UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Simulador Distribuído para Redes Neurais Artificiais

por

DINAMÉRICO SCHWINGEL

Dissertação submetida à avaliação, como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dante Augusto Couto Barone

Orientador

Porto Alegre, agosto de 1995.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Schwingel, Dinamérico

Um Simulador Distribuído para Redes Neurais Artificiais / por Dinamérico Schwingel. — Porto Alegre: CPGCC da UFRGS, 1995.
99 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR–RS, 1995. Orientador: Barone , Dante Augusto Couto

1. Processamento Distribuído. 2. Processamento Paralelo. 3. Redes Neurais Artificiais. I. Barone , Dante Augusto Couto II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-graduação: Prof. Cláudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Agradeço a todas as pessoas que direta ou indiretamente contribuíram para a realização deste trabalho.

Em especial a minha futura esposa Beth, que tem minha gratidão pela compreensão das muitas horas dedicadas a este trabalho e pelo amor e apoio a mim dedicados durante os últimos anos.

Aos meus amigos e parceiros Daniel, Fernando e Rodrigo pela amizade e companheirismo, em especial neste último ano de desafios conjuntos.

Ao amigo Alex Guazzelli, que foi influenciador deste trabalho, pelo apoio e pelos anos em que convivemos e trabalhamos juntos. Agradeço também por permitir utilizar sua implementação de *Combinatorial Neural Model* para basear a paralelização e por revisar parte do texto, contribuindo com excelentes observações.

Ao meu orientador Dante Barone, que deu a direção deste trabalho.

A todos os amigos e colegas com quem compartilhei os últimos 7 anos de Universidade, pelo companheirismo e amizade.

Ao Roland Teodorowitsch por permitir basear a implementação paralela de *back propagation* em seu algoritmo seqüencial.

Aos demais ex-colegas de mestrado, que sempre estiveram dispostos a colaborar.

À minha família, em especial à minha mãe, Adélia, pela sua força interior que me serviu de exemplo, e aos meus irmãos Paulo e Carla.

À sociedade brasileira, por manter instituições que permitem a realização de trabalhos de pesquisa como este, subsidiando a sua realização.

A Deus, a quem devo a minha existência.

Este trabalho é dedicado ao meu pai,
Aldino Schwingel, em sua memória.

Sumário

Lista de Figuras

Lista de Tabelas

Siglas e Termos Utilizados

- **Aprendizado**: o termo aprendizado, do original em inglês *learning*, tem sido usado, normalmente, para se referir ao processo de apresentação dos padrões a uma rede neural e seu ajuste interno para que, posteriormente, reconheça esses padrões. Alguns autores têm utilizado o termo treinamento, em inglês *training*, para designar esse processo. Certamente é mais adequado se referir a esse processo de máquina com um processo de treinamento do que de aprendizado. Neste texto será dado preferência ao termo treinamento, a não ser nos casos em que a bibliografia original se referia como aprendizado.

- **Concorrência Lógica**: diz-se que duas tarefas seqüenciais S e T apresentam concorrência lógica se sua execução paralela, em qualquer ordem, não modificar os resultados por elas produzidos. Em outras palavras S e T não apresentam nenhum tipo de dependência.

- **Distribuição ou Balanceamento de Carga**: do original, em inglês, *load balancing*. Conceito baseado na premissa de que o tempo de execução de uma tarefa dividida em P porções, e executada em P diferentes processadores, é igual ao tempo de execução no processador mais lento. O termo Balanceamento de Carga se refere aos procedimentos que visam minimizar o tempo ocioso dos processadores, maximizando sua ocupação e diminuindo o tempo total de processamento. Existem, basicamente, duas formas de executar a distribuição de carga:

  - Estática: realiza a alocação dos recursos apenas uma vez e aguarda o término da execução para realocá-los;

- Dinâmica: realiza a alocação dos recursos no início do processamento e fica constantemente monitorando para realocá-los a qualquer momento.

- **Granularidade**: classificação da forma de exploração do paralelismo inerente aos programas, que se divide, basicamente, em duas categorias:

  1. *fine-grained*: exploração do paralelismo ao nível de instruções de máquina;
  2. *coarse-grained*: explora o paralelismo no nível de laços e subrotinas.

  É possível, ainda, acrescentar a categoria SPMD que explora o paralelismo no nível de processo. Basicamente, o termo grão (*grain*) se refere ao conjunto de instruções/subrotinas/programas que é executado sem interação (troca de mensagens) com outros processadores.

- **MIMD**: abreviatura de *Multiple Instruction Stream Multiple Data Stream*, que designa máquinas paralelas com múltiplos fluxos de execução e com múltiplos fluxos de dados, dentro da classificação de Flynn.

- **Multicomputador**: na literatura recente, as máquinas MIMD com memória distribuída têm sido denominadas de multicomputadores.

- **Multiprocessador**: as máquinas MIMD com memória compartilhada têm sido denominadas multiprocessadores, em contraposição aos multicomputadores.

- **Processamento Heterogêneo**: no âmbito deste trabalho, o termo será empregado para designar o processamento cooperativo realizado por uma rede de máquinas com arquitetura ou sistema operacional diferente. O mesmo termo também é empregado para designar o processamento cooperativo realizado por uma rede de máquinas com desempenhos diferentes [LOU 92].

- **SIMD**: abreviatura de *Single Instruction Stream Multiple Data Stream*, que designa máquinas paralelas com um único fluxo de execução e com múltiplos fluxos de dados, dentro da classificação de Flynn.

- **SPMD**: o termo *Single Process Multiple Data* se refere a um paradigma de processamento paralelo bastante similar ao das máquinas SIMD. Contudo,

neste último, há elementos processadores em *hardware* que executam as mesmas instruções simples sobre diferentes dados, enquanto que no paradigma SPMD as instruções podem ser vistas como processos que executam a mesma tarefa sobre diferentes sub-conjuntos (normalmente disjuntos) dos dados de entrada. Nos dois casos, isso é feito sob um controle centralizado. Embora seja um paradigma similar a máquinas SIMD, o paradigma SPMD é utilizado em máquinas MIMD.

Resumo

Este trabalho analisa o uso de redes de estações de trabalho como uma única máquina a ser utilizada para permitir o processamento de problemas que não poderiam ser computados, aceitavelmente, em apenas um de seus nodos, seja por causa do tempo dispendido ou de recursos físicos necessários, como memória principal.

São enfocados dois algoritmos de redes neurais artificiais — *Combinatorial Neural Model* e *Back Propagation* — que apresentam os problemas enunciados acima, e uma proposta de um esquema para distribuição dessa classe de algoritmos, levando em consideração as vantagens disponíveis no ambiente em questão, é apresentada.

A implementação do modelo proposto, sob a forma de um simulador distribuído baseado no conceito de servidor está descrita no trabalho, assim como as estratégias de paralelização dos algoritmos.

Ao final, são apresentados os resultados obtidos, quantitativa e qualitativamente, e uma avaliação mais detalhada da paralelização do algoritmo *Back Propagation* é exposta.

Palavras-chave: Processamento Paralelo, Processamento Distribuído, Redes Neurais.

TITLE: "A DISTRIBUTED NEURAL NETWORK SIMULATOR"

Abstract

The use of workstation networks as distributed multicomputers to solve resource demanding problems that cannot be feasibly solved in one node is the main concern of this work.

Two different artificial neural network algorithms, Combinatorial Neural Model and Back Propagation, are faced and a scheme for distributing this class of algorithms is presented. The several advantages of the environment are focused in the proposal along with its disadvantages.

This work also presents the implementation of the proposed scheme allowing an *in loco* performance evaluation.

At the end results are shown and a more in depth evaluation of the Back Propagation parallelization is presented.

Keywords: Parallel Processing, Distributed Processing, Artificial Neural Networks.

1 Introdução

A crescente evolução nos computadores seqüenciais, notadamente os conhecidos como Estações de Trabalho (ETs) tem trazido para as instituições em geral um poder computacional que, há poucos anos, era restrito a centros super especializados em poucos países do mundo e que era tratado como caso de segurança nacional. Aliado ao aumento de desempenho, houve uma redução nos preços, o que possibilitou o acesso de muitas instituições a essas facilidades.

Paralelamente, os chamados super computadores continuaram dominando o mercado de aplicações científicas pesadas com alguns aperfeiçoamentos tecnológicos. No entanto algumas empresas começaram a produzir equipamentos com novas arquiteturas voltadas para oferecer um elevado poder de processamento e interessadas em obter uma melhor relação de desempenho *vs.* custo, que é propiciada pelo uso de diversos processadores em paralelo.

A motivação inicial deste trabalho surgiu ao longo do mestrado no CPGCC, que conta com um ambiente de diversas ETs interligadas, propiciando um excelente laboratório para experimentação de métodos e técnicas de processamento distribuído, visando alto desempenho a um baixo custo. Custo este que se torna ainda menor quando se considera o poder computacional latente que é desperdiçado enquanto esses equipamentos não estão em uso, uma vez que são máquinas eminentemente voltadas para trabalho interativo. Além disso, a paralelização/distribuição de algoritmos pesados pode levar a uma melhor utilização dos recursos disponíveis, não implicando compra de novos equipamentos.

Posteriormente, foi publicado um trabalho [SCH 92b], analisando a exploração de paralelismo em ambiente distribuído, concentrando-se no modelo de paralelismo de dados e utilizando um algoritmo de síntese de imagens baseado na técnica de *ray-tracing*. Esse trabalho, juntamente com outros desenvolvidos no CPGCC, forneceram as bases para o que é proposto nesta dissertação.

Através de discussões com o Prof. Dante Barone, chegou-se a proposta de trabalhar com paralelização de modelos de Redes Neurais Artificiais (RNAs), sua principal área de interesse, enfocando problemas desses algoritmos que, a princípio não se adaptam bem a um ambiente de processamento distribuído. Tinha-se como objetivo oferecer ferramentas para possibilitar ao pesquisador de RNAs um aumento de desempenho nos algoritmos, principalmente na parte de treinamento das redes, um processo que é, na maioria das vezes, bastante lento.

Com isso, chegou-se ao presente trabalho, que apresenta um modelo de simulador de redes neurais independente de plataforma de *hardware*, independente de modelo de RNA e voltado para processamento distribuído de alto desempenho, sem excluir a utilização de máquinas dedicadas e com forte ênfase na modularidade e facilidade de expansão.

Para a experimentação do modelo proposto foram implementados dois modelos de RNAs no ambiente de processamento distribuído disponível nos laboratórios do CPGCC. Tais modelos estão apresentados no capítulo seguinte, juntamente com uma breve apresentação do que são RNAs. O texto não pretende ser extensivo e assume prévio conhecimento de conceitos básicos de RNAs. Para leituras introdutórias são fornecidas referências.

No capítulo posterior, apresentam-se os motivos que influenciam o atual crescente interesse em torno de processamento distribuído de alto desempenho. São discutidos também aspectos cruciais no projeto de sistemas distribuídos de alto desempenho, notadamente os que se diferenciam de outros ambientes de processamento paralelo, como máquinas paralelas fortemente acopladas.

O modelo do simulador proposto neste trabalho é apresentado no capítulo 4, e começa com uma descrição dos objetivos a serem atingidos. A seguir está a proposta do modelo, sua adequação aos objetivos e a explanação de como foi realizada a implementação. Ao final do capítulo, descrevem-se os pontos a serem atacados nos dois modelos de RNA, juntamente com a forma como eles foram paralelizados no âmbito do simulador.

Os resultados obtidos, em comparação com execuções seqüenciais dos mesmos algoritmos são apresentados no capítulo final. Ali também está uma discussão sobre possíveis aperfeiçoamentos que podem ser realizados no simulador, a partir do que se tem implementado hoje e uma análise mais detalhada do algoritmo paralelo *back propagation.*

2 Redes Neurais Artificiais

Este capítulo apresenta de uma forma genérica as Redes Neurais Artificiais (RNAs) e, mais especificamente, os dois modelos — *Combinatorial Neural Model* e *Back Propagation* — utilizados neste trabalho. O texto não pretende explicar o que são RNAs, mas apenas introduzir alguns conceitos relevantes ao trabalho. Uma introdução às RNAs pode ser encontrada em [GUA 92], onde outras referências estão disponíveis.

Ao contrário do que se pode supor pela atual explosão de interesses em torno de Redes Neurais Artificias, elas foram criadas há algumas décadas.

McCulloch e Pitts apresentaram seu modelo de neurônio artificial, o *Psychon*, em 1943; Hebb elaborou uma regra de aprendizado em 1949. Na década de 60 muitos trabalhos foram realizados: a proposição do perceptron, por Rosemblatt; a criação dos modelo ADALINE e MADALINE, por Widrow e Hoff; o modelo booleano SLAM, de Aleksander; e outros avanços que ocorreram, propiciando o atual interesse na área e culminando com diversos modelos bastante recentes como ART, *Back Propagation*, GSN e CNM.

Um dos avanços mais significativos que ocorreu foi a elaboração de um algoritmo de aprendizado para redes baseadas em perceptrons multi- camadas, que superou uma limitação dessas redes quanto ao reconhecimento de problemas não linearmente separáveis, como a função ou-exclusivo (*XOR*). Tal algoritmo, conhecido como *Back Propagation Learning Rule*, foi proposto por Rumelhart *et al* [RUM 86].

Atualmente, muita pesquisa está sendo realizada em RNAs, com concentração em três grandes áreas: modelagem, aplicação e implementação, sendo esta última a que abrange o presente trabalho. Quanto a aplicações, as RNAs vem sendo usadas com sucesso em diversos problemas, entre eles [GUA 92]: processamento adaptativo de sinais, reconhecimento de voz, sistemas especialistas e outros que requeiram generalização, tolerância a falhas e aprendizado de funções muito complexas. Elas não tem mostrado bom desempenho no cálculo de valores exatos e no aprendizado de algoritmos que não podem trabalhar com margem de erros.

Na visão deste autor, as RNAs trazem uma grande contribuição para a ciência de computação, pois quase a totalidade dos computadores que hoje usamos derivam de máquinas de calcular ou teares mecânicos que evoluíram para máquinas baseadas em acumulador com programa armazenado. Dentro desse enfoque o avanço foi muito grande: novas arquiteturas, sistemas operacionais multi-usuário, máquinas massivamente paralelas, etc. Contudo não se mudava a proposta de como programar essas máquinas nem se alterava a visão de máquina de calcular.

Com o enfoque das RNAs tenta-se modelar o processo computacional de uma forma semelhante ao cérebro dos animais superiores que, no mínimo, trata-se de uma mudança radical de enfoque, pois, agora, pode-se entender a programação de uma rede neural, ou um computador neural, como um processo de treinamento. Até então, era necessário codificar um programa para "ensinar" um computador a realizar uma determinada tarefa. Com um computador neural pode-se simplesmente mostrar o que ele deve fazer com alguns casos de treinamento e ele, idealmente, vai generalizar para outros casos e tomar atitudes adequadas, mesmo para entradas que não foram previstas no treinamento. Isso quebra totalmente com o conceito tradicional de computador, baseado em entrada-processamento-saída.

Contudo a situação está longe do ideal, muito há a ser feito no processo de mapeamento dos problemas reais para modelos aceitáveis como entrada para as RNAs que temos hoje, e não deve ser esquecido que as RNAs são baseadas no modelo do cérebro animal e, como ele, apresentam o mesmo tipo de deficiências ou características que não as tornam aplicáveis em certos problemas. Porém é uma

questão em aberto se essas limitações não serão resolvidas por algum modelo ainda por ser desenvolvido.

2.1 O Modelo Neural Combinatório

Esta seção descreve o modelo de redes neurais artificiais denominado *Combinatorial Neural Model* (CNM).

Trata-se de um modelo recente, mas que vêm provando sua eficiência, conforme estudos comparativos [GUA 94].

2.1.1 Objetivos do Modelo

O Modelo Neural Combinatório — *Combinatorial Neural Model* — foi criado por Machado [MAC 89], inspirado em pesquisas neurofisiológicas, lógica incerta e com forte influência de engenharia do conhecimento.

A estrutura da rede segue muito próxima ao método para análise e aquisição de conhecimento heurístico de especialistas desenvolvido por Lêao, Rocha *et al.* Nesse método, os especialistas expressam seu conhecimento através de grafos unidirecionais que conectam as evidências às hipóteses, passando, freqüentemente, por nodos intermediários que agrupam informação logicamente, através das operações lógicas E e OU (*AND* e *OR*).

Quando este autor começou a conhecer RNAs, em 1990, uma característica chamou a atenção, notadamente no modelo *Back Propagation*: a rede recebia diversos padrões de treinamento e modificava seu estado interno para reconhecê-los, contudo não era possível, para a rede, explicar como ela tinha aprendido ou porque ela tinha escolhido uma determinada saída. O CNM, pela sua base em engenharia de conhecimento, tem a capacidade de explanação que aquele não possuía e isso o distingue de forma particular de outros modelos.

O modelo trabalha com aprendizado heurístico, a partir do conhecimento prévio de especialistas, e seu posterior refinamento, o que, segundo Machado, corresponde a mecanismos neuronais básicos como adaptação, atenuação e condicionamento.

Essas características o tornam um modelo excelente para utilização em conjunto com sistemas especialistas, de forma híbrida, superando outros modelos como apresentado em [GUA 94].

2.1.2 Descrição do Modelo

Os modelo conexionistas, usualmente, são formados por um conjunto de elementos processadores simples, chamados neurônios, que trabalham em paralelo e se comunicam através de uma topologia pré-fixada. Este modelo não é exceção e, nele, cada unidade executa um processamento simples que pode ser uma operação lógica E ou OU. As unidades computam seu estado de ativação, pertencente ao intervalo [0,1], baseadas apenas em suas entradas. As unidades podem operar tanto em modo booleano, caso em que o estado de ativação assume os valores 0 ou 1, quanto em modo *fuzzy*, no qual os valores de saída assumem qualquer valor do intervalo [0,1].

Arcos, ou sinapses, conectam as unidades e são classificados em excitatórios ou inibitórios. Possuem, também, um peso que varia de 1 (conexão completa) a 0 (sem conexão) e conectam as unidades de forma unidirecional (*feed-forward*).

A figura 2.1 exibe os dois tipos de unidades que são usadas no modelo. As entradas da unidade i são denominadas y_j, $j = 1, ...n$ e podem assumir qualquer valor do intervalo [0,1].

A saída, ou estado de ativação, y_i também assume valores da faixa [0,1] e representa o grau de possibilidade do conceito representado pelo neurônio. O grau de aceitação, ou crença, no conceito representado em cada entrada da unidade é

atenuado por um peso denominado w_{ij} que varia de 0 a 1. w_{ij} representa o peso da conexão (sinapse) que parte da unidade j para a unidade i.

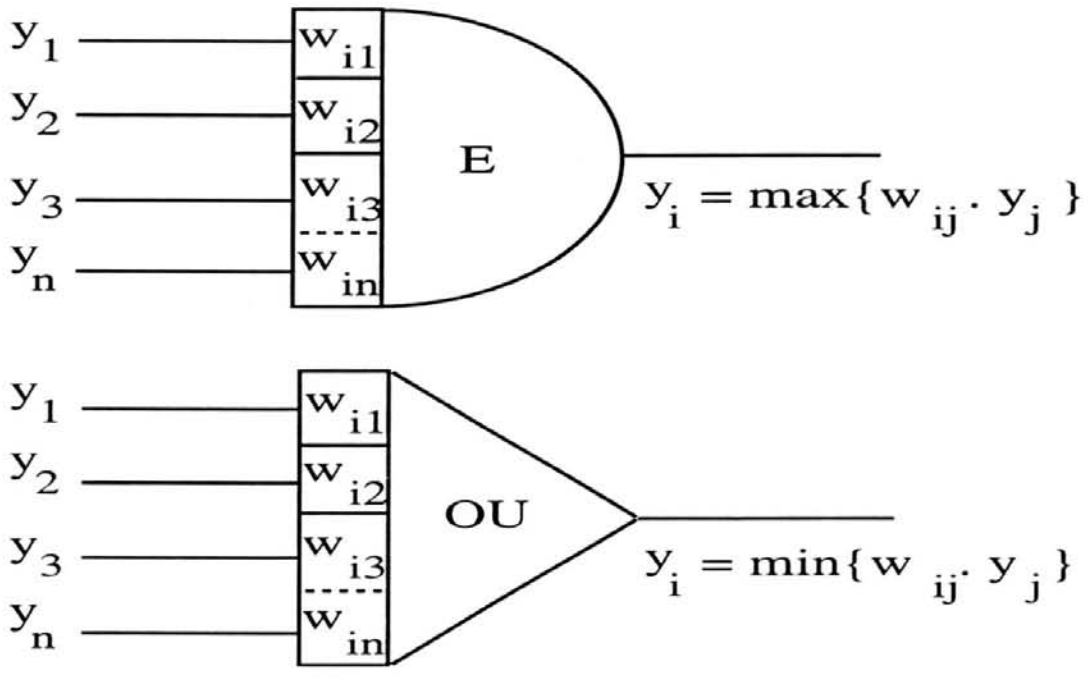

FIGURA 2.1 — Unidades do Modelo Neural Combinatório

O estado de ativação de uma célula depende exclusivamente dos valores encontrados em suas entradas e dos pesos associados a cada uma das conexões. Para as unidades OU, a saída é calculado da seguinte forma:

$$y_i = max(w_{i1}.y_1, w_{i2}.y_2, ..., w_{in}.y_n)$$

A unidade i adota em sua saída o valor máximo encontrado em suas entradas, atenuados pelos pesos das conexões. De acordo com o autor do modelo, isso corresponde à implementação do mecanismo de competição entre as entradas. De maneira inversa, as unidades E calculam seu estado de ativação como o mínimo de suas entradas:

$$y_i = min(w_{i1}.y_1, w_{i2}.y_2, ..., w_{in}.y_n)$$

As unidades são conectadas em uma topologia com três ou mais camadas, sendo que qualquer rede com mais de três camadas pode ser representada por outra rede com até três. Assim, os exemplos encontrados na literatura se limitam a esse número de camadas.

A primeira camada é denominada de Camada de Entrada (*Input Layer*) e recebe os dados do ambiente externo à rede. Os dados são apresentados com um grau de crença que varia de 0 (descrença) a 1 (crença total). Cada unidade dessa camada representa um conceito e o grau de confiança no conceito é representado por seu estado de ativação.

