvolver um algoritmo seqüencial menos eficiente para que fosse possível armazenar os 35^2 neurônios no espaço de memória de um único processador. Tendo isto em vista, verificamos que em nenhum momento foi alcançado um *speedup* ≥ 1. Este fato ocorre principalmente devido ao grande número de trocas de mensagens.

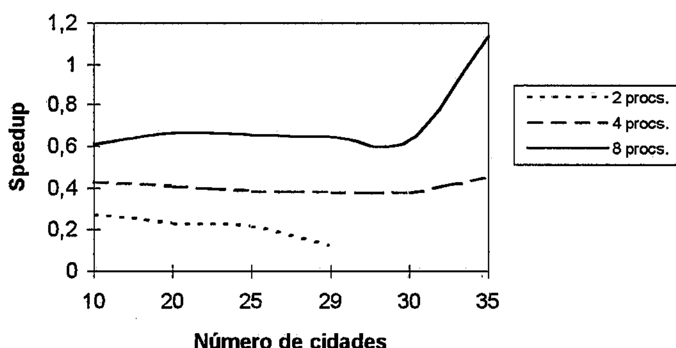


Fig. 4.8 - *Speedup* da rede neuronal contínua de Hopfield na resolução do problema do caixeiro viajante.

4.5. Resolução de Sistemas Lineares

Como teste para este simulador, utilizamos sistemas lineares compostos por equações cujos coeficientes foram gerados aleatoriamente e cada componente b_i é o resultado do somatório dos coeficientes da equação correspondente. Temos então como resultado dos sistemas vetores x's cujos elementos são iguais a 1.

Na figura 4.9 mostramos o *speedup* produzido pela resolução de sistemas quadráticos cuja dimensão (que corresponde a dimensão da matriz A) varia de 200x200 a 1000x1000.

Na figura 4.10 apresentamos o resultado da resolução de sistemas retangulares cuja primeira dimensão é igual a 1000 e a segunda varia de 100 a 900. Observando esta figura, verificamos que os melhores *speedups* ocorrem em sistemas lineares na faixa de dimensão 1000x400 a 1000x600. Nos sistemas de dimensões 1000x100 a 1000x300, o *speedup* é menor porque mais que 3/4 dos processadores são alocados com neurônios do mesmo tipo e, portanto, teremos um número de trocas de mensagens muito grande. Estudando a alocação dos neurônios, notamos que na faixa de maior *speedup*, existe sempre um processador que se comunica apenas com seus vizinhos diretos. Isto é tão marcante, que ao verificarmos a alocação do sistema linear dimensionado em 1000x700 (o causador de uma queda brusca do *speedup*), confirmamos que nesta alocação todos os processadores se comunicam indiretamente com pelo menos um processador. Concluímos então que este fato supera o tempo despendido com as mensagens adicionais geradas pelo sistema retangular.

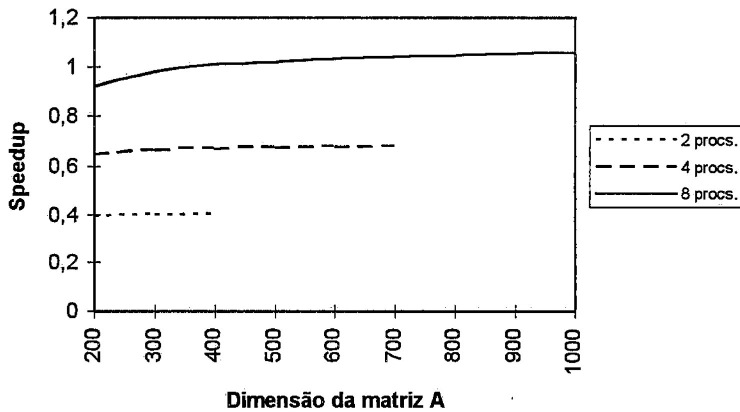


Fig. 4.9 - *Speedup* produzido pela resolução de sistemas lineares quadráticos.