A camada intermediária é denominada Camada Combinatória (*Combinatorial Layer*) e é composta por unidades que realizam a operação lógica E (AND). O objetivo desta camada é associar evidências de entrada em blocos (*clusters*) de informação, baseado em que essa é uma característica fundamental do raciocínio de especialistas e de que isso é feito, freqüentemente, através da operação lógica E.

A versão completa do modelo inclui todas as combinações de evidências de entrada começando com $C(n,2)$ até $C(n,n)$, onde n representa o número de unidades na camada de entrada.

Percebe-se, claramente, um problema de espaço para a representação física de tais combinações em um computador. O autor sugere uma limitação de ordem para a rede, introduzindo o conceito de Modelo de Ordem m que inclui apenas as combinações agrupadas até o valor de m, *i. e.* até $C(n,m)$ e sugere, baseado em estudos de Psicologia e na observação de grafos de conhecimento criados por especialistas, a adoção do valor sete, mais ou menos, dois. Visa, dessa forma, minimizar o problema da explosão combinatória que surge mesmo para valores não tão elevados de n, contudo, com a diminuição da ordem, a entropia da rede diminui, reduzindo sua capacidade de aprendizado.

Um exemplo do modelo completo pode ser visto na figura 2.2 para uma rede com 3 evidências e duas hipóteses, onde estão representadas as conexões originadas por cada combinação.

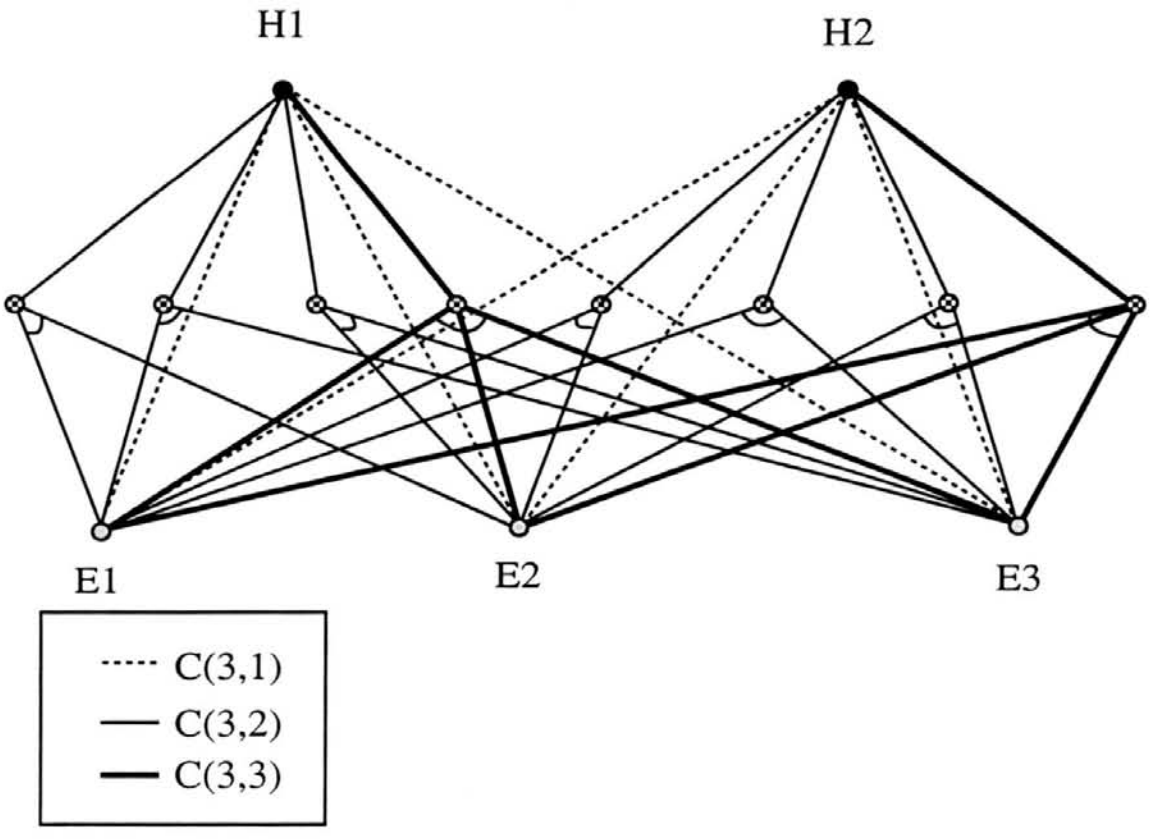

FIGURA 2.2 — Modelo Completo para 2 Hipóteses e 3 Evidências

A última camada é a Camada de Saída (*Output Layer*), formada por células OU (*OR*), cada uma representando uma hipótese no domínio do problema.

Um exemplo de uma rede CNM mais simples pode ser visto na figura 2.3. Tal rede é composta por 3 unidades de entrada (evidências), 2 unidades intermediárias e 1 hipótese.

2.1.3 Algoritmos Aplicados ao Modelo

A seguir estão descritos os algoritmos aplicados ao Modelo Neural Combinatório para a fase de treinamento e para a fase de teste da rede.

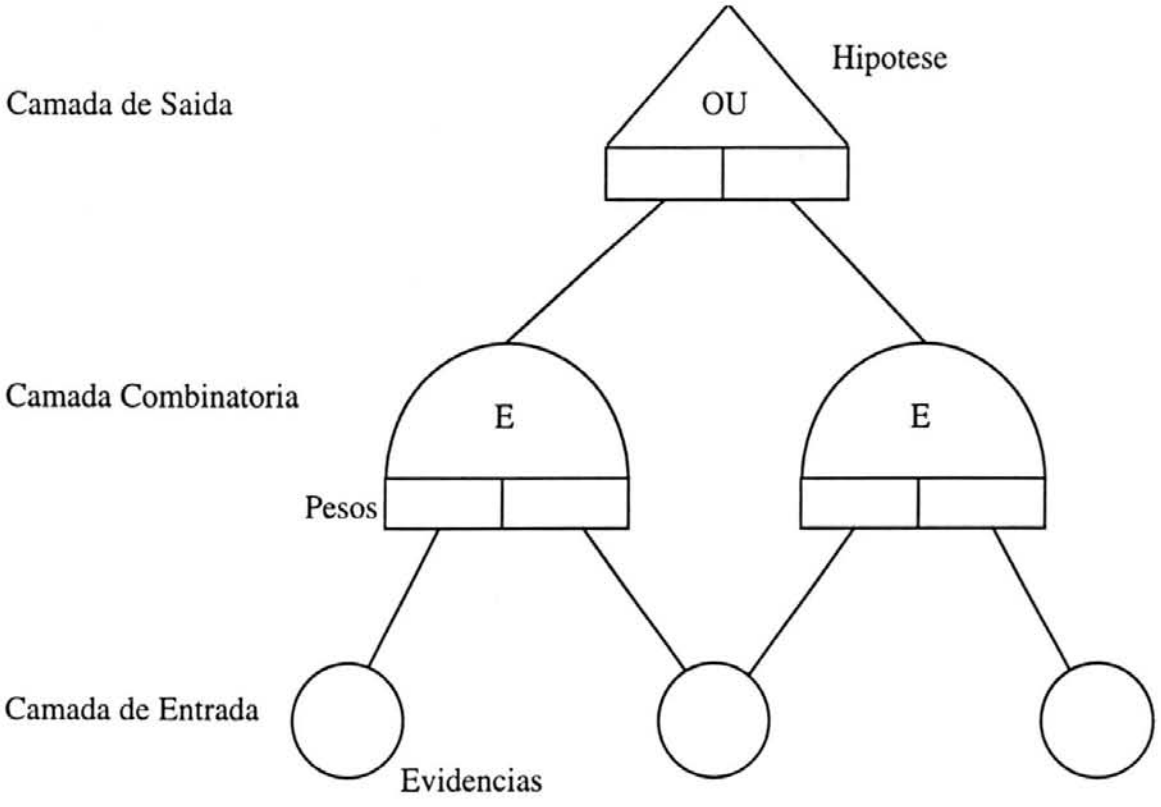

FIGURA 2.3 — Exemplo de uma Rede CNM

2.1.3.1 Algoritmo de Treinamento

A base de dados que contém os casos a serem ensinados à rede contém, também, a classificação correta das hipóteses ativadas. O algoritmo é supervisionado e é aplicado em todos os casos da base de dados em uma iteração.

Regra de Aprendizado Punição e Recompensa:

1. Para cada conexão da rede atribua um acumulador com valor inicial zero;
2. Para cada caso do conjunto de treinamento faça:
   (a) Propague as evidências das unidades de entrada até as unidades de saída;
   (b) Para cada conexão que chega a uma unidade da saída (com nível de atividade v) faça:
   Se a a hipótese atingida corresponde à classe correta
   Então retro-propague desta unidade até as unidades de entrada, somando

aos acumuladores das conexões percorridas o valor v
(Recompensa);
Senão retro-propague desta unidade até as unidades de entrada, subtraindo dos acumuladores das conexões percorridas o valor v (Punição);

O algoritmo processa todos os casos em uma iteração e produz uma rede treinada, com acumuladores variando de $[-T, +T]$, onde T é o número de casos do conjunto de treinamento.

2.1.3.2 Algoritmo de Poda e Normalização

Para que a rede fique operacional, é necessário passá-la por um processo de poda (*prunning*) e de normalização em que todas as sinapses (conexões) com acumuladores negativos ou abaixo de um determinado limiar são eliminadas. O algoritmo a seguir realiza essas operações.

Regra de Poda da Rede

1. Remova da rede todas as conexões cujos acumuladores estejam abaixo do limiar;
2. Remova todas as unidades das camadas de entrada e combinatória que fiquem desconectadas de todas as unidades da saída;
3. Deixe os pesos das conexões iguais ao seu valor dividido pelo maior valor encontrado entre os pesos da rede para normalizá-los;

2.2 O Modelo de Redes Neurais Artificiais Back Propagation

Esta seção descreve o modelo de Redes Neurais Artificiais conhecido como *Back Propagation* que é o mais difundido e utilizado nos dias de hoje. Devido a isso,

muitas variações e aperfeiçoamentos têm sido propostos sobre o modelo original [RUM 86].

Este modelo têm sido usado com sucesso em muitas pesquisas e em aplicações comerciais nos últimos anos.

2.2.1 Fundamentos do Modelo

As unidades (neurônios) deste modelo são baseadas em estudos sobre associação de padrões que utilizavam as unidades conhecidas como *perceptrons*.

Um dos teoremas provados para os *perceptrons* foi o que dizia que era sempre possível encontrar um conjunto de pesos [1] para solucionar o problema apresentado, desde que esse conjunto existisse. O principal problema é que esse conjunto nem sempre existe.

Muito há escrito na literatura a respeito de funções não separáveis linearmente (*not linearly separable*), usando como exemplo a função ou-exclusivo (XOR), para exemplificar porque os *perceptrons* não resolvem essa classe de problemas.

Porém, com a adição de uma camada ao modelo original mostra-se que o problema do ou-exclusivo pode ser resolvido pelos *perceptrons*. A partir desse fato, procurou-se um algoritmo de aprendizado (treinamento) que fosse genérico para uma rede multi-camadas. Para isso, partiu-se do algoritmo conhecido como *Least Mean Square* similar ao usado nos *perceptrons* que visa minimizar o erro da rede[2] Chegou-se, assim, ao *Back Propagation Learning Rule*, que foi citado anteriormente.

[1] Configuração da rede

[2] O erro é a diferença entre o valor encontrado na saída de rede e o valor desejado (alvo).

2.2.2 Descrição do Modelo

O modelo *Back Propagation* compreende, então, redes *perceptrons* multi-camadas, usando um algoritmo próprio para o treinamento da rede. A figura 2.4 exibe uma típica rede *back propagation* com três camadas, já introduzindo os nomes que serão usados nas fórmulas a seguir. Nela, as notações são referentes ao neurônio da camada de saída e estão representadas apenas parte das conexões, pois cada neurônio possui conexão com todos as unidades da camada seguinte.

A primeira camada não é composta por elementos processadores (neurônios), fazem parte dela os próprios sinais de entrada injetados na rede. As camadas intermediárias e a de saída são constituídas por elementos processadores.

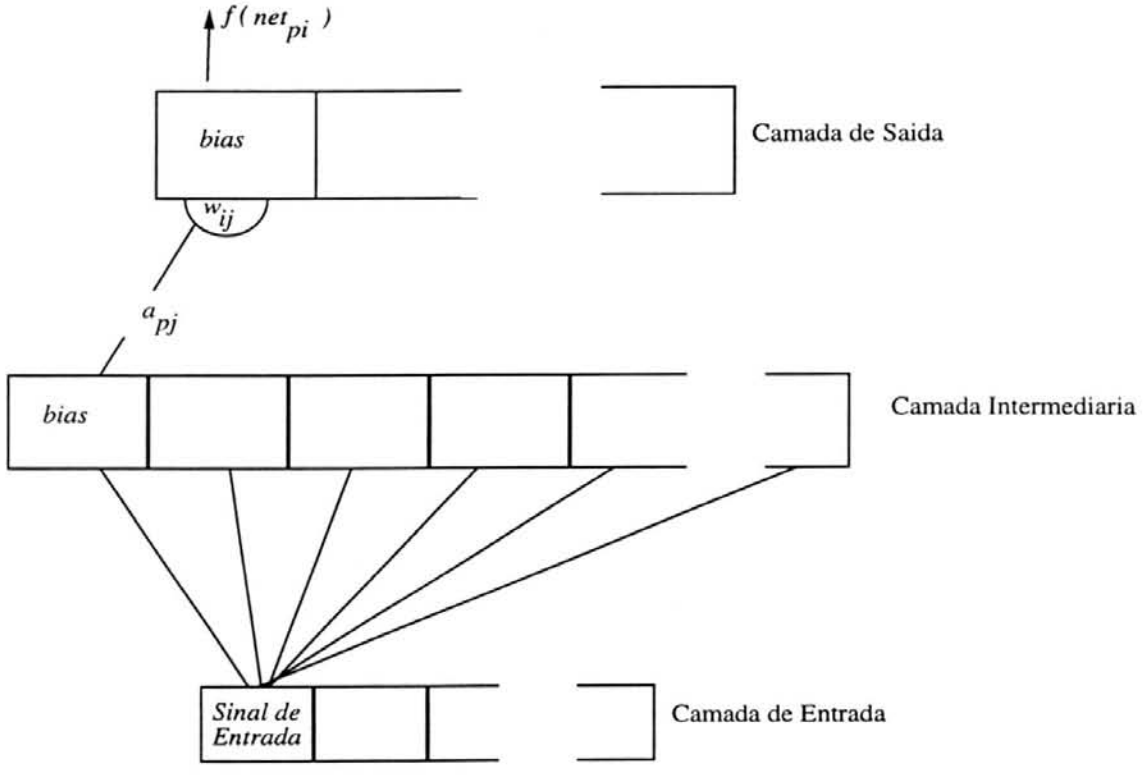

FIGURA 2.4 — Modelo Completo para 2 Hipóteses e 3 Evidências

A seguir são apresentadas as operações realizadas pelas unidades, nas diferentes fases de operação da rede.

2.2.2.1 Fase de Teste ou Reconhecimento

Na fase de teste ou reconhecimento, as unidades recebem os sinais das unidades da camada predecessora e vão, por eles, sendo ativadas ou não, de modo que os sinais obtidos nas unidades da camada de saída sejam iguais (ou aproximados) ao alvo que foi apresentado à rede, na fase de treinamento.

Trata-se de um processo unidirecional (*feed-forward*) e as operações executadas pelos neurônios na fase de teste ou reconhecimento são simples e expressas pelas equações a seguir.

Cada conexão entre os neurônios possui um peso e, para determinar seu sinal de saída, net_{pi}, o neurônio multiplica cada sinal de entrada a_{pj} pela conexão associada w_{ij}, somando os resultados ao valor que está armazenado em sua estrutura $bias$.

$$net_{pi} = \sum w_{ij} a_{pj} + bias_i$$

Esse valor é, então, passado a uma função de ativação $f(net_{pi})$ que determinará se o sinal calculado resultará em uma saída ativa ou não.

$$f(net_{pi}) = 1/(1 + e^{net_{pi}})$$

2.2.2.2 Fase de Treinamento ou Aprendizado

A fase de aprendizado está sendo apresentada depois porque nela é realizado o mesmo processamento anterior, com o acrécimo de um passo, executado em sentido contrário, que muda o valor dos pesos das conexões w_{ij} e dos $bias_i$, visando fazer com que os cálculos anteriores gerem o sinal apresentado como alvo.

A fórmula para a mudança dos pesos e do $bias$ de cada neurônio é a seguinte:

$$w_{ij}(t+1) = \epsilon(\delta_{pi} a_{pj}) + \alpha w_{ij}(t)$$

Onde t representa a iteração, ϵ — coeficiente de aprendizado — multiplica o valor da alteração nos pesos e α — coeficiente de amortecimento — determina a influência dos pesos anteriores na composição do novo valor. O valor de δ é determinado de formas diferentes para camadas de saída e camadas intermediárias.

Para camadas de saída, δ é obtido com a seguinte equação:

$$\delta_{pi} = a_{pi}(1 - a_{pi})(t_{pi} - a_{pi})$$

Onde a_{pi} é o valor de ativação do neurônio e t_{pi} é o alvo, *i. e.*, o valor que deveria ter a_{pi} para que o neurônio tivesse sido treinado.

Para uma camada intermediária genérica, δ_{pi} é calculado em função dos pesos das conexões subseqüentes à camada e expresso pela equação a seguir:

$$\delta_{pi} = a_{pi}(1 - a_{pi}) \sum \delta_{pk} w_{jk}$$

A expressão $a_{pi}(1 - a_{pi})$, que é usada nas duas equações, é a derivada da função de ativação. Ela determina a direção do passo, no domínio dos pesos, para que seja atingido o mínimo da função de erro.

O procedimento de treinamento em *back propagation*, de forma similar ao seu antecessor *Least Mean Square*, baseia-se na derivada para percorrer a superfície de erro, sempre de forma descendente (*gradient descent*), visando atingir o mínimo da função.

O procedimento *Least Mean Square* não tinha problema quando aplicado aos *perceptrons* pois as superfícies de erro sempre eram bem comportadas, *i. e.*, havia apenas um mínimo. Entretanto em redes multi-camadas aparecem mínimos locais nessa superfície, revelando um problema do algoritmo que pode, agora, ficar preso em um mínimo local e não avançar até o mínimo global, dependendo dos parâmetros de controle e da função de erro.

3 Processamento Paralelo e Distribuído

Este capítulo apresenta uma visão geral sobre processamento paralelo e distribuído, na qual primeiro enfoca-se a parte histórica, o escopo e, posteriormente, são discutidos pontos chave no projeto de sistemas distribuídos de alto desempenho.

3.1 Introdução

A medida que os computadores passam de sua atividade primária, que é, o processamento de dados para outras que exigem poder computacional sempre crescente [HWA 84], surge a necessidade por máquinas que ofereçam esse poder.

O uso de diversas unidades processadoras para diminuir o tempo de processamento, principalmente os cálculos de aplicações científicas, foi cogitado há várias décadas e, ainda nos anos 50, a Bull contruiu o Gamma 60 que contava com múltiplas unidades funcionais [WIL 94]. Posteriormente, foram construídas máquinas baseadas em processadores matriciais (*array-processors*), *pipelines*, fluxo de dados (*data flow*) e outras variações, sempre visando um aumento de desempenho.

Como resultado dessas pesquisas houve a construção de diversas máquinas conhecidas comercialmente como supercomputadores que dominam, ainda hoje, o mundo das aplicações científicas.

Contudo essas máquinas eram e algumas ainda são, em sua maioria, baseadas em um único processador muito veloz. Assim, questionava-se um limite teórico para esse tipo de enfoque, uma vez que a velocidade de propagação da eletricidade seria um teto onde o desempenho iria bater.

A solução visualizada era o uso de diversos processadores muito velozes para continuar aumentando o desempenho desses computadores e isso é o que tem sido feito, concentrando muito esforço em pesquisas para estudar redes de interconexão de processadores e memórias que minimizem os problemas surgidos. Redes como *shuffle-exchange*, *cube connected cycles*, *hypercubes*, *trees*, *mesh* e outras são utilizadas em diversas máquinas e experimentos.

Atualmente, avanços em algoritmos de roteamento, com o *Wormhole Routing* [SEI 88] têm proporcionado a construção de máquinas com centenas de processadores como Intel IPSC, Paragon XPS, Ametek 2010 e outras. Tais máquinas fornecem um multicomputador dedicado para uso por um ou mais usuários, cada um podendo ter pedaços da máquina apenas para suas tarefas. Alguns sistemas também fornecem estratégias para alocação automática de processadores e distribuição de carga.

Este trabalho possui um escopo conceitualmente similar a este último apresentado, concentrando-se na utilização de redes de estações de trabalho como um Computador Distribuído de Alto Desempenho.

Há hoje, nas diversas redes espalhadas pelo globo, muitos computadores com razoável capacidade de processamento que estão sendo subutilizados durante boa parte do dia, o que resulta em um poder computacional latente muito grande que pode ser aproveitado para aplicações computacionalmente intensivas sem que seja

necessário investir em uma máquina de alto custo como um supercomputador tradicional.

Essa é uma área que vem crescendo nos últimos anos e deve resultar em sistemas operacionais voltados para processamento cooperativo que ofereçam facilidades bem maiores do que as disponíveis hoje. Algumas iniciativas nesse sentido já existem, como o PVM [SUN 90], mas ainda no nível de aplicação.

3.2 Tópicos a considerar no Projeto

Esta seção discute os principais pontos a serem considerados no projeto de um algoritmo paralelo, seja a arquitetura forte ou fracamente acoplada. Entretanto as soluções e técnicas discutidas restringem-se ao escopo de paralelismo de dados em ambiente distribuído, por motivos que estão expostos no texto.

3.2.1 Eficiência de Comunicação

A eficiência das comunicações é o ponto fundamental do projeto de sistemas paralelos. A ordem de grandeza do tempo envolvido na troca de informações entre processadores com memória distribuída é muito superior a que se costuma trabalhar em máquinas seqüenciais ou com memória compartilhada. Por isso deve-se projetar muito bem o esquema de comunicação entre os processadores, até mesmo fazendo operações redundantes, pois, normalmente, é mais rápido calcular um dado do que solicitá-lo a um processador que já o calculou.

Em um ambiente distribuído, a comunicação entre processadores não é feita a mesma velocidade encontrada em máquinas fortemente acopladas. Este projeto foi desenvolvido sobre uma rede de comunicação Ethernet com velocidade de 10 Mbit/s, enquanto que máquinas paralelas como os *transputers* valem-se de, pelo menos, 4

links de comunicação por processador dedicados e concorrentes que operam a 20 Mbit/s.

Outro fator que agrava a comunicação neste ambiente é o compartilhamento do meio de comunicação. Tipicamente, em redes de estações de trabalho, há um meio físico comum para troca de mensagens, gerando problemas de conflitos no uso desse meio, além da banda gasta com dados do próprio protocolo.

Em vista desses fatores agravantes, deve ser dispensada atenção muito especial à eficiência de comunicação neste ambiente e, desde já, fica claro que um algoritmo com muita dependência de dados, exigindo muita troca de informação, terá dificuldade para conseguir bom desempenho.

3.2.1.1 A exploração da Concorrência Lógica

Há, basicamente, duas formas de se explorar a concorrência lógica presente em algoritmos. Eles podem apresentar paralelismo de dados, paralelismo de processo ou, o que é mais comum, os dois. A exploração dos dois tipos de paralelismo pode, até mesmo, ser feita em diferentes níveis de abstração e com diferentes paradigmas.

Paralelismo de processo se refere à execução de diferentes intruções sobre um mesmo fluxo de dados, por unidades processadoras independentes. Se olharmos o código de um programa como um fluxo de dados, sob o ponto de vista do processador, é fácil perceber que um processador *pipeline* explora esse tipo de paralelismo, que está presente no código do programa.

O paralelismo de dados, por outro lado, está presente em algoritmos que podem calcular o resultado de seu processamento sem depender de valores anteriores. Tipicamente, são codificados dessa forma algoritmos muito complexos e que demoram muito tempo para resolver uma pequena parte do problema. O cálculo da imagem de uma função muito complexa pode ser feito dessa forma, pois a imagem do ponto x não depende da imagem de nenhum outro ponto no domínio. Dessa forma, cada processador pode calcular uma parte da imagem de forma independente.

A seguir, é apresentada uma formulação, baseada em um exemplo simples, que mostra porque é mais vantajoso explorar paralelismo de dados do que paralelismo de processo no ambiente em questão.

Suponha-se que os dados a serem processados estejam contidos em uma matriz bidimensional $A[N, M]$ e que sejam necessárias k unidades de tempo para processar o dado $A[i, j]$.

A figura 3.1 exibe a modelagem que explora o paralelismo contido nos processos a serem executados sobre o dado, usando P processadores e um fluxo de dados. Idealmente, k/P unidades de tempo seriam utilizadas por cada processador para o dado $A[i, j]$, ao final das quais, uma mensagem seria enviada ao próximo processador, transmitindo o resultado de seu processamento sobre o dado. Percebe-se que o número de mensagens, designado por F de fluxo, é dado pela expressão:

$$F = NMP$$

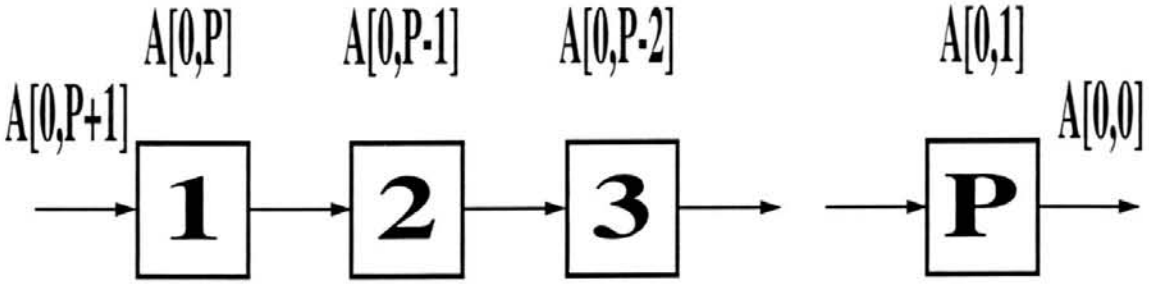

FIGURA 3.1 — Modelagem explorando paralelismo de processo

Explorando-se o paralelismo de dados a comunicação diminui bastante, uma vez que há uma fase de inicialização, a partir da qual os processadores somente se comunicam ao final do trabalho (figura 3.2), gerando um fluxo de mensagens bastante inferior. Esse fluxo é dado pela expressão:

$$F = 2P$$

No primeiro caso, o número de mensagens é proporcional às dimensões do problema. Dessa forma o processamento de maiores volumes de dados tende a gerar mais mensagens.

Entretanto, no segundo caso, isso não ocorre. O fluxo de mensagens é linearmente proporcional ao número de processadores e não depende do tamanho do problema. Assim, aumentado o número de processadores, aumenta-se o fluxo de mensagens, mas o poder computacional também.

Desse modo, a maior restrição ao primeiro paradigma é o uso de um meio de comunicação compartilhado, que tende a ficar saturado, principalmente com o uso de grandes volumes de dados.

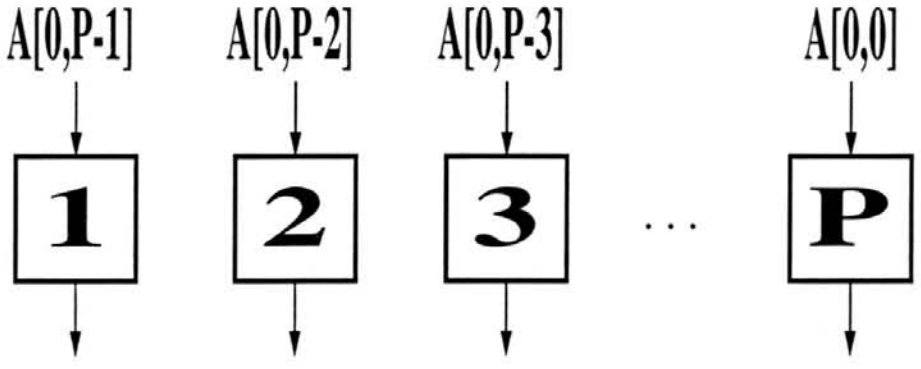

FIGURA 3.2 — Modelagem explorando paralelismo de dados

Apesar de tudo, nem sempre é possível dividir igualmente o trabalho entre os processadores em apenas um passo de inicialização. Nesse caso, para atingir um bom balanceamento de carga, pode-se dividir os dados em T porções para que sejam distribuídas dinamicamente, gerando $2TP$ mensagens.

Considerando as fórmulas anteriores, pode-se afirmar que: para o mesmo número de processadores a exploração de paralelismo de dados será mais eficiente somente se $T < MN/2$.

Percebe-se, assim, claramente a relação de compromisso entre balanceamento de carga e fluxo de mensagens: quanto maior o tamanho de cada porção, menor o número de mensagens, entretanto mais grosseiro será o balanceamento de carga.

3.2.2 Balanceamento de Carga

Depois das comunicações, este é o assunto mais importante em processamento paralelo, uma vez que o tempo de execução de um tarefa, dividida em T porções e levada a cabo por P processadores, é igual ao tempo de execução do processador mais lento. Assim, para que não haja processadores ociosos deve haver uma forma de se distribuir a mesma quantidade de trabalho para cada processador. Isso, na maioria das vezes, não significa o mesmo número de porções porque elas podem ter cargas diferentes.

Em um ambiente paralelo fortemente acoplado, por exemplo em uma rede de *transputers*, pode-se dispor de processadores dedicados o que facilita a monitoração da carga. Em ambiente distribuído é necessário considerar que não se está trabalhando com máquinas dedicadas, de modo que a carga do sistema pode variar por fatores externos, quer seja por processos de outros usuários ou por processos de sistema.

Uma estratégia eficaz e largamente utilizada para distribuição de carga, quando se explora paralelismo de dados, é a divisão dos dados em pacotes, valendo-se de uma distribuição dinâmica destes. Na maioria das aplicações, a carga de cada pacote não pode ser estimada previamente, ficando por conta de uma monitoração da carga do processador a decisão de enviar outro pacote aquele processador. Essa monitoração pode ser feita pelo próprio processador que decide enviar parte de seu trabalho a outro ou, de uma forma menos complexa, pelo processo que distribui os pacotes.

3.2.3 Processamento Heterogêneo

O uso de protocolos e meio físico de interconexão comuns entre os diversos fabricantes de estações de trabalho permite que suas máquinas sejam conectadas em uma mesma rede e operem de forma cooperativa. Isso possibilita que o problema seja

resolvido por máquinas construídas sobre processadores diferentes, o que é chamado de processamento heterogêneo.

Para que diferentes máquinas operem cooperativamente é necessário que disponham das mesmas estruturas de comunicação, o que pode ser provido pelo uso de um padrão ou então exige a implementação de um protocolo próprio de comunicação.

Com os dois lados da comunicação operando sobre as mesmas primitivas é facilitada a manutenção do programa, uma vez que existe apenas um código fonte para todas as máquinas e a interface dependente de máquina não é visível na codificação da solução do problema.

3.2.4 Tolerância a Falhas

Assim como a carga de cada processador pode ser alterada por fatores externos, muito facilmente pode-se perder contato com outro processador por esses mesmos fatores. Uma rede fracamente acoplada pode estar dispersa até mesmo em edifícios vizinhos e, muitas vezes, não há garantia de que as máquinas estarão operacionais nos mesmos horários. Ainda, falhas adicionais podem ser introduzidas pelos usuários que estão operando localmente as máquinas.

Além de falhas que podem ocorrer nos processadores, também a rede de comunicação pode ser danificada obrigando o sistema a manter informações sobre o trabalho de cada processador.

4 Redes Neurais Distribuídas

Este capítulo apresenta a modelagem proposta para atingir os objetivos deste trabalho e descreve a implementação do protótipo que foi desenvolvido para testar a modelagem. Também são apresentadas as ferramentas de *software* utilizadas e as estratégias de paralelização dos modelos aqui discutidos.

4.1 Fundamentos e Objetivos

As redes neurais artificiais têm, hoje, a atenção de inúmeros pesquisadores ao redor do planeta e isso está proporcionando um rápido crescimento da área com aplicações em diversas outras áreas do conhecimento humano. Contudo a implementação dos algoritmos computacionais que representam fisicamente essas redes têm levado a programas com grandes exigências computacionais, principalmente no processo de treinamento.

Como já citado anteriormente, este trabalho visa apresentar uma proposta de um simulador paralelo de redes neurais, enfocando principalmente os aspectos de software e pretendendo ser adaptável a diferentes plataformas de hardware. Pretende-se oferecer uma ferramenta que permita ao pesquisador de RNAs realizar experimentos com grandes quantidades de dados e que exijam grande poder computacional sem se preocupar com os aspectos de implementação do algoritmo de

forma paralela, *i. e.*, o sistema deve se comportar da mesma forma como se fosse um sistema local e seqüencial.

Ao mesmo tempo que se visa oferecer uma ferramenta ao pesquisador de RNAs, também se tem como objetivo preparar um ambiente aberto e flexível que possa servir para a implementação e teste de novos algoritmos paralelos/distribuídos de RNAs sem que isso cause impacto nos que já estão implementados no sistema. Com isso pretende-se agregar o trabalho de pesquisadores da área de processamento paralelo.

Além desses objetivos primários — alto desempenho e uma ferramenta de pesquisa — outros objetivos mais específicos, que nortearam este trabalho, são apresentados a seguir:

1. Transparência: qualquer que seja o modelo do simulador paralelo, ele deve esconder do usuário a implementação física da rede neural, *i. e.*, o usuário deve interagir da mesma forma com o simulador esteja a rede neural implementada em um supercomputador vetorial, *chips* neurais ou em uma rede de estações de trabalho.

2. Modelagem orientada para multicomputador: este enfoque foi adotado por ser esta a realidade onde o trabalho foi desenvolvido, de modo que seria possível testar as idéias com um protótipo. Contudo esse não foi o motivo mais forte, mas sim as grandes vantagens oferecidas por um ambiente desse tipo, principalmente no que tange a custo de equipamentos *vs.* capacidade computacional, e a crescente expansão no uso desses ambientes em quase todos os tipos de corporações - como visto na seção 3. Por esse enfoque, o simulador não deveria apresentar restrições para que algoritmos desenvolvidos em outras plataformas fossem incorporados, mas deveria obrigatoriamente permitir que o paradigma de multicomputador pudesse ser utilizado.

3. Simulador baseado em blocos: tem como principal objetivo permitir a fácil integração de outros modelos de redes neurais ao simulador. Como objetivos secundários busca-se permitir fácil atualização de algoritmos, disponi-

bilização de diferentes versões de algoritmos e, principalmente, independência de plataforma de *hardware*.

4. Distribuição de carga: o simulador deve prover alguma forma de distribuir a carga computacional por ele administrada. Neste trabalho, dada a complexidade do tema, optou-se por tratar apenas da distribuição de carga estática sob o ponto de vista da paralelização dos algoritmos.

5. Interatividade: o simulador deve permitir seu uso interativo por um pesquisador de redes neurais que queira fazer seguidos experimentos variando parâmetros do algoritmo. Preferencialmente, deve permitir o uso de uma interface gráfica para facilitar seu uso por parte de usuários menos experientes também.

   Müller [MUL 94] cita que o processo de configuração para um ou mais experimentos deve possuir algumas características e advoga o uso de uma linguagem de programação especialmente projetada para isso. Abaixo estão listadas as características que ele considera essenciais para essa linguagem de programação e observações quanto ao uso de uma interface gráfica no lugar de uma linguagem são apresentadas.

   - Interatividade: seu principal objetivo é eliminar a fase de compilação. Observa-se, no entanto, que uma interface gráfica oferece maior grau de interatividade que qualquer linguagem de programação. Contudo os serviços oferecidos pelo servidor devem ser os mesmos que a linguagem ofereceria, para que a pontencialidade da linguagem não seja perdida.
   - Facilidade de aprendizado e uso: uma interface gráfica oferece maiores facilidades do que uma linguagem de programação.
   - Flexilidade: uma linguagem tem maior facilidade em oferecer flexibilidade ao usuário. Se este for o ponto fundamental então uma linguagem deve ser fortemente considerada, pois para uma interface gráfica ofecer flexibilidade teria que perder em simplicidade e facilidade de uso e, mesmo assim, talvez nao conseguisse igualar-se à linguagem.

- Laços, expressões condicionais e variáveis: características específicas de uma linguagem. Este item está muito ligado à flexibilidade e cabe o mesmo comentário anterior.
- Interface transparente para as funções de redes neurais: em uma linguagem de programação há formas de abstração de código bem conhecidas, contudo uma interface gráfica pode fornecer a mesma potencialidade, e a transparência pode ser oferecida já ao nivel do servidor.

Em resumo, há certamente uma perda em flexibilidade com o uso de uma interface gráfica comum, mas é possível também optar-se por uma interface complexa e que permita descrever um fluxo de tarefas. Nesse caso poderia não haver perda de flexibilidade. Por outro lado há um ganho na facilidade de uso e na interatividade que pode ser significativo para pesquisadores não habituados a linguagens de programação. Baseado nisso, pensou-se em adotar um enfoque que facilite a utilização de uma interface gráfica para o pesquisador de redes neurais.

4.2 Modelo do Simulador

Para a elaboração do modelo do simulador buscou-se atingir os objetivos anteriormente elencados, e tais exigências, combinadas com a análise de outras aplicações distribuídas, direcionaram para uma implementação baseada no modelo cliente-servidor.

O paradigma cliente-servidor vem sendo utilizado de forma crescente nos últimos anos e é, certamente, o modelo mais utilizado de computação distribuída, tanto que sua introdução chega a ser comparada com a mudança que houve quando os computadores mudaram de processamento em *batch* para *on-line* [REN 94].

O modelo é composto por dois tipos de entidades: um servidor e, pelo menos, um cliente, que podem executar em diferentes máquinas conectadas ou, até mesmo, em uma máquina apenas.

O cliente é um processo ativo que executa algum processamento local e solicita a execução de tarefas específicas ao servidor. Um cliente pode utilizar mais de um servidor ao mesmo tempo, sejam servidores iguais ou com serviços diferentes.

O servidor, como o próprio nome diz, provê serviços para clientes. Trata-se de um processo reativo que fica aguardando requisições para processá-las e, se for o caso, enviar os resultados para quem solicitou o serviço.

Embora o conceito seja mais restrito do que outros como *peer-to-peer*, por exemplo, em que os processos tem mais liberdade para interagir um com o outro, ele permite construir diferentes meios de acesso — através de diferentes clientes — possibilitando formas variadas de uso e isso é importante dentro do escopo deste trabalho. Além disso, sua simplicidade conceitual facilita o desenvolvimento de clientes.

Inúmeras aplicações se baseiam neste paradigma, que está na base dos conceitos de computação distribuída, e pode-se citar como exemplo sua crescente utilização em sistemas de bancos de dados com acesso distribuído e os serviços definidos pelo conjunto de protocolos TCP/IP, largamente utilizados e, hoje, um padrão de fato.

A seguir está reapresentada a lista de objetivos para o simulador e, para cada item, é descrito como o paradigma cliente-servidor se adapta a ele:

1. Transparência: o uso de um servidor para fazer a comunicação com o usuário, que executa um programa cliente, já é uma realidade bastante comum nas redes de computadores e tem mostrado sua eficácia para esconder os detalhes no lado do servidor. Em outras palavras, o servidor implementa uma camada conceitual que será a única acessada pelo usuário, com um conjunto de serviços

bem definidos que não mudarão mesmo que as redes neurais estejam implementadas em diferentes computadores, inclusive com arquiteturas diferentes.

2. Modelagem orientada para multicomputador: o uso de cliente-servidor em máquinas MIMD com memória distribuída não apresenta nenhum problema, permitindo a utilização de tal arquitetura através do uso de mensagens para comunicação com o cliente e com os processos que implementarão os modelos das RNAs.

3. Simulador baseado em blocos: através de um servidor, tem-se o controle centralizado do simulador o que permite fácil tratamento dos diferentes blocos que o compõem. Por exemplo, para agregar um novo modelo de redes neurais ao simulador, basta sua implementação e o acréscimo de novos serviços, no servidor, que tratarão desse modelo.

4. Distribuição de carga: a distribuição de carga estática pode ser controlada pelo servidor sem que isso acarrete sobrecarga de comunicações, que haveria sem controle centralizado. Haverá um processo monitorando constantemente a carga dos processadores e atualizando uma base de dados que será acessada pelo servidor para determinar que recursos alocar. Sob esse aspecto têm se uma distribuição de carga estática por serviço, mas pode-se fazer uma analogia com uma distribuição de carga dinâmica se for enfocado como sendo a carga do simulador a soma das cargas de todos os serviços ao longo do tempo.

5. Interatividade: um servidor, a priori, não proporciona nenhum tipo de interatividade, entretanto toda a interatividade fica por conta do cliente que pode implementar qualquer tipo de interface para se comunicar com o servidor. Pode-se optar por uma interface espartana, orientada a caracter para uso com terminais, como também pode-se utilizar uma interface gráfica para prover alta facilidade de uso ou até mesmo uma linguagem de programação que implemente chamadas aos serviços oferecidos pelo simulador, de forma a atender plenamente aos requisitos enunciados por Muller.

Na figura 4.1 pode-se ver um esquema de execução do modelo proposto. Há um ou mais clientes que utilizarão serviços do simulador através de uma rede de interconexão entre suas máquinas. O servidor, por sua vez, estará executando em uma máquina qualquer que esteja conectada à rede.

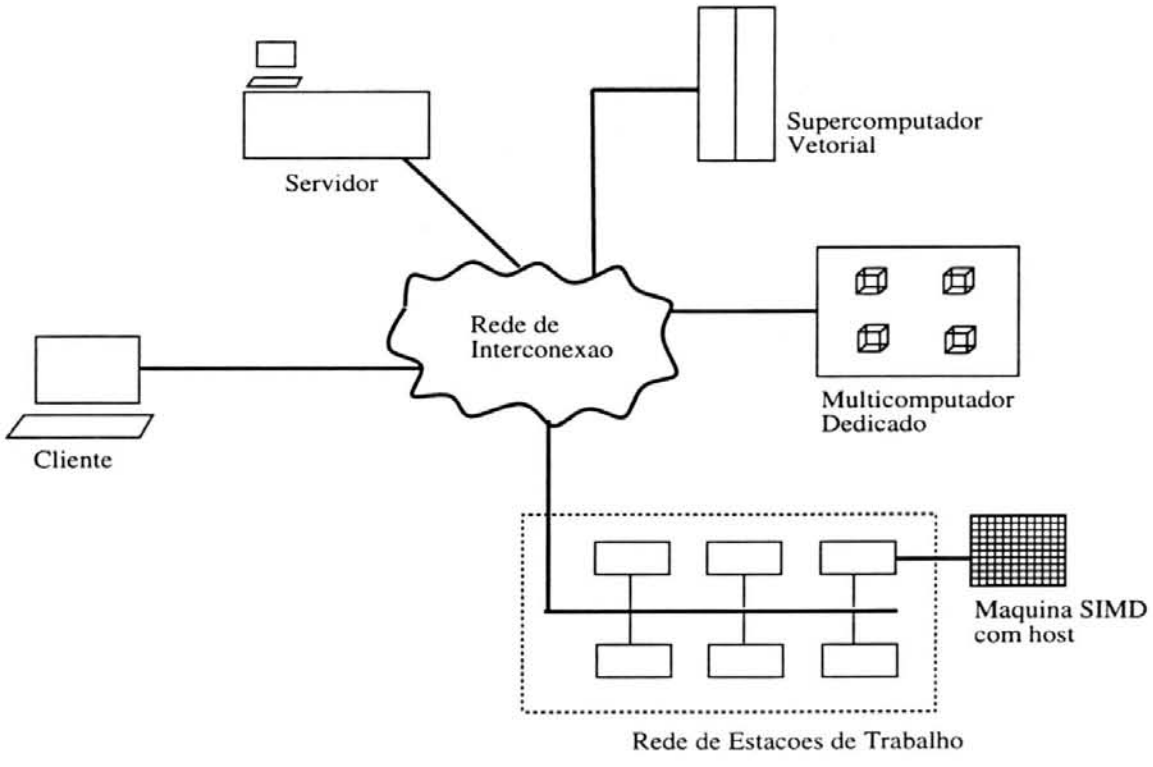

FIGURA 4.1 — Esquema de Execução do Modelo

Diversos outros computadores podem estar conectados a essa rede e podem ser utilizados para a implementação dos algoritmos de redes neurais de forma paralela, como na figura: um supercomputador vetorial, uma rede de estações ou um multicomputador dedicado. O servidor fornecerá uma interface consistente e independente das plataformas de *hardware* para o acesso aos serviços ao mesmo tempo que poderá suportar diferentes paradigmas de paralelização para diferentes modelos de redes neurais, servindo tanto para pesquisa em redes neurais como em processamento de alto desempenho.