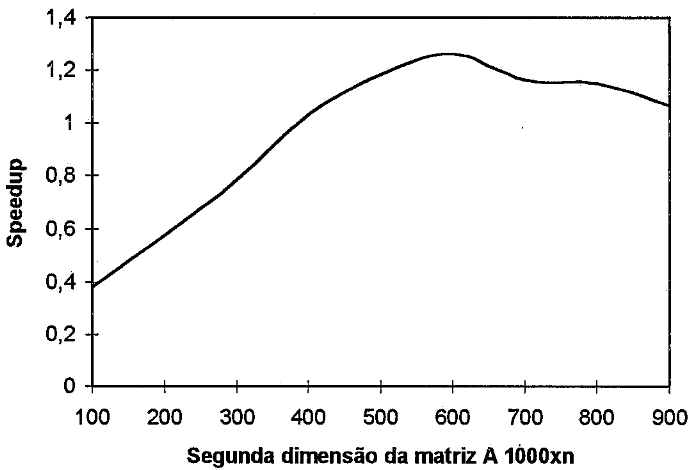


Fig. 4.10 - *Speedup* produzido pela resolução de sistemas retangulares. O eixo x do gráfico representa a segunda dimensão da matriz *A* cuja primeira dimensão é 1000.

4.6. Observações Finais

Tendo em vista as similaridades existentes entre os modelos, inicialmente tínhamos como objetivo o desenvolvimento de uma ferramenta que unificasse todos os modelos. Devido a forma como estes foram implementados, isto não foi possível, pois para cada modelo foi necessário utilizar estruturas de dados diferentes e o núcleo de cada simulador (processo *principal*) opera de forma distinta. Em adição, a linguagem OCCAM não

fornecia nenhuma forma de alocação dinâmica de memória de forma que dependendo do modelo a ser simulado alocaríamos uma determinada estrutura de dados. Como não foi possível esta generalização, foi necessário desenvolver um programa para cada modelo, e a execução destes programas é controlada por um programa de *interface* com o usuário.

A experiência com a linguagem OCCAM foi muito construtiva e facilitou consideravelmente a implementação do simulador. Por ter sido desenvolvida com o objetivo de ser uma linguagem paralela, ela oferece todas as ferramentas necessárias à programação concorrente. Portanto, a linguagem OCCAM supera as outras que originalmente são voltadas para programação seqüencial e, com o aparecimento dos sistemas distribuídos, foram acrescidas de ferramentas indispensáveis à programação paralela.

As maiores dificuldades encontradas se referem ao ambiente de programação TDS. A sua ferramenta de depuração ainda é rudimentar e o compilador apresenta alguns *bugs* a serem eliminados.

Tendo em vista os resultados apresentados, observamos que o maior custo do sistema se relaciona a comunicação. Observamos que quanto menor o número de mensagens trocadas melhor é o *speedup* do sistema. Além disso, quanto menos mensagens são roteadas, melhor é a eficiência. Isto demonstra a necessidade de haver uma ferramenta de roteamento embutida no *hardware* ao se desenvolver sistemas que exigem muita comunicação entre os processadores.

A máquina paralela NCP I é de granularidade grossa, isto é, o número de nós da rede de processadores é pequeno de forma que é necessário haver uma quantidade muito grande de processamento em cada nó para superar o custo de comunicação. Este problema está espelhado nos resultados obtidos, onde na maioria dos casos os *speedups* foram baixos. A solução para este problema seria o acoplamento de um processador de comunicação, já citado, ou o aumento da memória dos nós de processamento possibilitando a execução de problemas maiores.

Referências Bibliográficas

[1] C. L. Amorim, R. Citro, A. F. Souza, e E. M. Chaves Filho, *The NCP I Parallel Computer System*, Publicações Técnicas, Programa de Engenharia de Sistemas e de Computação, COPPE, Universidade Federal do Rio de Janeiro, 1991.

[2] Inmos Ltd., *Transputer Development System*, Prentice-Hall, 1988.

[3] A. Burns, *Programming in OCCAM 2*, Addison-Wesley Pub. Co., 1988.

[4] J. Galletly, *OCCAM 2*, Pitman Pub., 1990.

[5] L. M. A. Drumond, *Projeto e Implementação de um Processador Virtual de Comunicação*, Tese de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, 1990.