Por outro lado, a escolha de uma específica plataforma de *hardware* também deverá ser suportada, pois o usuário muitas vezes possui conhecimento de que isso pode levar a um melhor desempenho, de acordo com o modelo.

A inclusão de novas implementações de modelos de RNAs não se limita a implementações paralelas/distribuídas, embora este seja o enfoque deste trabalho, e não deve interferir com os outros modelos.

De um modo geral, pode-se fazer uma forte analogia com o paradigma de orientação a Objetos, uma vez que os serviços são ativados através de mensagens, da mesma forma que os métodos de um objeto. De forma similar, o servidor esconde os detalhes de implementação das redes neurais da mesma forma que o encapsulamento de dados e código proporcionado pelo modelo de objetos. Por esse motivo, o modelo do simulador será apresentada em forma de diagrama de análise orientada a objetos [COA 92], embora ele não tenha sido implementado em uma linguagem orientada a objetos, visando facilitar o entendimento de sua estrutura.

4.2.1 Descrição dos Serviços

Esta seção descreve os serviços oferecidos pelo servidor e que são acessados através de RPCs. Os detalhes de uso de cada um dos serviços podem ser encontrados no apêndice A, no qual há um exemplo de chamada para cada um dos serviços e a definição da interface na linguagem de programação C.

4.2.1.1 Serviços Gerais

Os serviços aqui descritos são comuns para todos os modelos de redes neurais suportados pelo servidor e devem ser implementados em todos os modelos que venham a ser acrescentados.

- Encerra a execução de uma rede previamente criada, liberando todos os recursos alocados (processador, memória, etc). Como argumento é passado o identificador da rede, que é retornado pelo servico de criação.

- Salva a estrutura da rede neural em um arquivo de modo que seu estado possa ser congelado para uso futuro. Recebe o nome do arquivo em que a rede será colocada.

- Carrega os dados de uma rede previamente salva pelo serviço anterior. Recebe o nome do arquivo onde a rede está armazenada.

- Determina o estado de uma rede. Recebe o identificador e retorna seu estado atual, informando se está inativa, em aprendizado, em teste, etc.

4.2.1.2 Serviços das Redes CNM

Aqui são enumerados os serviços disponíveis no servidor para a manipulação de redes neurais CNM.

- Cria uma nova rede neural CNM no servidor, gerando a alocação dos recursos necessários à sua execução. Recebe o número de hipóteses, número de neurônios de entrada e a ordem da rede a ser criada. Retorna o identificador da rede, que será usado nos acessos posteriores.

- Dispara o processo de aprendizado inicial (*starter learning*) na rede especificada. Recebe os limiares de aceitação e poda (*acceptance and prunning thresholds*), o número de casos de teste e o nome do arquivo que os contém.

- Executa o algoritmo da rede para uma série de valores de entrada e retorna o estado das hipóteses. Após o aprendizado, este serviço é utilizado para testar se a rede foi treinada adequadamente e também para uso normal da rede. Para efeito de implementação este serviço foi dividido em dois, conforme pode ser visto no apêndice A.

- Dispara o processo de aprendizado incremental (*incremental learning*) de uma rede CNM.

- Estabelece o valor do limiar de poda (*prunning threshold*), permitindo variá-lo entre experimentos.

- Estabelece o valor do limiar de aceitação (*acceptance threshold*), com o mesmo objetivo.

4.2.1.3 Serviços das Redes Back-Propagation

- Cria uma rede *back propagation*, inicializando as estruturas de dados e alocando os recursos necessários. Recebe o número de unidades de saí e de entrada, o número de camadas intermediárias e o número de neurônios por camada, o coeficiente de aprendizado e o coeficiente de amortecimento. Retorna o identificador da rede criada;

- Dispara o processo de treinamento. Recebe o nome do arquivo com os casos de teste, o número de casos, o número máximo de iterações e o valor do erro a ser atingido.

- Testa a saída da rede para uma dada entrada;

- Estabelece o coeficiente de aprendizado;

- Estabelece o coeficiente de amortecimento.

4.3 Descrição da Implementação

Na seção anterior, foi discutido o modelo conceitual do simulador e concentrou-se em o que ele deve oferecer. Esta seção apresenta como foi realizada a implementação para atingir aqueles objetivos e oferecer os serviços apresentados.

O ambiente utilizado para a implementação consistiu de uma rede de estações de trabalho da *Sun Microsystems, Inc* com o sistema operacional SunOS 4.1, conectadas através de uma rede Ethernet de 10Mbit/s com o conjunto de protocolos TCP/IP.

4.3.1 Ferramentas Utilizadas

A seguir são apresentadas as ferramentas utilizadas nesta implementação que estão disponíveis no sistema operacional utilizado.

4.3.1.1 Chamada de Procedimentos Remotos (*Remote Procedure Call* - RPC)

O sistema operacional SunOS oferece uma facilidade para a construção de sistemas cliente-servidor conhecida como *Remote Procedure Call* que consiste na chamada de um procedimento que será executado em outro programa, podendo este estar localizado em qualquer máquina conectada com TCP/IP.

No modelo RPC, existe um programa servidor que cadastra sua identificação em uma porta de comunicação da máquina onde está executando e passa a atender todas as requisições que chegam a essa porta. Do outro lado, há programas cliente que enviam essas requisições e que devem saber previamente onde o servidor está cadastrado.

No âmbito deste trabalho, RPC foi utilizado para fazer a comunicação entre o servidor e os clientes, oferecendo uma interface bem conhecida e consistente para o acesso dos serviços.

4.3.1.2 Intercomunicação entre Processos

Introduzidos na versão 4.3 da *Berkeley Software Distribution* (BSD) os *sockets* implementam canais bidirecionais *full-duplex* para comunicação entre processos. Estão baseados no conceito de descritores de arquivo e permite as mesmas operações que são normalmente aplicadas a estes.

Os *sockets* foram utilizados neste trabalho para realizar a comunicação entre processos executando em paralelo, conforme será descrito na paralelização a seguir.

4.3.1.3 Representação de Dados

Quando se trabalha com diferentes arquiteturas e sistemas operacionais, é necessário que os dados trocados por processos obedeçam a um padrão. Neste trabalho foi utilizado o padrão *External Data Representation* (XDR) que se trata de um formato de representação de dados definido pela *Sun Microsystems, Inc*, independente de plataforma, que visa ser utilizado na troca de informações entre máquinas de arquiteturas diferentes. O mecanismo de RPC utiliza XDR, o que permite que um cliente possa acessar um servidor executando em uma máquina com arquitetura diferente da sua.

A especificação XDR utiliza conversores locais para o formato específico da máquina e o padrão XDR para o tráfego em redes de comunicação.

4.3.1.4 Compilador e Depurador

Todos os programas foram implementados na linguagem C de programação e foi utilizado o compilador padrão do SunOS 4.1 que não segue a especificação ANSI. Devido a isso o código foi escrito com o cuidado de portabilidade e a maioria é totalmente portável para outros sistemas similares ao Unix. A única restrição se refere ao uso da biblioteca de processos leves do SunOS.

Na parte de depuração, o ambiente não fornece nenhum suporte para depuração de tarefas paralelas, tendo sido utilizado o depurador seqüencial também fornecido com o sistema operacional.

O processo de depuração de um sistema paralelo/distribuído com ferramentas seqüenciais não é trivial. Para o caso deste sistema foi adotado um enfoque de módulos de forma que cada pedaço pudesse ser testado seqüencialmente antes de compor o sistema paralelo. Assim, começou-se a depuração com os módulos mais simples implementando-se um programa que testava todas as funções desse módulo e, após a certificação de sua funcionalidade, o módulo era incorporado ao sistema.

Antes de adotar esse enfoque tentou-se depurar o sistema como um todo, mas isso logo se mostrou ineficaz, dada a quantidade de variáveis que interferiam nos testes. O método de depuração por módulos é utilizado normalmente em programação seqüencial, no âmbito de funções e procedimentos, e aqui provou ser a melhor forma para se depurar um sistema paralelo dispondo apenas de ferramentas seqüenciais, utilizando-o no âmbito de programas/processos.

4.3.1.5 Threads

Um processo é um programa em execução e apresenta um fluxo de execução definido por um registrador normalmente designado como *instruction pointer* que define a próxima instrução a ser executada e que, normalmente, está duplicado em uma estrutura do sistema operacional enquanto a UCP está executando outro programa.

O conceito de *threads* amplia essa visão de processo para um programa em execução com múltiplos fluxos de execução. Isso difere um pouco da programação concorrente tradicional, pois há apenas um processo executando e não é necessário alocar a mesma quantidade de recursos do sistema operacional para uma *thread* em relação a um processo. Por esse motivo *threads* são também chamadas de processos leves (*lightweight processes*) em contraposição aos processos tradicionais.

Os múltiplos fluxos de execução dentro de um processo são utilizados para reduzir o tempo em que o processo fica sem fazer nada ou fica fazendo *polling*. Com o uso de *threads* propicia-se o surgimento de processamento oportunista dentro de um processo, pois as tarefas sem atividades são bloqueadas e uma outra tarefa é escolhida para continuar executando, sem interferência do programador.

Em aplicações muito similares, onde não foi propiciado processamento oportunista houve uma sensível perda de eficiência na paralelização conforme pode ser comparado dos resultados obtidos em [CAL 92] e em [SCH 92b].

Adicionalmente, a biblioteca de *threads* utilizada fornece primitivas de comunicação para sincronismo e envio de mensagens entre elas.

4.3.2 Implementação do Servidor

O modelo de implementação do servidor pode ser visto na figura 4.2, onde é apresentado com um modelo de análise orientada a objetos, de acordo com a notação usada em [COA 92].

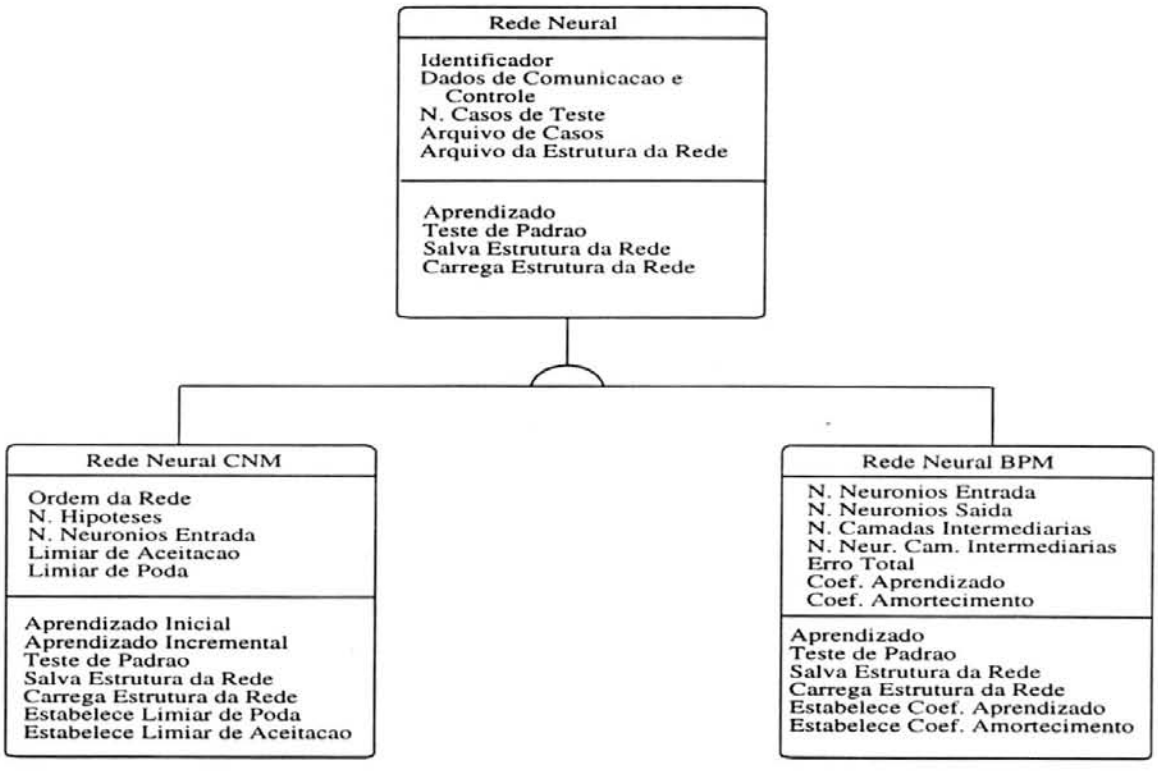

FIGURA 4.2 — Modelo do Simulador em Notação Orientada a Objetos

Na figura está apresentado o modelo em alto nível de abstração [1], onde pode-se ver uma estrutura de generalização/especialização que relaciona a classe genérica Redes Neurais com suas especializações. Como citado anteriormente, o modelo orientado a objetos está sendo utilizado para apresentar a estrutura do simulador, porém não foi utilizada uma linguagem orientada a objetos para a implementação.

[1] Estão representados apenas os métodos e atributos importantes para as RNAs, outros atributos concernentes à distribuição e comunicação foram suprimidos para simplificar.

Para que o mapeamento do modelo em uma linguagem orientada a objetos atinja os objetivos deste trabalho, é necessário que haja suporte a objetos distribuídos, *i. e.*, objetos que podem ser alocados em máquinas diferentes. Caso isso não exista, o desempenho do simulador fica comprometido, pois tem-se apenas um processador disponível.

Embora a implementação tenha sido realizada em linguagem C, o modelo orientado a objetos descreve claramente a estrutura do código, principalmente no que diz respeito a sua modularidade.

Na figura, pode-se ver que cada modelo de rede neural é tratado como uma classe distinta, herdando alguns atributos de uma classe mais genérica e especializando seus serviços de acordo com as caraterísticas de cada modelo.

O esquema de execução será composto por diversos processos, e um exemplo pode ser visto na figura 4.3. Nessa figura, a linha pontilhada delimita uma visão virtual do simulador, *i. e.*, o simulador pode ser compreendido como tudo o que está dentro da área delimitada. Ali podemos ver o processo do servidor, propriamente dito, e outros processos que representam instâncias de RNAs em execução. Observa-se que essa representação também é virtual, pois a implementação de cada modelo pode gerar diversos processos, conforme será visto mais adiante.

Um outro processo, até então não mencionado, está representado na figura sob o nome de Monitor de Desempenho. Esse processo monitora a carga das máquinas que estão disponíveis para o simulador e repassa a informação ao servidor em intervalos de tempo. Posteriormente, o servidor utilizará essas informações para determinar que máquinas alocar, quando da criação de novas instâncias. Como já mencionado, isso configura um esquema de distribuição estática de carga.

Em um nível mais detalhado, a implementação do programa servidor consistiu de três partes: uma para estabelecer a configuração inicial do serviço de RPC, uma segunda com a rotina que seleciona o serviço quando há uma requisição e uma última parte com a implementação da chamada aos serviços. Observa-se que esta

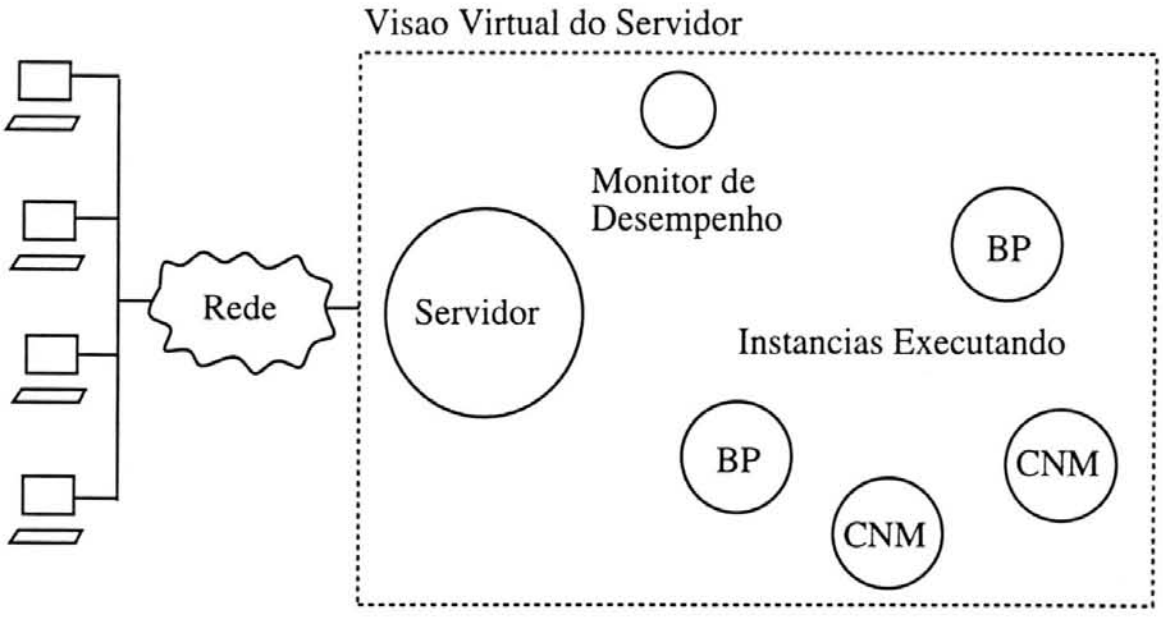

FIGURA 4.3 — Esquema de Execução

última não implementa os serviços, pois de acordo com a estrutura de camadas do simulador, o servidor deve apenas atuar como ponto de entrada das requisições. Os serviços, assim como a própria estrutura de dados da rede, ficam em outros processos que os implementam, de forma independente.

Dessa forma, o algoritmo executado pelo servidor pode ser resumido nos seguintes passos:

1. Inicializa as estruturas de dados e registra seu serviço de RPC;
2. Repete indefinidamente
   (a) Aguarda requisição de serviço;
   (b) Seleciona o serviço e verifica parâmetros;
   (c) Invoca o serviço requisitado;
   (d) Aguarda o fim do serviço, pega os resultados e retorna para quem requisitou;

4.3.2.1 Considerações sobre os Serviços

Como pode ser observado na descrição do algoritmo anterior, o servidor executa apenas o repasse das requisições, o que caracteriza a simplicidade de seu algoritmo. No entanto, a definição da implementação dos serviços não é tão simples e merece algumas considerações.

A implementação do serviço que dispara o treinamento em uma instância mereceu especial consideração, pois o volume de dados envolvidos no processo de treinamento é, freqüentemente, bastante elevado.

O serviço requisita que uma determinada instância inicie seu processo de treinamento e, para isso, deve passar parâmetros de controle do treinamento, que são específicos do modelo, e toda ou parte da base de casos de treinamento. Deve-se analisar, então, como proceder para transferir a base de dados para a máquina responsável pelo treinamento de uma determinada parte da base. Isso pode ser feito de duas maneiras: através de envio explícito dos casos ou de sua colocação em um lugar de acesso comum, tipicamente um arquivo.

Dado o grande volume de dados envolvido nos treinamentos de problemas complexos, optou-se pelo uso de arquivos e não do envio de mensagens. Como citado no capítulo 3 o custo de envio de mensagens neste ambiente é muito alto e além disso o modelo de sistema de aquivos distribuído permite que as outras máquinas acessem, de forma transparente, os dados sem que isso implique modificações no simulador. Assim, faz-se uso de uma facilidade já disponível no sistema operacional para agilizar a manutenção, portabilidade e desenvolvimento do simulador como um todo.

Contudo, isso traz uma limitação quanto a disponibilidade dos sistemas de arquivos, pois nem sempre é possível montar um sistema de arquivos em todas as máquinas que se queira utilizar. Apesar disso, no âmbito deste trabalho, essa limitação não foi considerada comprometedora e, de qualquer forma, é muito simples acrescentar outro serviço que, em vez de arquivo, utilize comunicação direta, dada a estrutura do simulador.

No serviço de inicialização, uma vez que é utilizado um mesmo espaço de nomes de arquivos, não precisa ser efetuado o envio do programa executável para a máquina que o executará. Em vez disso, apenas o nome dele é enviado e ele é acessado através do sistema de arquivos distribuído.

Outros serviços beneficiados são aqueles que gravam e lêem a estrutura de redes a partir de arquivos, pois podem efetuar operações locais de gravação/leitura, enquanto enviam os dados para uma outra máquina.

Serviços de configuração, como estabelecer atributos das instâncias, são implementados de forma direta através do envio dos valores pelas primitivas de comunicação.

4.3.3 Implementacão de Clientes

Foram implementados alguns clientes, visando testar o simulador e medir tempos de execução, todos eles com interface orientada a caracter e um deles foi selecionado para ser utilizado como base da implementação de clientes por outros desenvolvedores.

No apêndice A encontram-se partes do código desse cliente com exemplos de requisições de serviços para que outros clientes, específicos para as necessidades dos pesquisadores de redes neurais, sejam desenvolvidos.

Orienta-se para que os clientes desenvolvidos tenham interface gráfica e façam uso de vantagens do simulador como sua versatilidade, que permite a execução do cliente em qualquer máquina conectada a rede Internet, se o servidor estiver em uma máquina nessa rede.

Isso permite, com as devidas implementações, que pesquisadores de uma instituição possam utilizar os recursos de outra, como um supercomputador, de forma facilitada e já orientada para sua área de atuação, não exigindo treinamento em ferramentas específicas de um computador ou sistema operacional.

4.3.4 Paralelização do Modelo CNM

Esta seção apresenta a forma como foi realizada a paralelização do modelo CNM, tendo como objetivo principal distribuir a ocupação de memória entre os processadores.

O modelo *Combinatorial Neural Model* apresenta concorrência lógica bastante similar a outras redes neurais artificiais, *i. e.*, os elementos processadores (unidades ou neurônios) são potencialmente paralelos, pois dependem apenas de seus dados de entrada para determinar seu estado de ativação. Contudo a exploração de paralelismo nesse nível não é adequada a ambientes de processamento distribuído. Além disso, o elevado número de conexões entre as unidades sugere que não é simples implementar uma rede de interconexão eficiente para tal modelo.

Numa primeira tentativa, poder-se-ia tentar paralelizar o algoritmo de treinamento dos casos. Entretanto os acumuladores, que são alterados a cada caso treinado, estabelecem uma dependência de dados entre cada passo da iteração. Parece possível resolver essa dependência através do treinamento de sub-conjuntos em diferentes máquinas e da combinação dos valores dos acumuladores antes do processo de poda e normalização. Porém não foi seguido esse enfoque porque ele não resolve o grande problema de CNM que é o uso de memória.

Através da observação da estrutura de interconexão das unidades, como exemplificado na figura 4.4, percebe-se que cada uma das hipóteses está conectada com todas as unidades da camada combinatória e com todas as unidades da camada de entrada. Dessa forma, o treinamento dos casos não apresenta dependência entre as redes de cada hipótese, *i. e.*, as hipóteses podem ser treinadas de forma independente.

Uma vez que isso é possível, cada processador faz o treinamento de uma hipótese e implementa apenas as conexões necessárias àquela saída. Isso permite que a necessidade de memória seja distribuída entre os processadores possibilitando a utilização de redes de maior ordem através do processamento cooperativo.

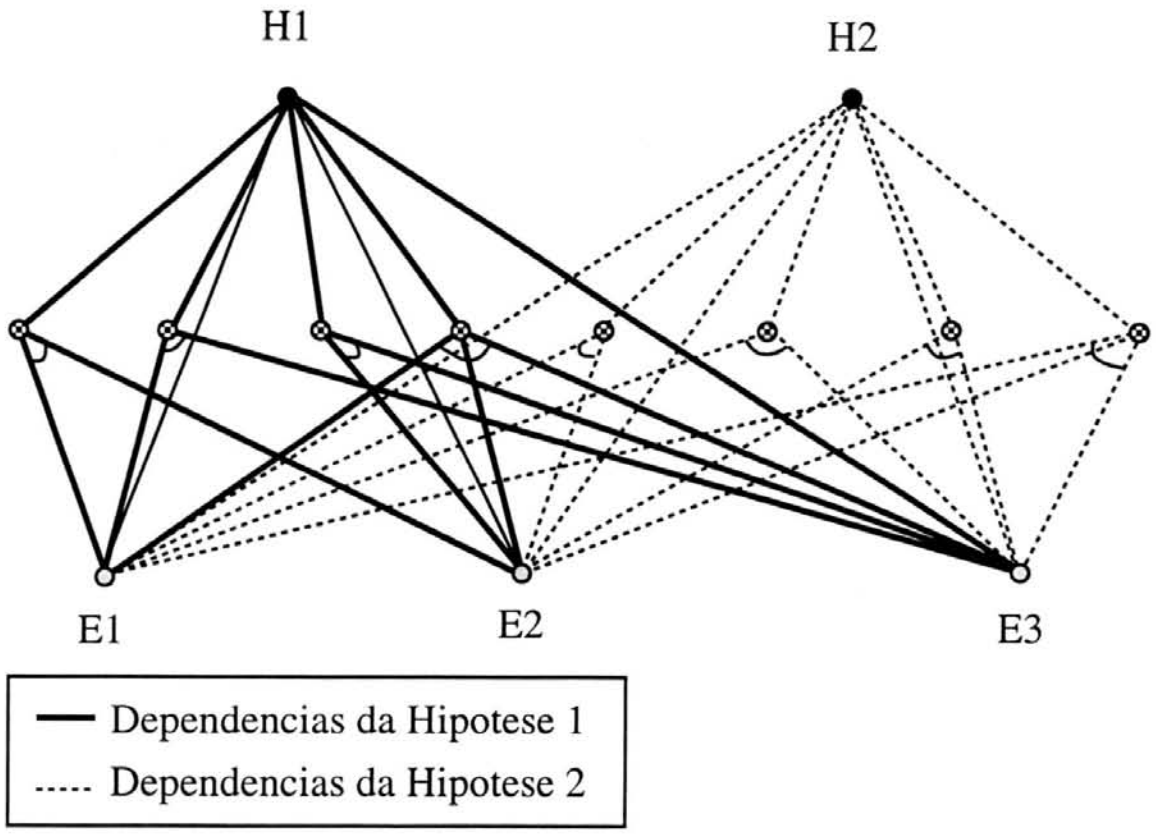

FIGURA 4.4 — Estrutura de Dependência de Dados em uma Rede CNM

Esse enfoque de paralelização se enquadra dentro do paradigma de paralelismo de dados (*Data Parallel*), uma vez que teremos diferentes dados de entrada e o mesmo algoritmo sendo aplicado sobre eles. A primeira vista, poder-se-ia pensar que os dados de entrada são os mesmos, pois são as evidências; contudo os alvos de cada uma das hipóteses também são dados de entrada para o algoritmo, e eles são diferentes para cada hipótese.

No capítulo 3 viu-se que a exploração de paralelismo de dados pode ser feita de forma eficiente em um ambiente de processamento distribuído e isso já foi comprovado com experimentos [SCH 92b].

Pela eficiência demonstrada nesses e em outros experimentos e por sua simplicidade conceitual, optou-se por um modelo de implementação baseado no paradigma mestre-escravo, que possui um processo mestre coordenando uma série de processos escravos, sendo que estes implementam o algoritmo que será aplicado a cada uma das porções dos dados.

Além disso, esse paradgima já foi utilizado em outro trabalho sobre paralelização de RNAs, com bons resultados [BAR 90]

4.3.4.1 Implementação do CNM

Para a implementação do paradigma mestre-escravo foi utilizada uma biblioteca para distribuição de aplicações desenvolvida anteriormente pelo autor. Algumas modificações foram necessárias na biblioteca original para suportar algumas necessidades específicas de comunicação.

A independência de dados do *Combinatorial Neural Model* permitiu que os escravos executassem uma variação do algoritmos seqüencial que introduzia comunicação em pontos bem controlados e que não degradava o tempo total devido ao pequeno número de troca de mensagens.

Partindo de uma versão sequëncial, foi necessário isolar o algoritmo intensivo computacionalmente no programa escravo e introduzir pontos de comunicação no início e no fim dos procedimentos, uma vez que não era necessária a interação entre os escravos para realizar o treinamento da rede.

4.3.5 Paralelização do Modelo Back Propagation

Por ser um modelo de larga utilização, o *Back Propagation* teve muitos estudos e experimentos no que concerne a sua paralelização. Não somente a utilização de computadores de propósitos gerais tem sido considerada, como também o desenvolvimento de *chips* neurais específicos tem sido levado a cabo por diversos pesquisadores, inclusive para outros modelos de redes neurais.

Neste trabalho foram adotados dois enfoques para tratar a aceleração do processamento em ambiente distribuído. Um dos enfoques baseia-se na paralelização dos algoritmos e o outro nas necessidades apresentadas pelo trabalho de Fernan-

des [FER 94], que visa determinar parâmetros otimizados para as redes através de experimentação.

4.3.5.1 Paralelização dos Algoritmos

O modelo, conforme descrito no capítulo 2, apresenta algoritmos com alta dependência de dados o que é evidenciado por sua alta densidade de conexões entre as unidades.

A paralelização deste modelo já foi objeto de estudo em outros trabalhos e aceita-se que há, basicamente, dois enfoques para sua paralelização [LIU 94]:

- Particionamento da Rede: aloca diferentes unidades e pesos em diferentes processadores, permitindo cálculo em paralelo da atualização de pesos e da ativação e erros das unidades;
- Particionamento dos Padrões: divide os padrões entre diferentes processadores, permitindo o ajuste dos pesos em paralelo.

No mesmo artigo, Liu apresenta um enfoque para paralelizar o modelo em um máquina CM-5, usando particionamento da rede. O algoritmo se vale de uma característica da CM-5, que é uma rede de controle dos processadores, para efetuar uma operação em todos os processadores ao mesmo tempo e evitar troca de mensagens explícita entre todos eles a cada iteração do algoritmo de aprendizado.

Contudo, em ambiente distribuído é inviável difundir (*broadcast*) uma mensagem para todos os processadores a cada iteração, pois a velocidade do meio de comunicação é muito inferior, além de ser compartilhado entre os processadores e outros usuários da rede. Desse modo, esse algoritmo não terá desempenho eficiente, pois o tempo de comunicação é muito maior que o tempo utilizado para a iteração do algoritmo, *i. e.*, algoritmos *fine-grained*[2] não apresentam ganho de eficiência em

[2]O algoritmo em questão se enquadra como *coarse-grained* para alguns autores.

ambiente distribuído, pelo contrário, seu desempenho em paralelo será inferior ao desempenho seqüencial.

Em outro artigo [PÉT 94], Pétrowksi analisa 3 diferentes tipos de implementações paralelas para *Back Propagation* e sugerere algumas recomendações para escolha entre as diferentes formas de paralelização, fornecendo funções *upper bound*. Ele analisa implementações em arquitetura matricial bidimensional com ligações *wrap around* (*2D torus*), em matriz *wave-front* e em hipercubo.

A implementação que mais se assemelha ao ambiente em questão neste trabalho é o hipercubo, que utiliza particionamento dos padrões de entrada como técnica de paralelização e implementa um algoritmo que atualiza globalmente os dados de todos os processadores após um certo número de iterações.

Em relação ao algoritmo anterior, poposto para a CM-5, este apresenta a vantagem de aumentar o tempo que o processador fica trabalhando antes de difundir seus dados. Contudo, a difusão gera um número muito grande de mensagens direcionadas para cada um dos nodos e aplica-se a máquinas fortemente acopladas como o hipercubo em análise no artigo, onde há $log_2 P$ canais de comunicação dedicados para cada processador, de um total de P processadores.

Para ambientes distribuídos este algoritmo é mais viável que o utilizado por Liu, entretanto o número de iterações que vai ser realizado entre cada difusão é o fator determinante da eficiência da paralelização.

Como observado por Pétrowski, pode-se usar uma forma de aceleração que é realizar ajuste de pesos apenas após a apresentação de um bloco composto por b padrões. Desse modo, a difusão ocorre apenas a cada b iterações do algoritmo, aumentando o grão do paralelismo. Poder-se-ia supor que, então, aumentando o valor de b seria suficiente para garantir uma boa paralelização, mas isso não é verdade, pois o algoritmo de aprendizado busca convergir para um estado de erro mínimo e o aumento de b interfere nessa convergência, modificando o número de iterações que

ele leva para atingir esse estado. No capítulo 6 há uma discussão mais detalhada sobre essa característica.

Tem-se, assim, uma relação de compromisso com o valor de b que é dependente do conjunto de dados de entrada, mas que pode levar a uma paralelização eficiente. Por esses motivos, optou-se por utilizar paralelização baseada na técnica de particionamento do conjunto de padrões e realizar experimentos para testar a eficiência, alterando o valor do passo de comunicação (b).

4.3.5.2 Otimização de Parâmetros por Experimentação

Como visto na seção anterior, o principal problema em ambiente distribuído é o tempo de comunicação, o que leva a explorar o paralelismo em grãos maiores (*coarse-grained*) para diminuir o fluxo de mensagens.

No trabalho de Fernandes [FER 94], busca-se determinar valores para os parâmetros ajustáveis do algoritmo *Back Propagation* que otimizem seu treinamento. Isso é feito de forma experimental, executando o algoritmo com diferentes parâmetros e testando a eficiência do treinamento.

Enfocando uma série de experimentos como uma única atividade lógica, tem-se que a dependência entre cada uma delas está apenas na variação dos parâmetros, o que permite explorar o paralelismo ao nível de processo, executando diferentes experimentos de forma paralela.

Propõe-se, então, uma forma de automatizar essa paralelização através do simulador aqui apresentado, que tem como objetivo fornecer uma interface simples que oculte do pesquisador a paralelização e apresente uma máquina virtual para a realização de todos os experimentos.

Para isso deveriam ser incorporados novos serviços ao simulador, que permitissem disparar um treinamento de *Back Propagation* seqüencial, pois agora não há

interesse em acelerar apenas um treinamento, mas sim obter o resultado de diversos treinamentos enfocados como uma única tarefa a ser paralelizada.

Com o enfoque aberto e orientado a blocos do simulador isso foi acrescentado através da inclusão de um novo serviço que cria uma rede neural *Back Propagation* em um processador apenas, sem paralelização, mascarando isso do desenvolvedor e atendendo aos outros serviços normais do modelo sem alteração.

4.3.5.3 Implementação do BPM

Ao contrário da implementação do *Combinatorial Neural Model* que permitiu isolar partes da rede, o algoritmo *Back Propagation* possui grande dependência de dados.

Baseado no artigo de Pétrowski foi realizada uma implementação distribuída do algoritmo que executa o algoritmo seqüencial nos escravos e, a cada b iterações, transfere seu conhecimento para o processo mestre. O mestre combina o conhecimento especializado das redes dos escravos, representado por seus pesos e *bias*, e gera uma nova rede, a qual tem seu treinamento checado contra a base de casos. Caso o erro esteja abaixo do limite especificado, o treinamento é encerrado; mas, caso contrário, a nova rede é repassada para os escravos que partirão daquele estágio para continuar o treinamento, ou seja, com o conhecimento de todos os escravos difundido.

Percebe-se claramente uma elevada taxa de comunicação, que pode estar sob controle através do valor de b. No capítulo 6 a variação do valor de b é analisada para o problema do ou-exclusivo.

Uma característica interessante da implementação foi o uso de uma *pipeline* para diminuir o tempo de espera dos escravos. O problema consiste no seguinte: os processos escravos realizam b iterações e retornam os dados para o mestre, que vai calcular o erro da rede resultante. Contudo nesse tempo os escravos ficam aguardando, sem fazer nenhum processamento.

Abandonando esse enfoque serial, pode-se realizar as próximas b iterações em paralelo com o cálculo do erro no mestre. Assim os escravos sempre farão b iterações a mais do que o necessário, contudo não ficarão aguardando o tempo que o mestre dispender para calcular o erro das b iterações anteriores, em cada conjunto de b iterações.

5 Resultados Obtidos

Esta seção apresenta os resultados obtidos com a paralelização dos dois modelos de RNAs discutidos neste trabalho. A seguir, estão apresentados os resultados quantitativos obtidos com a paralelização dos modelos e, no capítulo seguinte, há uma discussão qualitativa sobre a proposta, enfocando seus pontos positivos e negativos, bem como alternativas de aperfeiçoamento para o simulador.

Todos os testes de desempenho foram realizados em estações de trabalho com o mesmo poder computacional, da *Sun Micrsystems*, com o sistema operacional SunOS 4.1. Os tempos foram medidos quando não havia usuários usando as máquinas, tipicamente à noite e em finais de semana, pois não seria possível avaliar o desempenho da paralelização se as medidas fossem realizadas em horários normais com usuários conectados. Foram realizados diversos experimentos e os tempos aqui apresentados refletem a média de várias medidas para cada uma das tabelas.

5.1 Resultados do CNM

A paralelização do CNM baseou-se na independência de dados entre as subredes que formam as hipóteses do modelo. Seu objetivo principal foi distribuir a alta ocupação de memória entre diversas máquinas de modo que fosse possível utilizar redes com maior ordem. Como resultado adicional, obteve-se uma paralelização

que propicia um aumento de desempenho diretamente proporcional ao número de hipóteses.

Para os resultados aqui apresentados foi utilizada uma base de casos de doenças renais, a mesma utilizada em [GUA 94]. Essa base é composta de 381 casos de doenças renais colhidos dos prontuários da Escola Paulista de Medicina e foi dividida em 254 casos para treinamento e 127 para teste. A rede possui 58 unidades de entrada para as diferentes evidências e 4 hipóteses para as doenças. Os parâmetros utilizados foram: aceitação = 0.3 e poda = 0.0.

Na tabela 5.1 estão apresentados os tempos de execução para o algoritmo de aprendizado inicial (*starter learning*) nas quatro máquinas que foram utilizadas para os testes, com ordem da rede igual a 2.

TABELA 5.1 — Tempos do Algoritmo CNM *Starter Learning* Seqüencial com Ordem 2

Máquina	Tempo
marfim	1593s
coral	1578s
ambar	1578s
perola	1601s
Média	1587.5s

Na tabela 5.2 estão apresentados os tempos de execução para o algoritmo de aprendizado incremental (*incremental learning*) nas quatro máquinas que foram utilizadas para os testes, com ordem da rede igual a 2. Foram realizadas 5 iterações do algoritmo incremental.

TABELA 5.2 — Tempos do Algoritmo CNM *Incremental Learning* Seqüencial com Ordem 2

Máquina	Tempo
marfim	4930s
coral	4913s
ambar	4875s
perola	4964s
Média	4920,5s

A seguir estão os tempos obtidos na execução paralela dos algoritmos nas mesmas máquinas. Nas tabelas seguintes, cada máquina realizou apenas um quarto da tarefa, por isso seus tempos individuais são menores e eles evidenciam que a distribuição de carga, embora seja estática, está sendo feita de forma igualitária entre os processadores.

Na tabela 5.3 está o tempo de execução para o algoritmo de aprendizado inicial (*starter learning*) nas quatro máquinas, com ordem da rede igual a 2.

TABELA 5.3 — Tempos do Algoritmo CNM *Starter Learning* Paralelo com Ordem 2

Máquina	Tempo
marfim	400s
coral	407s
ambar	397s
perola	403s
Média	401,75s

Na tabela 5.4 está o tempo de execução para o algoritmo de aprendizado incremental (*incremental learning*) nas quatro máquinas, com ordem da rede igual a 2.

TABELA 5.4 — Tempo do Algoritmo CNM *Incremental Learning* Paralelo com Ordem 2

Máquina	Tempo
marfim	1255,0s
coral	1225,5s
ambar	1272,5s
perola	1199,0s
Média	1238s

A medida de eficiência de um algoritmo paralelo é dada pela fórmula

$$E = (t_s/t_p)/n$$

que divide o ganho obtido pelo número de processadores utilizados. Na fórmula, t_s é o tempo seqüencial, t_p representa o tempo da execução paralela e n, o número de processadores utilizados.

Para paralelizar a rede CNM composta de 4 hipóteses, seriam necessárias 4 máquinas. Além dessas, foi utilizada uma máquina adicional para executar o servidor e o processo mestre de modo a não interferir na distribuição de carga. Isso resulta em um total de 5 máquinas para a execução paralela do algoritmo.

Substituindo-se, então, os valores das tabelas anteriores na fórmula, obtém-se uma eficiência de 79%, tanto para o algoritmo de aprendizado inicial quanto para o algoritmo de aprendizado incremental.

A pimeira vista, pode parecer que 79% é uma eficiência muito baixa para se atingir, contudo uma análise mais cuidadosa das tabelas de tempos mostra que a paralelização, propriamente dita, teve uma eficiência na ordem de 99%, em média, para os dois algoritmos. O que puxou a eficiência para baixo foi a inclusão de uma máquina adicional para a execução do servidor e do processo mestre. Entretanto essa máquina será utilizada por todas as redes neurais criadas a partir daquele servidor e seu uso será compartilhado por processos apenas de controle e que não realizam operações computacionalmente caras.

Ilustrando com números, suponha-se uma rede CNM com 10 hipóteses e a mesma eficiência na paralelização, *i. e.*, 99%. O acréscimo de uma máquina adicional não seria tão significativo, pois a eficiência cairia apenas para 90%, e quanto maior o número de hipóteses, menos representará a máquina adicional do servidor e do mestre. Apesar disso, o enfoque depende da topologia da rede para uma boa paralelização.

5.2 Resultados do *Back Propagation*

Foi escolhido um problema bem conhecido e largamente utilizado na literatura sobre *Back Propagation* para realizar os testes comparativos da implementação paralela *versus* a seqüencial. Esse problema visa fazer com que uma rede *back propagation* seja treinada para resolver a operação booleana ou-exclusivo (*XOR*). O problema possui quatro casos possíveis para as entradas e gera um único valor como saída.

A grande dependência de dados do algoritmo *back propagation* é evidenciada por seu processo iterativo, que necessita da finalização do laço anterior para resolver o seguinte. Devido a isso, sua implementação eficiente em um ambiente em que o custo de troca de mensagens é muito alto se torna tarefa difícil de realizar.

Na implementação realizada neste trabalho não foi possível atingir uma paralelização eficiente, apesar do uso de uma estratégia para minimizar a troca de mensagens, conforme visto na seção 4.3.5.

A seguir são apresentados os resultados obtidos com três redes selecionadas aleatoriamente, onde são fornecidos subsídios para compreender o problema. Sobre todas as redes foram realizados experimentos variando o número de processos escravos e o número de iteração de intervalo para comunicação entre os processos.

Os experimentos consistiram na execução do algoritmo de treinamento para a função ou-exclusivo até que fosse atingido um erro de, no máximo, 0.01. Foram utilizados como parâmetros, para efeito de reprodução dos resultados aqui obtidos: coeficiente de aprendizado = 0.8 e coeficiente de amortecimento = 0.9; 2 unidades de entrada, 1 de saída e 2 e uma camada intermediária. No apêndice B, são fornecidas as descrições detalhadas das três redes utilizadas.

O procedimento de teste consistiu na execução do algoritmo paralelo até que ele convergisse ou atingisse um certo número de iterações. Depois disso, a rede era gravada em um arquivo e, posteriormente, testada através do algoritmo seqüencial

para confirmar o erro da rede e eliminar a possibilidade de algum defeito na implementação paralela adulterar os resultados.

O primeiro teste consistiu na execução com dois processos escravos, cada um lendo uma diferente metade da base de casos, totalizando 2 casos por processo. Foi variado o número de iterações que os processos ficavam sem realizar a difusão dos dados de 1 até 10, e a figura 5.1 ilustra a evolução do erro ao longo do tempo, para os passos de 1 a 5, na Rede 1.