[6] V. C. Barbosa, e P. M. V. Lima, *On the Distributed Parallel Simulation of Hopfield's Neural Networks*, Software–Practice and Experience, Vol. 20(10), pp. 967-983, 1990.

[7] V. C. Barbosa, e E. Gafni, *Concorrency in Heavily Loaded Neighborhood-Constrained Systems*, ACM Trans. Programming Languages and Systems, Vol. 11, pp. 562-584, 1989.

[8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, e E. Teller, *Equations of State Calculations by Fast Computing Machines*, Journal of Chemical Physics, Vol. 21, pp. 1087-1091, 1953.

[9] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann Pub., Inc., San Mateo, Califórnia.

[10] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley Pub. Co., 1973.

[11] L. M. R. Eizirik, V. C. Barbosa, e S. B. T. Mendes, *A Bayesian-Network Approach to Lexical Disambiguation*, Cognitive Science, Vol. 17, pp. 257-283, Abril 1993.

[12] E. P. Alves, *Uma Aplicação em Redes Semânticas Utilizando Redes Bayesianas*, Tese de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, 1993.

[13]L. A. V. Carvalho, e V. C. Barbosa, *A TSP Objective Function that Ensures Feasibility at Stable Points*, Proceedings International Neural Network Conference, Vol. 1, pp. 249-253, Paris, Julho 1990

[14]M. Maekawa, A. E. Oldehoeft, e R. R. Oldehoeft, *Operating Systems - Advanced Concepts*, The Benjamin/Cummings Pub. Co., Inc., 1987.

[15]G.Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, e D. Walker, *Solving Problems on Concorrent Processors - General Techniques and Regular Problems*, Vol. 1, Prentice-Hall, 1988.

[16]C. L. Amorim, V. C. Barbosa, e E. S. T. Fernandes, *Uma Introdução à Computação Paralela e Distribuída*, VI Escola de Computação, Campinas, 1988.

[17]G. Josin, *Neural-Network Heuristics*, Byte Magazine, pp. 183-192, Outubro de 1987.

[18]R. L. Rivest, e W. Remmele, *Machine Learning: The Human Connection*, Siemens Review, Fevereiro de 1988.

[19]J. J. Hopfield, e D. W. Tank, *Computing with Neural Circuits: A Model*, Science, Vol. 233, pp. 625-633, 1986.

[20]J. J. Hopfield, *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proc. Natl. Acad. Sci. USA, Vol. 79, pp. 2554-2558, 1982.

[21]J. J. Hopfield, *Neurons with Graded Response have Collective Computational Properties like those of Two-state Neurons*, Proc. Natl. Acad. Sci. USA, Vol. 81, pp. 3088-3092, 1984.

[22]L. A. V. Carvalho, e V. C. Barbosa, *Fast Linear System Solution by Neural Networks*, COPPE, Universidade Federal do Rio de Janeiro, Caixa Postal 68503, 21945 Rio de Janeiro - RJ, Brasil.

[23]V. C. Barbosa, *On the Time-Driven Parallel Simulation of Two Classes of Complex Systems*, Publicações Técnicas, Programa de Engenharia de Sistems e Computação, COPPE, Universidade Federal do Rio de Janeiro, 1991.

[24]V. C. Barbosa, e M. C. S. Boeres, *An Occam-Based Evaluation of a Parallel Version of Simulated Annealing*, Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, Caixa Postal 68511, 21945 Rio de Janeiro - RJ, Brasil.

[25]V. C. Barbosa, *Redes Neuronais e "Simulated Annealing" como Ferramentas para Otimização Combinatória*, Programa de Engenharia de Sistemas e Computação,

COPPE, Universidade Federal do Rio de Janeiro, Caixa Postal 68511, 21945 Rio de Janeiro - RJ, Brasil.

[26] V. C. Barbosa, e E. Gafni, *A Distributed Implementation of Simulated Annealing*, Journal of Parallel and Distributed Computing, Vol. 6, pp. 411-439, 1989.

[27] J. H. M. Korst, e E. H. L. Aarts, *Combinatorial Optimization on a Boltzmann Machine*, Journal of Parallel and Distributed Computing, Vol. 6, pp. 331-357, 1989.