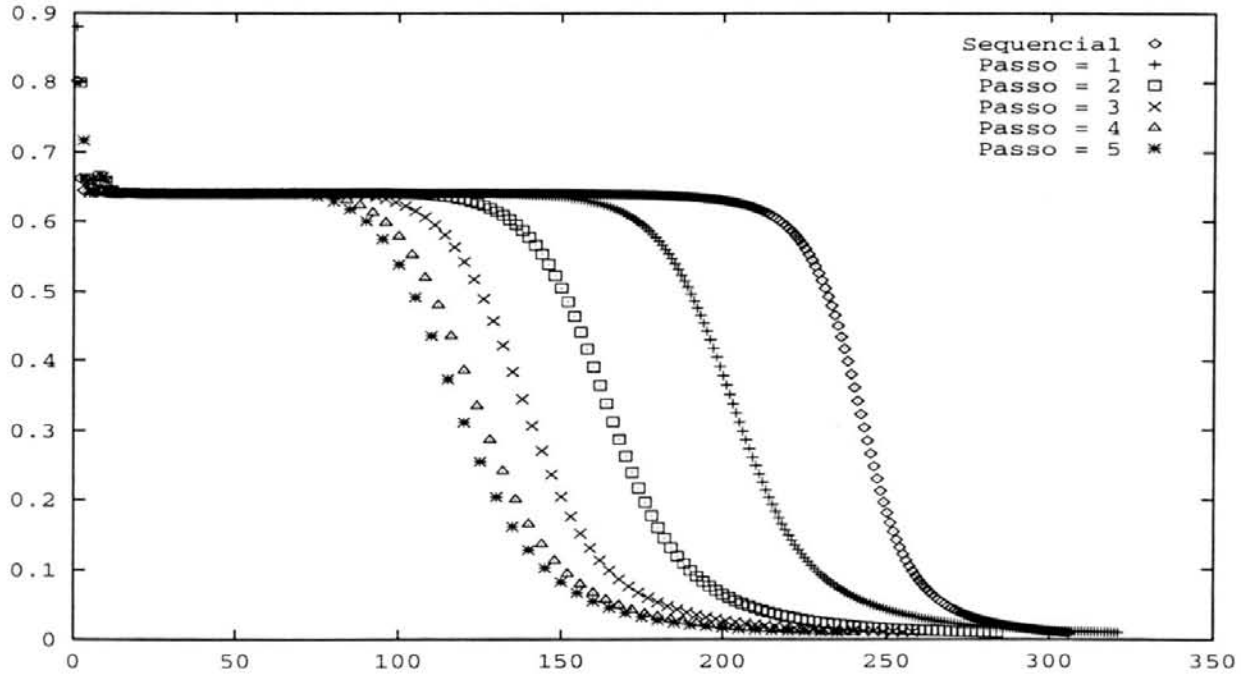

FIGURA 5.1 — Evolução do Erro com 2 Escravos

Sem considerar o aspecto tempo de execução é particularmente interessante observar um comportamento até certo ponto inesperado na evolução do erro. Com o aumento do número de passos, a rede tende a convergir em um menor número de iterações. Contudo, a partir de um certo valor, a convergência é prejudicada de forma significativa, conforme pode ser visto nas figuras B.1 B.3 B.9, que estão no anexo B.

Nos teste com quatro processos escravos, o resultado não foi o mesmo e as redes, em quase todos os testes, não convergiram até 2000 iterações, que foi estipulado como

valor máximo. A evolução do erro na Rede 1 pode ser vista na figura 5.2 e no anexo B.

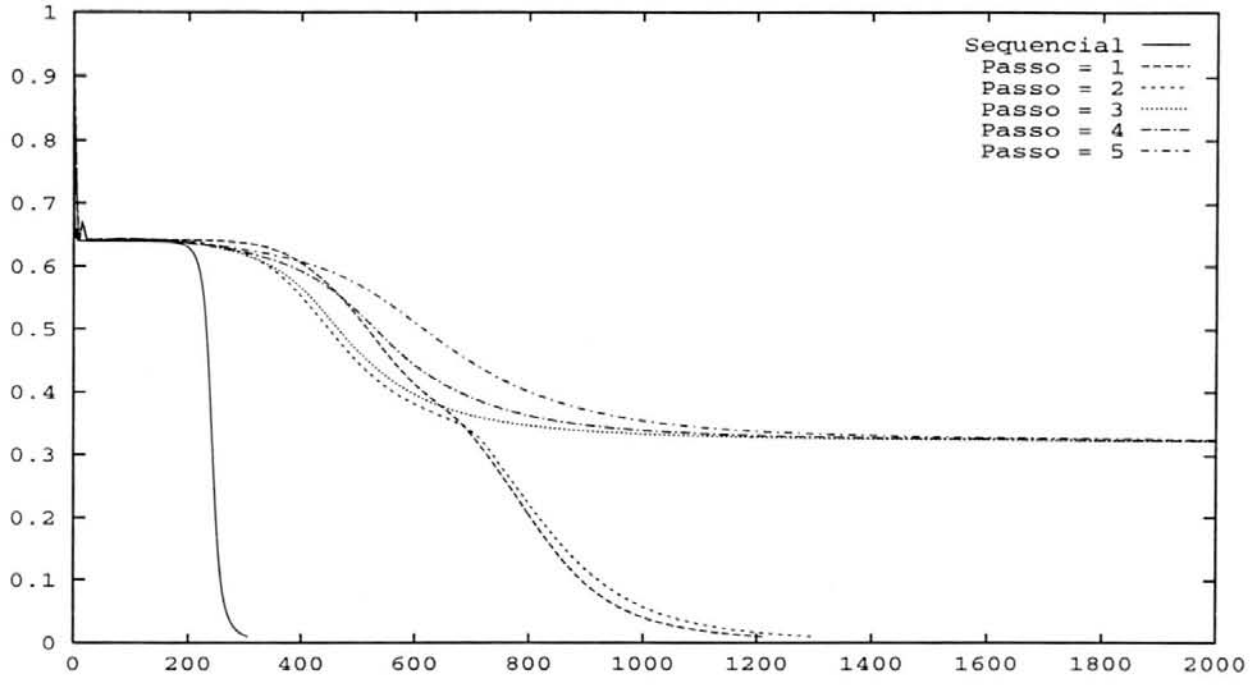

FIGURA 5.2 — Evolução do Erro com 4 Escravos

Nos testes com dois escravos, há duas redes semi-independentes treinando para reconhecer uma diferente metade da base de casos. Essas redes começam iguais e a cada passo de comunicação trocam informações sobre seus valores internos, visando gerar uma terceira rede formada da combinação das duas que reconheça todos os casos da base. No caso das Redes 1 (figura 5.1) e 3 (figura B.4) parece que a especialização das duas redes semi-independentes combinada com o grau certo de sinergia entre elas permite essa redução no número total de iterações. Contudo esses mesmos fatores, quando não adequados, podem gerar o que aconteceu com a Rede 2 (figura B.2), na qual houve aumento no número de iterações. Além disso é fácil perceber que, com a diminuição das comunicações a sinergia diminui e a rede não converge com a mesma velocidade.

Não obstante, com base nos resultados obtidos, pode se afirmar que a variação do número de passos de comunicação pode levar a rede a convergir em um menor número de iterações.

Cabe analisar, então, qual seria o valor ideal para esse passo de comunicações e, por isso, foram calculados os números médios de iterações necessárias para que as redes convergissem em cada um dos diferentes passos testados. Tais valores se encontram na tabela B.1 e uma representação gráfica pode ser vista na figura 5.3

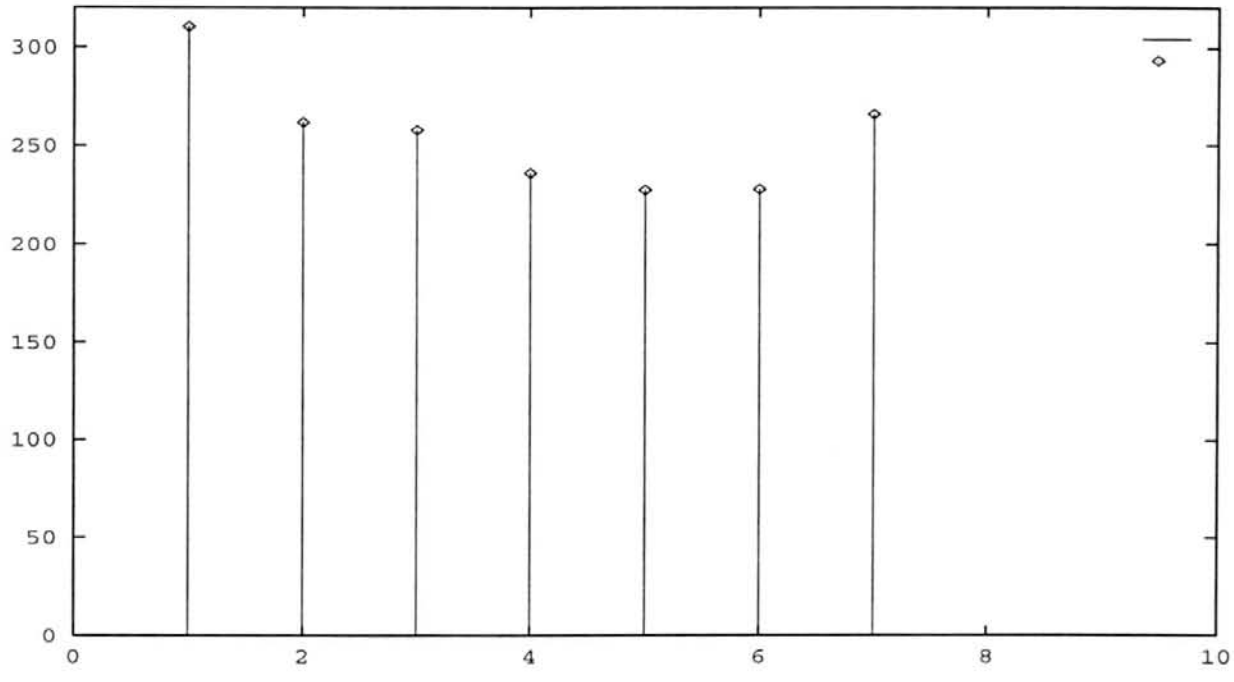

FIGURA 5.3 — Número Médio para Convergir com 2 Escravos

Entretanto tais valores não incluem uma informação importante, que é o número de redes que convergiram usando esse passo. Para uma representação gráfica desse valor, a média foi dividida pelo número de redes que convergiram e obteve-se o gráfico da figura 5.4, no qual as menores colunas indicam os melhores passos.

O menor número de iterações foi conseguido com o uso do passo igual a 5, mais ou menos 1[1], indicando que a escolha do passo de comunicação igual a 5 mostrou ser boa para o ou-exclusivo com dois escravos, contudo isso pode variar caso outros problemas sejam enfocados. De qualquer forma, pode ser usado como um ponto de referência para um valor de teste inicial.

[1]Observa-se a exclusão do passo igual a 1 nesta análise, devido a excessiva troca de mensagens.

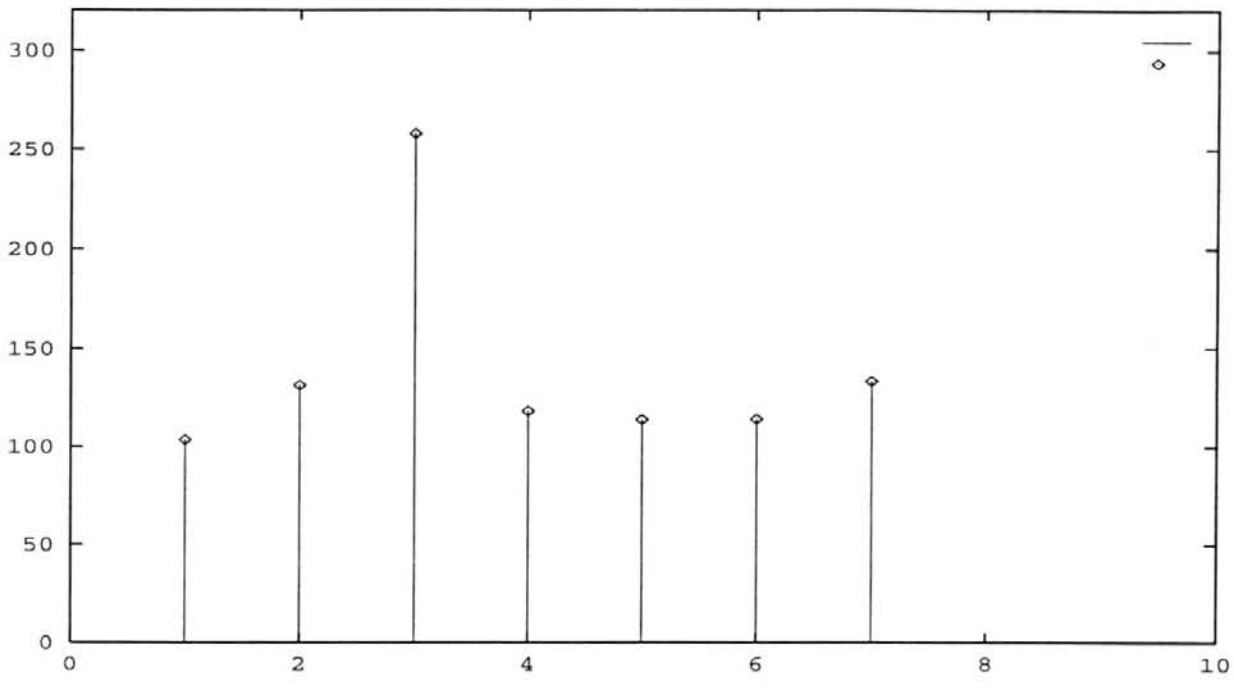

FIGURA 5.4 — Número Médio para Convergir

Como análise final tem-se que uma diminuição na troca de mensagens, diminuiu o número de iterações necessárias para convergir, mas, a partir de um certo ponto, essa diminuição prejudica muito a convergência.

Cabe ressaltar aqui que a análise e a interpretação da evolução do erro no algoritmo *back propagation* paralelo implementado neste trabalho não pretendem ser e não são conclusivas, pois apenas um problema foi analisado e isso reduz a base para a formulação de hipóteses. Ademais, uma análise detalhada está fora do escopo deste trabalho e, sendo assim, pretendeu-se apenas fornecer subsídios para um posterior trabalho que queira investigar as aplicações de tal característica do algoritmo.

Com relação ao tempo de execução da implementação paralela, o problema do ou-exclusivo não é adequado para testes, uma vez que o tempo de execução seqüencial é muito pequeno, cerca de 8 segundos nas máquinas utilizadas, e corresponde somente ao tempo de inicialização do algoritmo paralelo. Assim, utilizou-se

outro problema, visando treinar a a rede para gerar o bit de paridade para uma entrada composta de 4 bits.

A figura B.11 mostra a evolução do erro ao longo das iterações, para alguns valores do passo de difusão. Nota-se uma menor instabilidade no algoritmo paralelo, mas a convergência do seqüencial foi melhor, em número de iterações, nos casos testados.

Na tabela 5.5 estão os tempos do algoritmo seqüencial nas diferentes máquinas utilizadas.

TABELA 5.5 — Tempo do Treinamento Seqüencial *Back Propagation* para a Paridade de 4 Bits

Máquina	Tempo
marfim	155,5s
coral	163,0s
ambar	151,0s
perola	160,0s
Média	157,37s

O tempo de execução do algoritmo paralelo, com 4 escravos, foi de 180,35 segundos, superior ao tempo seqüencial, atingindo uma eficiência de apenas 17% e demonstrando que não é viável a implementação deste algoritmo em abiente distribuído. Não foi possível uma boa paralelização, variando o valor do passo de comunicação, que era o teste proposto a ser feito neste trabalho.

Para obter um melhor desempenho deve-se tentar modificar o algoritmo, no objetivo de diminuir a dependência de dados ou introduzir novas características que reforcem a tendência do algoritmo em convergir melhor que o seqüencial, em alguns casos. Uma estratégia, que não foi testada neste trabalho, pode ser a inicialização das redes dos escravos com valores aleatórios diferentes, de modo a tentar começar o algoritmo com uma abrangência maior, similarmente ao que é feito nos algoritmos genéticos, ressalvando todo o processo de seleção e escolha de indivíduos que é totalmente particular daquele tipo de algoritmo. Outra estratégia não testada foi a utilização de conjuntos não disjuntos de casos para os escravos para verificar se

isso provocaria alguma alteração na velocidade de convergência. Também pode-se verificar a influência da escolha aleatória de diferentes subconjuntos de casos.

Um melhoramento que pode ser feito a nível de implementação é manter uma cópia local da base de casos, de modo que a rede não seja sobrecarregada pelo acesso ao servidor de arquivos para ler a base de casos. Uma alternativa melhor, se a base for pequena, é o seu armazenamento em memória principal, eliminando o acesso ao disco e acelerando o processamento, tanto do algoritmo seqüencial como do paralelo.

6 Conclusões

Este trabalho apresentou uma proposta para um simulador de Redes Neurais Artificiais que visa atingir melhor desempenho através de Processamento Paralelo e Distribuído e, possivelmente, tornar interativa a configuração e experimentação de RNAs por parte de pesquisadores da área, mediante o uso de diversos computadores de alto desempenho. Por outro lado, a sua proposta modular visa permitir que pesquisadores da área de processamento de alto desempenho possam implementar novos modelos de RNAs de forma simples.

Dentro desse enfoque, foram implementados dois modelos de RNAs e seus desempenhos foram comparados com as versões sequênciais nas quais eles foram baseados.

As medidas de desempenho do simulador para o algoritmo *Combinatorial Neural Model* foram muito boas, atingindo uma eficiência de até 99% na paralelização. Isso foi possível devido a ausência de dependência de dados entre partes da rede. O enfoque adotado neste trabalho levou a uma implementação em que o ganho em tempo de processamento depende da topologia da rede, mas precisamente do número de hipóteses. Contudo conseguiu-se uma implementação em que o ganho é diretamente proporcional ao número de hipóteses e, exemplificando, para uma rede com apenas duas hipóteses o treinamento é realizado na metade do tempo.

No caso de *Back Propagation* os resultados não foram satisfatórios. A alta dependência de dados obriga uma troca excessiva de mensagens para uma rede

compartilhada de 10 Mbit/s e não propicia que se alcance aumento de desempenho. Uma estratégia para diminuir o número de mensagens foi analisada, mas não produziu resultados positivos. No entanto o algoritmo paralelo mostrou ser capaz de convergir em menos iterações do que o seqüencial e isso levou a uma análise um pouco mais detalhada dessa característica, mas uma avaliação em profundidade foi considerada fora do escopo deste trabalho e deixada para o futuro.

Em linhas gerais, a proposta do servidor aqui apresentada mostrou ser eficaz nos testes em que seu desempenho foi fator importante, como ficou evidenciado na paralelização do *Combinatorial Neural Model*. Para o caso de *Back Propagation* o modelo não teve relação com a baixa eficiência apresentada, basta ver que o seu modelo independente de arquitetura é talvez sua característica mais importante, pois permite o uso de qualquer computador conectado à rede local. Com isso pode-se implementar um algoritmo, como o *Back Propagation*, em um computador matricial dedicado que forneca a velocidade de comunicação necessária, sem que isso modifique a interface com o programa que utiliza o servidor.

No entanto alguns pontos podem ser melhorados, pois um servidor com controle centralizado, como o proposto neste trabalho, obviamente introduz um gargalo, que se manifestará em algum momento. Apesar de aplicações com controle centralizado poderem gerenciar um bom número de outros processos [SCH 92b] e o meio físico de troca de mensagens ser o principal fator limitante, é necessário prever alguma forma de minimizar esse gargalo quando o servidor estiver sobrecarregado.

Como alternativa propõe-se a utilização de um esquema de transferência de controle para outro servidor, criado sob demanda, das requisições dos clientes. Para minimizar o uso do meio físico pode-se guardar informações sobre a topologia da rede e executar o servidor o mais próximo possível da maioria dos processos clientes.

Para não modificar o paradigma cliente-servidor utilizado, que tem sua base em um processo reativo e diversos processos ativos, teria que ser incluído um campo de redirecionamento em todas as chamadas de serviço dos clientes.

Contudo deve-se avaliar o custo de transferência de controle das redes já criadas ou transferir apenas a criação de novas redes. Outro fator que pode vir a ser considerado é a transferência de processos muito pesados para outras estações quando usuários querem utilizá-las em modo interativo, no caso específico das estações de trabalho.

Embora as ferramentas utilizadas no desenvolvimento tenham se mostrado satisfatórias, deve-se considerar a possibilidade de uma reimplementação do simulador utilizando um melhor suporte para a construção de aplicações paralelas, como o PVM, de modo a facilitar sua expansão a manutenção.

Cabe ressaltar, no entanto, que o presente trabalho foi realizado dentro do enfoque de Processamento Paralelo/Distribuído de Alto Desempenho e teve também como objetivo possibilitar a capacitação técnica no desenvolvimento de aplicações desse tipo sem o uso de ferramentas de mais alto nível.

Acredita-se ter contribuído com a apresentação de uma proposta de um Simulador Distribuído para Redes Neurais Artificiais e as discussões sobre suas necessidades e características.

Espera-se que o presente trabalho estimule a continuidade e a implementação paralela de outros modelos em diferentes máquinas, bem como uma análise mais aprofundada de alguns resultados aqui obtidos, em especial do algoritmo paralelo de *Back Propagation*.

Anexo A Programação de um Cliente para o Simulador

A.0.1 Acesso aos Serviços

Esta seção descreve os serviços oferecidos pelo servidor e que são acessados através de RPCs. Para cada serviço é fornecido um exemplo de chamada em código fonte, na linguagem C e, para não ser excessivamente longo, apenas os serviços CNM estão descritos. As variáveis utilizadas nos exemplos são as seguintes:

- `cl`: variável do tipo CLIENT, utilizada em todas as chamadas para identificar o servidor que deve ser usado. Ela é estabelecida através de uma chamada à função `clnt_create` como no seguinte pedaço de código:

```
CLIENT cl;

cl = clnt_create( DNN_SERVER_HOST, SERVER_PROGRAM_NUMBER,
SERVER_VERSION_NUMBER, "udp" )
```

No exemplo `DNN_SERVER_HOST` contém o nome da máquina em que o servidor está executando e poderia ser, por exemplo, `"perola.inf.ufrgs.br"`; `SERVER_PROGRAM_NUMBER` e `SERVER_VERSION_NUMBER` estão definidos no arquivo `rpcregs.h` e não devem ser alterados; a palavra `"udp"` estabelece que as

chamadas de procedimentos remotos devem ser feitas utilizando o protocolo UDP (*User Datagram Protocol*) e deve ser usada se o servidor foi compilado com a opção `USE_UDP_RPC`, caso contrário deve-se usar `"tcp"`, que usa o protocolo TCP (*Transmition Control Protocol*).

- `dnn_pars`: variável do tipo `t_DNN_service_parameters` utilizada para passar os parâmetros dos serviços genéricos, *i. e.*, aqueles comuns a todos os modelos.

- `cnm_pars`: variável do tipo `t_CNM_service_parameters` que contém os parâmetros dos serviços específicos ao modelo CNM.

A.0.1.1 Serviços Gerais

- Encerra a execução de uma rede previamente criada, liberando todos os recursos alocados (processador, memória, etc). Como argumento é passado o identificador da rede, que é retornado pelo servico de criação.

```
dnn_pars.ident = Identificador_da_Rede_a_ser_Encerrada;
client_call( cl, SV_KILL_NET, xdr_t_DNN_service_parameters,
&dnn_pars, xdr_t_DNN_service_parameters, &dnn_pars, timeout );
```

- Salva a estrutura da rede neural em um arquivo de modo que seu estado possa ser congelado para uso futuro. Recebe o nome do arquivo em que a rede será colocada.

```
dnn_pars.ident = Identificador_da_Rede_a_ser_Salva;
strcpy( dnn_pars.files.dump_fname, "arquivo_salvamento" );
client_call( cl, SV_DUMP_NET, xdr_t_DNN_service_parameters,
&dnn_pars, xdr_t_DNN_service_parameters, &dnn_pars, timeout );
```

- Carrega os dados de uma rede previamente salva pelo serviço anterior. Recebe o nome do arquivo onde a rede está armazenada.

```
dnn_pars.ident = Identificador_da_Rede_a_ser_Carregada;
strcpy( dnn_pars.files.load_fname, "arquivo_salvamento" );
```

```
client_call( cl, SV_LOAD_NET,
xdr_t_DNN_service_parameters, &dnn_pars,
xdr_t_DNN_service_parameters, &dnn_pars, timeout );
```

A.0.1.2 Serviços das Redes CNM

Aqui são enumerados os serviços disponíveis no servidor para a manipulação de redes neurais CNM.

- Cria uma nova rede neural CNM no servidor, gerando a alocação dos recursos necessários à sua execução. Recebe o número de hipóteses, número de neurônios de entrada e a ordem da rede a ser criada. Retorna o identificador da rede, que será usado nos acessos posteriores em `cnm_pars.ident`.

```
cnm_pars.order = Ordem_da_Rede_a_ser_Criada;
cnm_pars.n_hypothesis = Numero_de_Hipoteses;
cnm_pars.n_input_neurons = Numero_Unidades_Entrada;
client_call( cl, SV_CREATE_CNM_NET,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Dispara o processo de aprendizado inicial (starter learning) na rede especificada. Recebe os limiares de aceitação e poda (acceptance and prunning thresholds), o número de casos de teste e o nome do arquivo que os contém.

```
cnm_pars.ident = Identificador_da_Rede;
cnm_pars.acc_threshold = Limiar_de_Aceitacao;
cnm_pars.prun_threshold = Limiar_de_Poda;
cnm_pars.n_cases = Numero_Casos;
strcpy( cnm_pars.files.cases_fname, "arquivo_de_casos" );
client_call( cl, SV_CNM_INCR_LEARNING,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Executa o algoritmo da rede para uma série de valores de entrada e retorna o estado das hipóteses. Após o aprendizado, este serviço é utilizado para testar se a rede foi treinada adequadamente e também para uso normal da rede. Para efeito de implementação este serviço foi dividido em dois: um deles recebe um arquivo como entrada e outro um vetor de valores. O arquivo deve ser utilizado para grandes volumes de dados na entrada e o vetor para pequena quantidade de dados, sendo que seu limite é definido na compilação do servidor. O exemplo a seguir utiliza o serviço baseado em arquivo:

```
cnm_pars.ident = Identificador_da_Rede;
strcpy( cnm_pars.files.cases_fname, "arq_entrada" );
strcpy( cnm_pars.files.dump_fname, "arq_saida" );
client_call( cl, SV_CNM_TEST_NET_FILE,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Dispara o processo de aprendizado incremental (*incremental learning*) de uma rede CNM. Recebe o número de refinamentos, o número de casos e o nome do arquivo de casos.

```
cnm_pars.ident = Identificador_da_Rede;
cnm_pars.n_refinements = Numero_de_Refinamentos;
cnm_pars.n_cases = Numero_de_Casos;
strcpy( cnm_pars.files.cases_fname, "arq_casos" );
client_call( cl, SV_CNM_INCR_LEARNING,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Estabelece o valor do limiar de poda (*prunning threshold*), permitindo variá-lo entre experimentos.

```
cnm_pars.ident = Identificador_da_Rede;
cnm_pars.prun_threshold = Limiar_de_Poda;
client_call( cl, SV_CNM_SET_PRUN_THRESHOLD,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Estabelece o valor do limiar de aceitação (*acceptance threshold*), com o mesmo objetivo.

```
cnm_pars.ident = Identificador_da_Rede;
cnm_pars.acc_threshold = Limiar_de_Aceitacao;
client_call( cl, SV_CNM_SET_ACC_THRESHOLD,
xdr_t_CNM_service_parameters, &cnm_pars,
xdr_t_CNM_service_parameters, &cnm_pars, timeout );
```

- Pega o estado de uma rede neural previamente criada. Recebe o identificador da rede.

```
dnn_pars.ident = Identificador_da_Rede;
client_call( cl, SV_CNM_GET_STATUS,
xdr_t_DNN_service_parameters, &dnn_pars,
xdr_t_DNN_service_parameters, &dnn_pars, timeout );
```

Anexo B Dados, Tabelas e Gráficos

B.1 Redes *Back Propagation* Utilizadas para o Ou-Exclusivo

Abaixo estão descritas as redes utilizadas nos experimentos com o modelo *back propagation* com.

Os seguintes dados são comuns a todas elas:

Entradas = 2

Saídas = 1

Camadas Intermediárias = 1

Neurônios na Camada Intermediária = 2

B.1.1 Rede 1

Bias = 0.071675 0.158185 0.457888

Pesos = 0.427875 0.516809 0.256462 0.296940 0.419793 0.830362

B.1.2 Rede 2

Bias = 0.072552 0.239361 0.424753

Pesos = 0.300868 0.811924 0.497438 0.817025 0.240905 0.754982

B.1.3 Rede 3

Bias = 0.683367 0.195635 0.500681

Pesos = 0.740707 0.015450 0.193116 0.016511 0.801884 0.013399

B.2 Rede *Back Propagation* Utilizada para a Paridade de 4 Bits

Entradas = 4

Saídas = 1

Camadas Intermediárias = 1

Neurônios na Camada Intermediária = 4

Bias = 0.511234 0.032109 0.903712 0.168689 0.975682

Pesos = 0.290019 0.004152 0.077354 0.893860 0.173170 0.111560 0.589775 0.250074 0.766331 0.713033 0.290795 0.509333 0.779288 0.974206 0.208570 0.268206 0.620304 0.417549 0.535739 0.430721

B.3 Gráficos de Evolução do Erro nas Redes *Back Propagation*

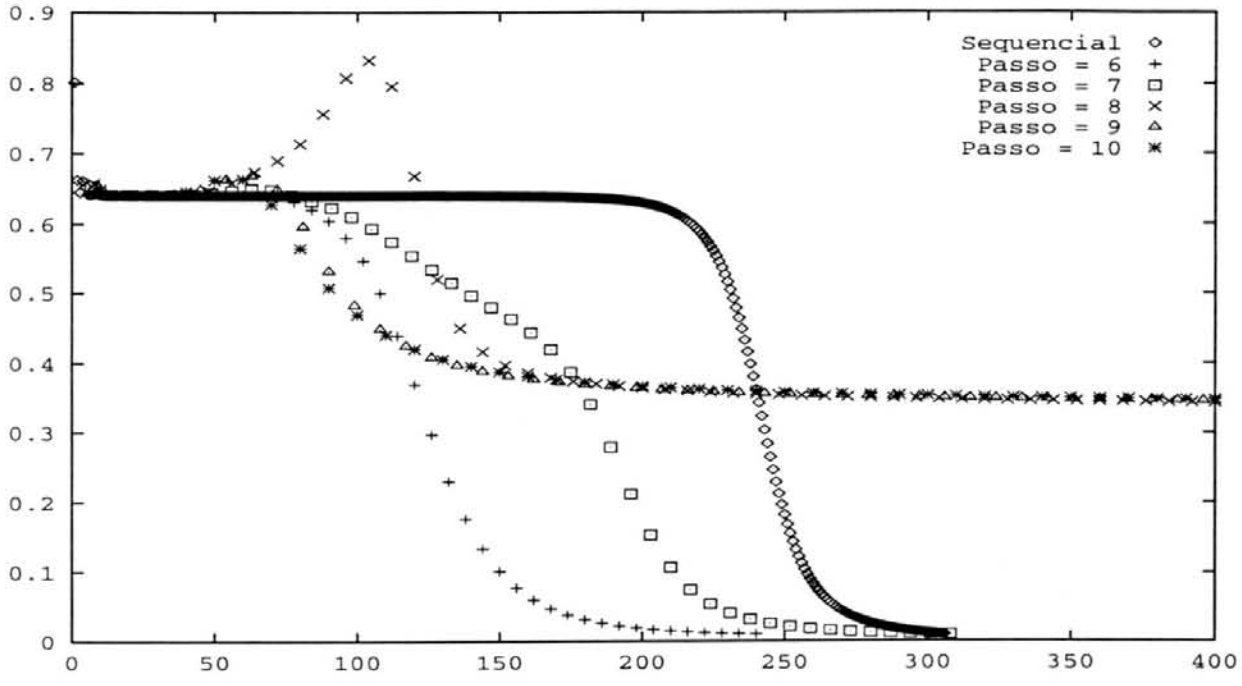

FIGURA B.1 — Erro com 2 Escravos e Passo=[6,10] (Rede 1)

TABELA B.1 — Número Médio de Iterações para Convergir nas 3 Redes com 2 Escravos

Intervalo de Comunicação	Iterações	Redes
1	310.33	3
2	262.00	2
3	258.00	1
4	236.00	2
5	227.50	2
6	228.00	2
7	266.00	2

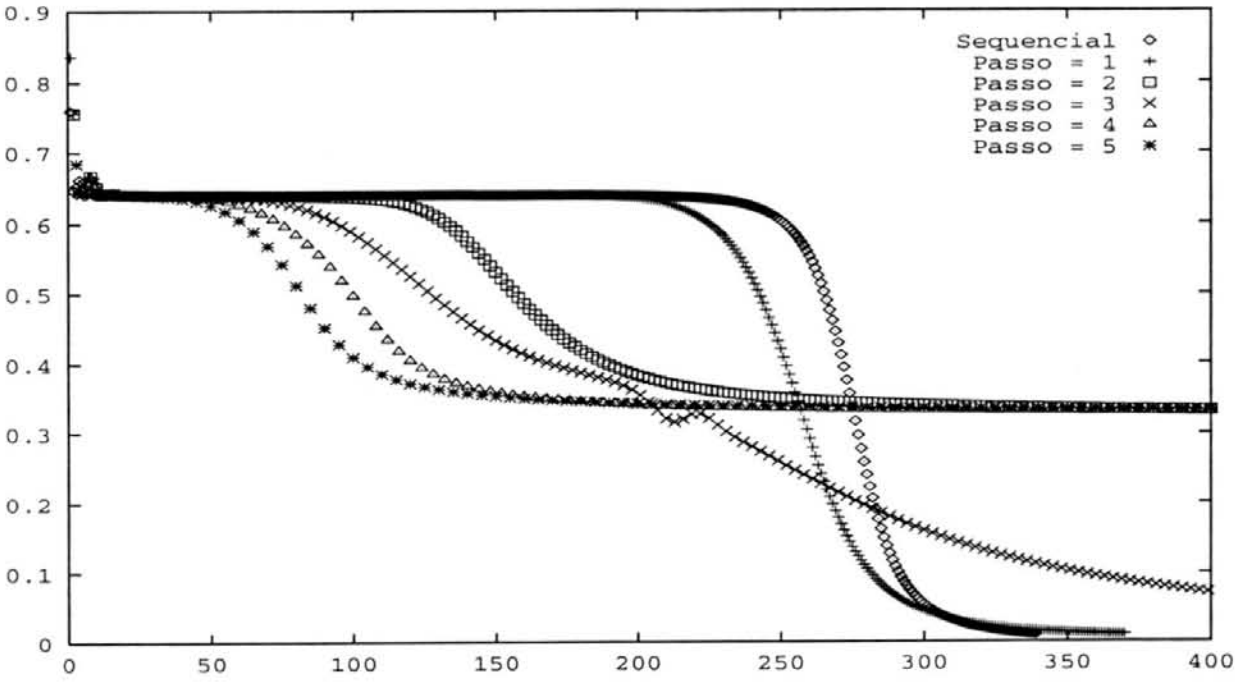

FIGURA B.2 — Erro com 2 Escravos e Passo=[1,5] (Rede 2)

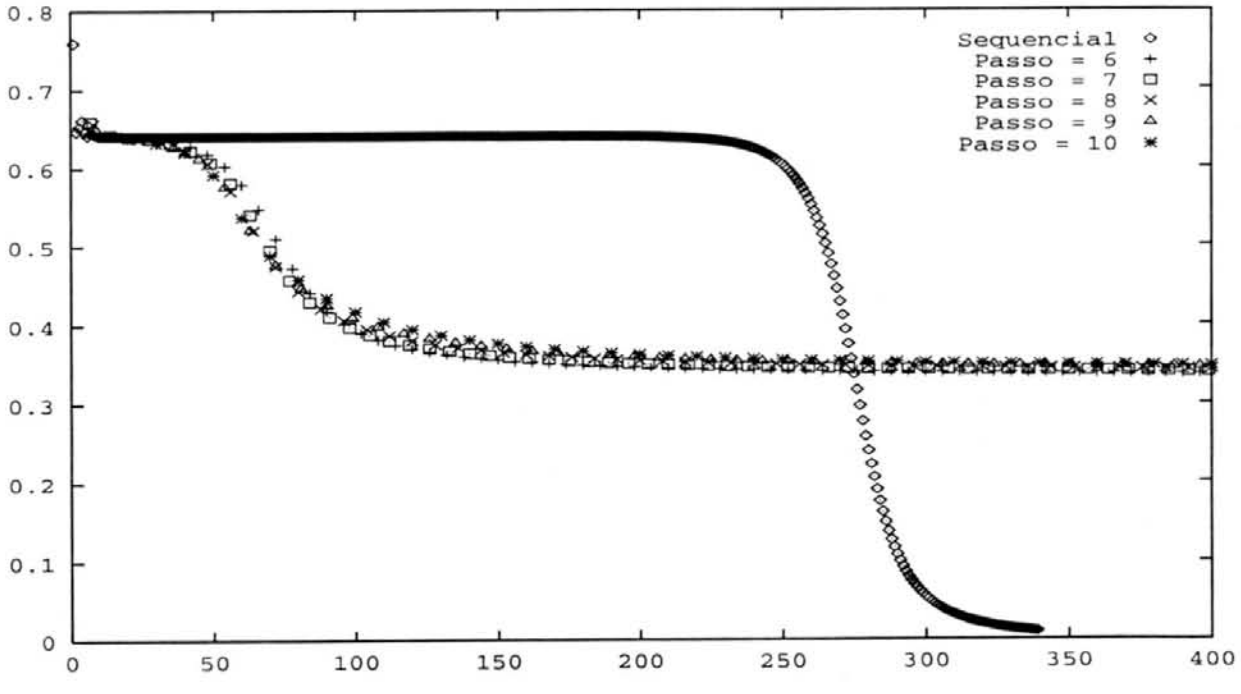

FIGURA B.3 — Erro com 2 Escravos e Passo=[6,10] (Rede 2)

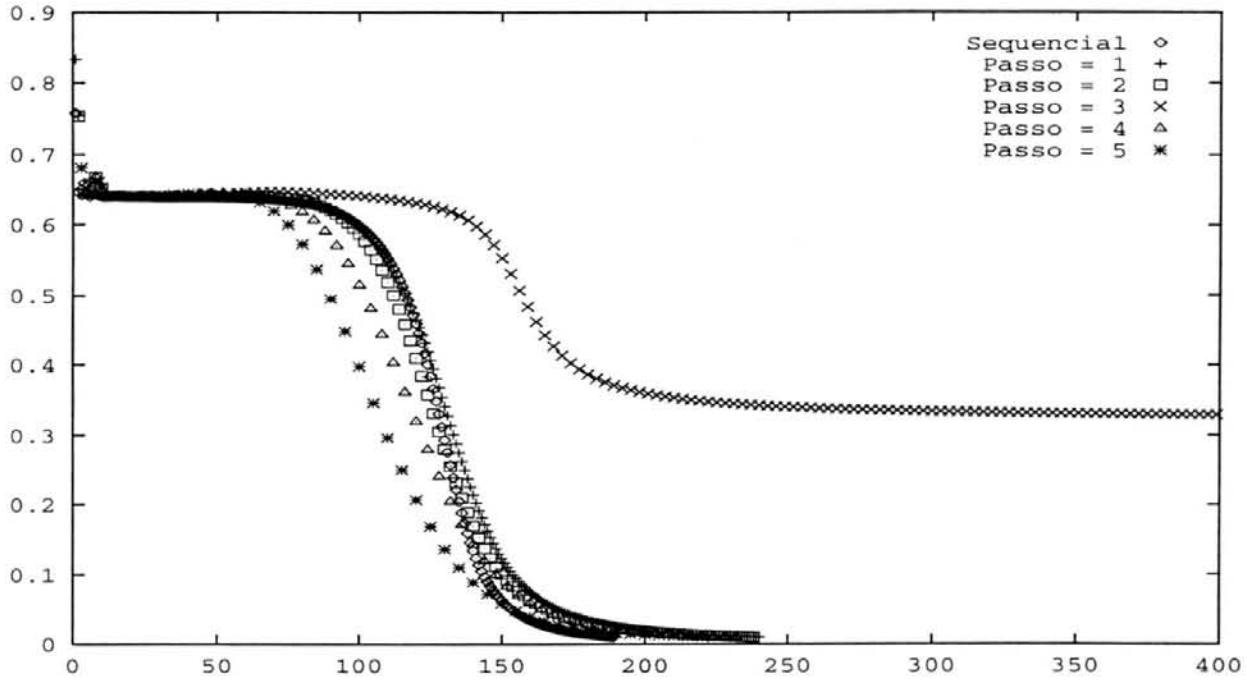

FIGURA B.4 — Erro com 2 Escravos e Passo=[1,5] (Rede 3)

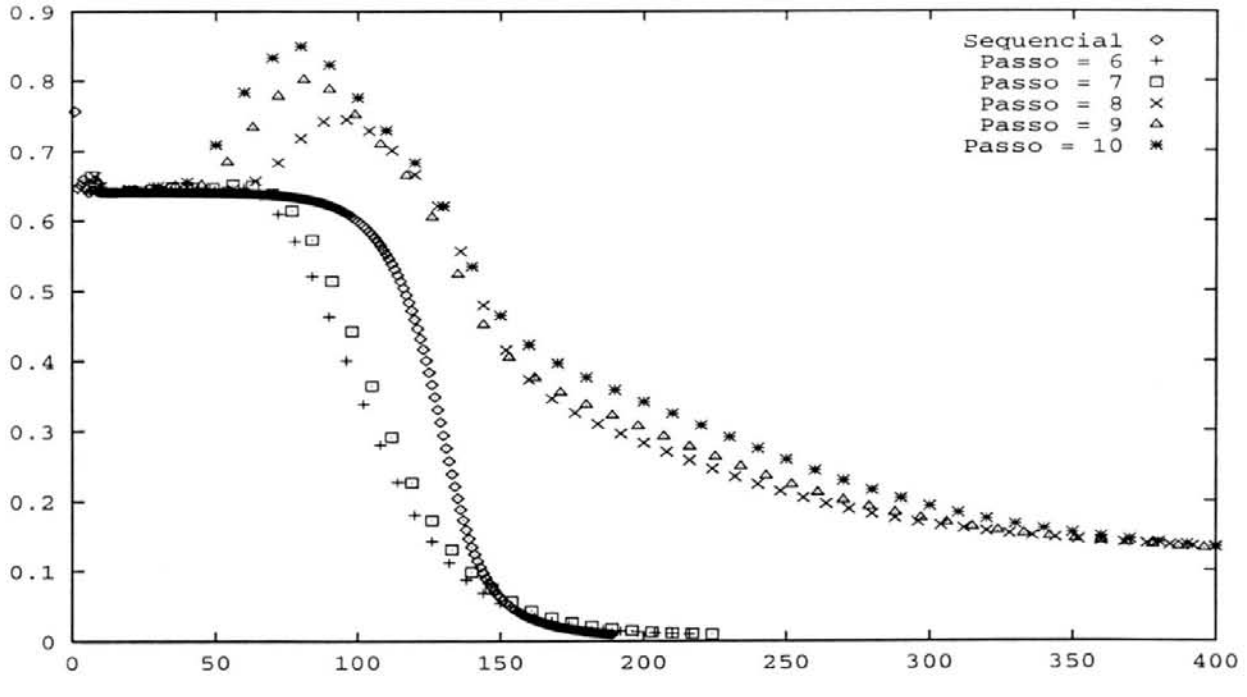

FIGURA B.5 — Erro com 2 Escravos e Passo=[6,10] (Rede 3)

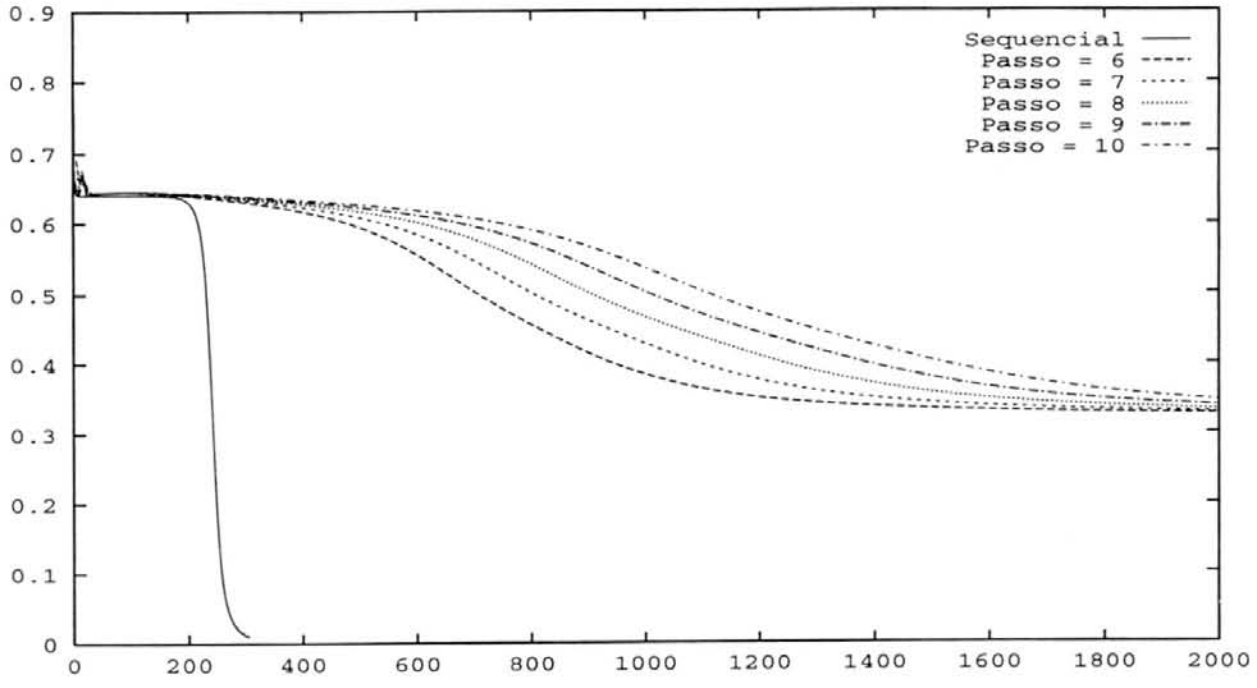

FIGURA B.6 — Erro com 4 Escravos e Passo=[6,10] (Rede 1)

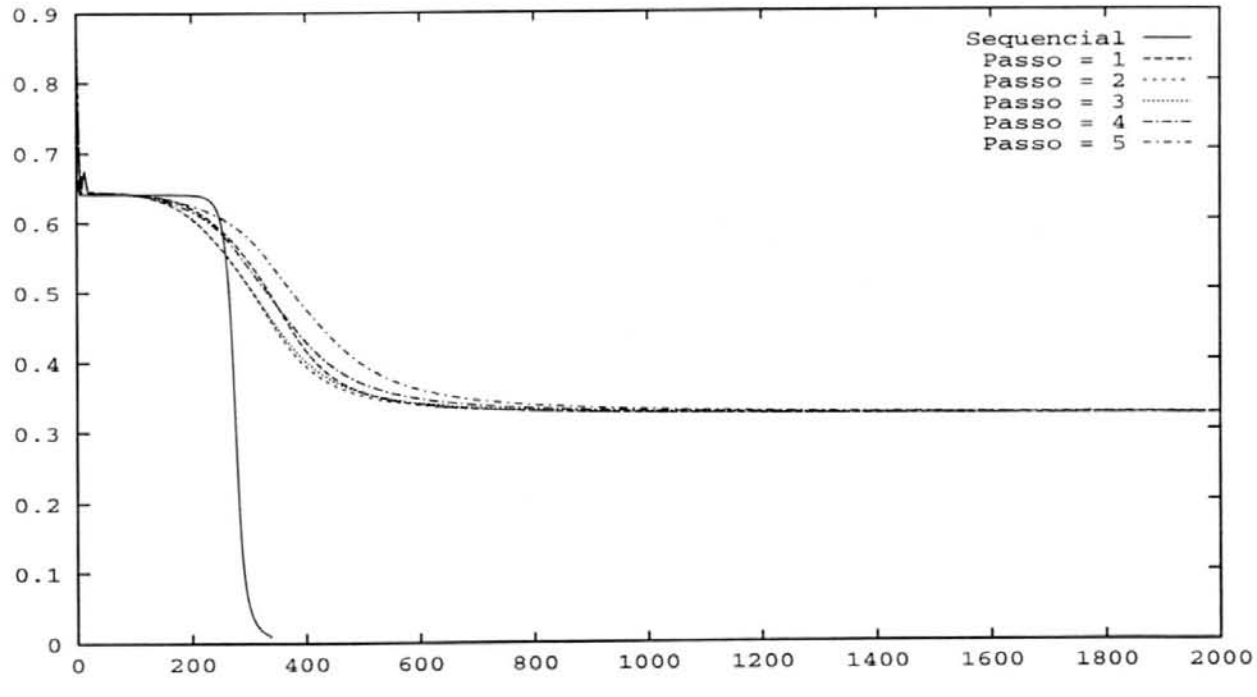

FIGURA B.7 — Erro com 4 Escravos e Passo=[1,5] (Rede 2)

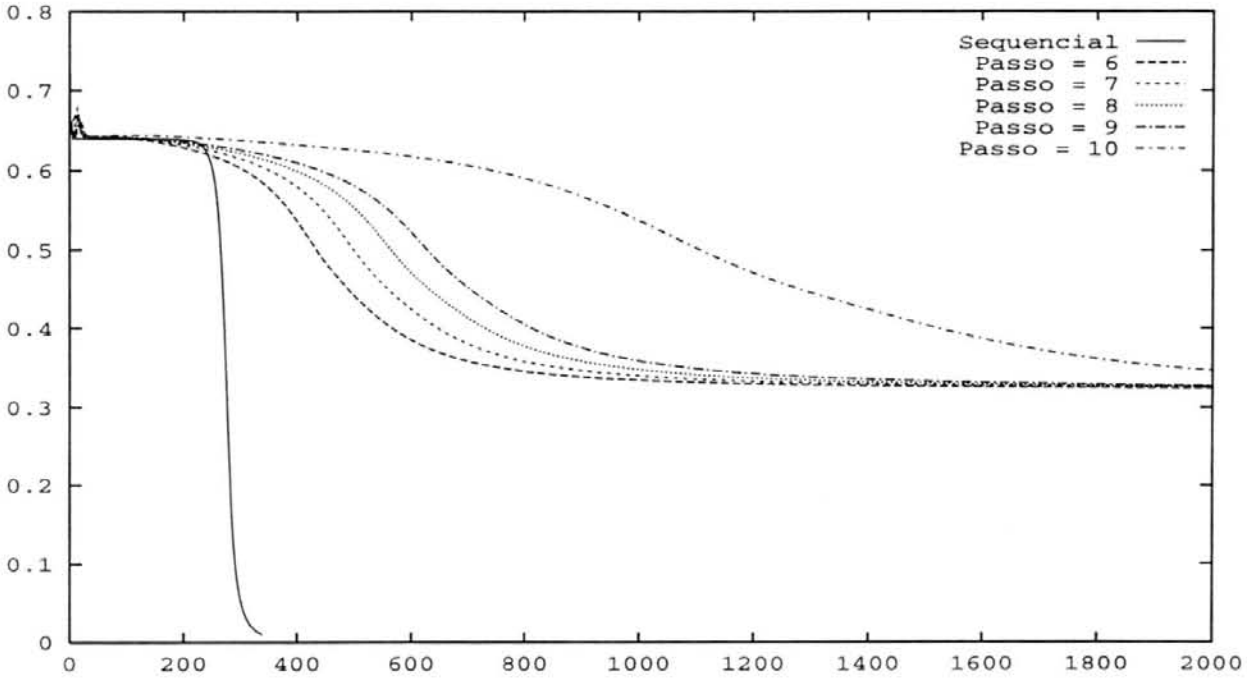

FIGURA B.8 — Erro com 4 Escravos e Passo=[6,10] (Rede 2)

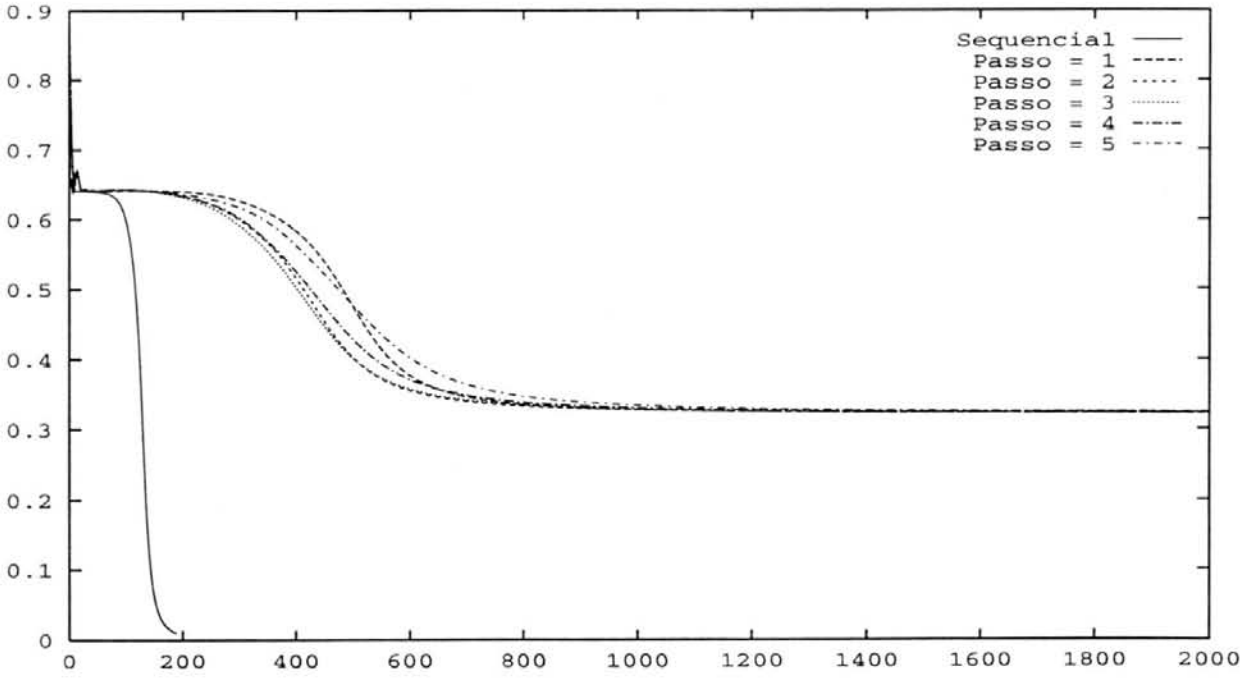

FIGURA B.9 — Erro com 4 Escravos e Passo=[1,5] (Rede 3)

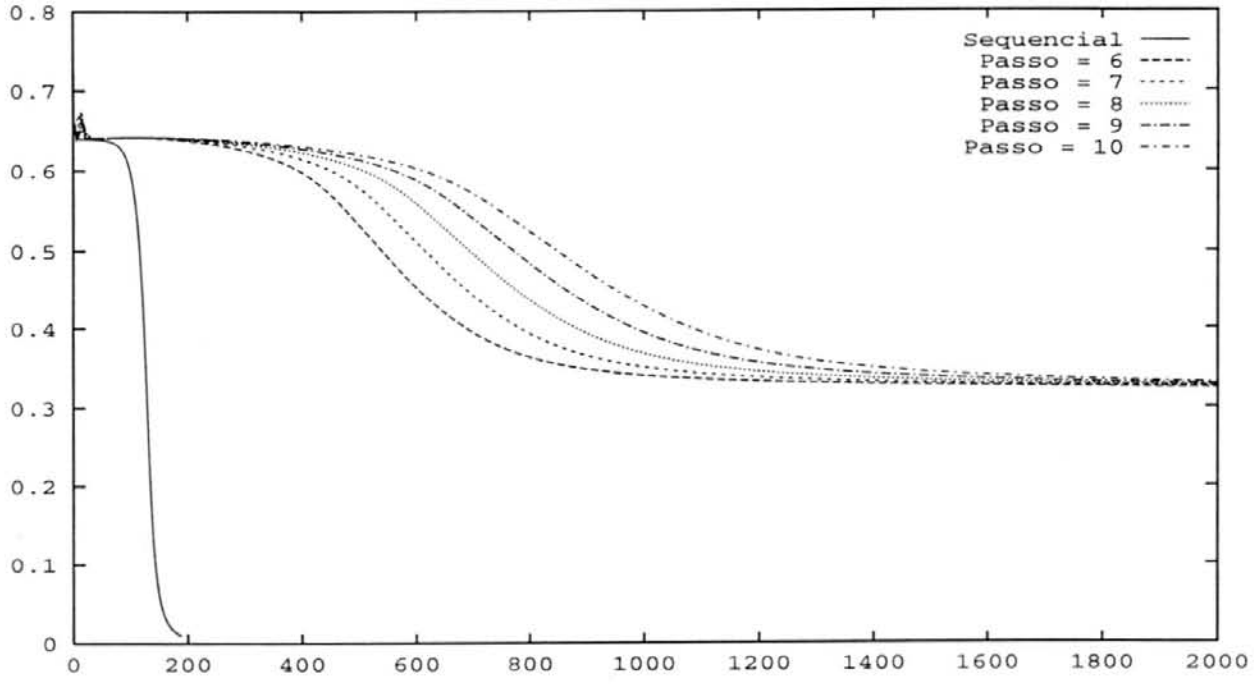

FIGURA B.10 — Erro com 4 Escravos e Passo=[6,10] (Rede 3)

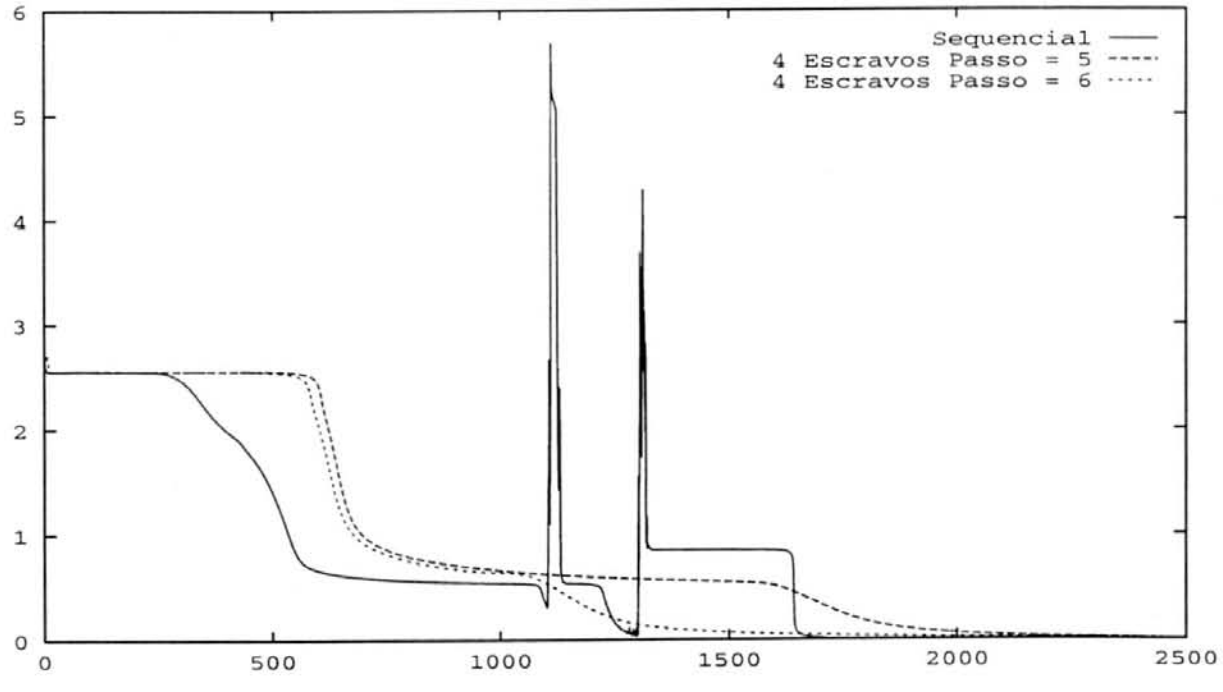

FIGURA B.11 — Erro para o Problema da Paridade

Bibliografia

[AKL 89] AKL, S. G. **The Design and Analysis of Parallel Algorithms.** New Jersey:Prentice-Hall, 1989. 401p.

[ASA 94] ASANOVIĆ, K. *et al.* A Supercomputer for Neural Computation. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1994, Orlando. **Proceedings...** Discataway:IEEE, 1994. v.1, p.5-9.

[BAR 90] BARBOSA, V. C.; LIMA, P. M. V. On the Distributed Parallel Simulation of Hopfield's Neural Networks. **Software Practice and Experience**, London, v.20, n.10, p.967-983, Oct. 1990.

[CAL 92] CALDAS, W. S. *et al.* SPD: Um Núcleo de Programação Distribuída em Redes de Computadores. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 4., 1992, São Paulo. **Anais...** São Paulo:EPUSP, 1992. p.445-457.

[COA 92] COAD, P.; YOURDON; E. **Análise Baseada em Objetos.** Rio de Janeiro:Campus, 1992. 225p.

[DUN 90] DUNCAN, R. A Survey of Parallel Computer Architectures. **IEEE Computer**, New York, v.23, n.2, p.5-16, Feb.1990.

[FER 92] FERNANDES, E. S. T.; SANTOS, A. D. **Arquiteturas Super Escalares**: Detecção e Exploração do Paralelismo de Baixo Nível. Porto Alegre:II-UFRGS, 1992. 135p.

[FER 94] FERNANDES, L. G. L.; NAVAUX, L. P. A. Um Estudo Preditivo das Redes Neurais Artificiais Comparado a Métodos Econométricos Tradicionais. In: SIMPÓSIO BRASILEIRO DE REDES NEURAIS, 1., 1994, Caxambu, MG. **Anais...** Porto Alegre:Pallotti, 1994. p.139-144.

[GAL 83] GALIL, Z.; PAUL, W. J. An Efficient General Purpose Parallel Computer. **Journal of the ACM**, New York, v.2, n.30, p.360-387, Apr.1983.

[GAU 88] GAUDET, S. *et al.* Multiprocessor Experiments for High-Speed *Ray-Tracing.* **ACM Transactions on Graphics**, New York, v.7, n.3, p.151-179, July 1988.

[GUA 92] GUAZZELLI, A. **Fundamentação de Modelos de Redes Neurais e seus Métodos de Aplicação no Reconhecimento de Caracteres.** Porto Alegre:II-UFRGS, 1992. 118p. Projeto de Diplomação.

[GUA 94] GUAZZELLI, A. Aprendizagem em Sistemas Híbridos. Porto Alegre:CPGCC da UFRGS, 1994. 131p. Dissertação de Mestrado.

[GUS 88] GUSTAFSON, J. L. Reevaluating Amdahl's Law. **Communications of the ACM**, New York, v.31, n.5, p.532-533, May 1988.

[HWA 84] HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing.** New York:McGraw-Hill. 1984. 846p.

[INM 88] INMOS LIMITED. **Products Catalog**. Bristol:INMOS Limited, 1988. 44p.

[INM 89] INMOS LIMITED. **3L Parallel C - User Guide**. Bristol:INMOS Limited, 1989. 261p.

[KER 78] KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. Englewood Cliffs:Prentice-Hall, 1978. 228p.

[LIU 94] LIU, X.; WILCOX, G. Benchmarking of the CM-5 and the Cray Machines with a Very Large Backpropagation Neural Network. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1994, Orlando. **Proceedings...** Discataway:IEEE, 1994. v.1, p.22-27.

[LOU 92] LOURES, E. F.; JUNIOR, G. E. P. S.; ALMEIDA, V. Análise de Desempenho de Programas Paralelos em Redes de *Workstations*. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 4., 1992. **Anais...** São Paulo:EPUSP, 1992. p.365-377.

[MAC 89] MACHADO, R. J.; ROCHA, A. F. **Handling Knowledge in High Order Neural Networks**: The Combinatorial Neural Model. Rio de Janeiro:IBM RSC, 1989. 22p. (Technical Report CCR076).

[MEA 94] MEANS, R. W. High Speed Parallel Hardware Performance Issues for Neural Network Applications. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1994, Orlando. **Proceedings...** Discataway:IEEE, 1994. v.1, p.10-16.

[MOR 92] MOREIRA, J. E. **Parallel Processing: Architecture and Programming**. São Paulo:EPUSP, 1992. 104p. Apostila do curso ministrado na Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, 2., 1992, São Paulo.

[MUL 94] MÜLLER, U. A. A High Performance Neural Net Simulation Environment. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1994, Orlando. **Proceedings...** Discataway:IEEE, 1994. v.1, p.1-4.

[PÉT 94] PÉTROWSKI, A. Choosing among several parallel implementations of the back propagation algorithm. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1994, Orlando. **Proceedings...** Discataway:IEEE, 1994. v.3, p.1981-1986.

[QUI 87] QUINN, M. J. **Designing Efficient Algorithms for Parallel Computers**. New York:McGraw-Hill, 1987. 288p.

[REN 94] RENAUD, P. E. **Introdução aos Sistemas Cliente/Servidor**. Rio de Janeiro:Infobook, 1994. 335p.

[RUM 86] RUMELHART, D. E.; McCLELLAND, J. L. **Parallel and Distributed Processing**. Cambridge:MIT Press, 1986. v.1.

[SCH 92a] SCHWINGEL, D. **APART^2 — Uma Arquitetura Paralela Assíncrona para** *Ray-tracing* **em** *Transputers*. Porto Alegre:II-UFRGS, 1992. 68p. Projeto de Diplomação.

[SCH 92b] SCHWINGEL, D.; BARONE, D. A. C. Exploração de Paralelismo de Dados em Ambiente Distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 4., 1992, São Paulo. **Anais...** São Paulo:EPUSP, 1992. p.459-470.

[SCH 93a] SCHWINGEL, D. **Uma Biblioteca para Distribuição de Aplicações**: Trabalho Individual. Porto Alegre:CPGCC da UFRGS, 1993. (TI-321).

[SCH 93b] SCHWINGEL, D.; SILVA, R. L.; BARONE, D. A. C. Um Sintetizador Paralelo de Imagens para ANIMAKER. In: SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO GRÁFICA E PROCESSAMENTO DE IMAGENS, 6., 1993, Recife. **Anais...** Recife:DI-UFPE, 1993.

[SEI 85] SEITZ, C. L. The Cosmic Cube. **Communications of the ACM**, New York, v.28, n.1, p.23-33, Jan. 1985.

[SEI 88] SEITZ, C. *et al.* **VLSI and Parallel Computation**. San Mateo:Morgan Kaufmann, 1988. 470p.

[STE 88] STEIN, R. M. T800 and Counting. **BYTE**, Hightstown, v.13, n.12, p.287-296, Nov.1988.

[SUN 90a] SUN MICROSYSTEMS. **Network Programming Guide**. Mountain View:Sun Microsystems, 1990.

[SUN 90b] SUN MICROSYSTEMS. **Programming Utilities and Libraries**. Mountain View:Sun Microsystems, 1990.

[SUN 90] SUNDERAN, V. S. PVM: A Framework for Parallel Distributed Computing. **Concurrency: Practice and Experience**, [S.l.], v.2, n.4, p.315-339, Dec.1990.

[TAK 90] TAKAHASHI, T.; LIESENBERG, H. K. E.; XAVIER, D. T. **Programação Orientada a Objetos**: uma visão integrada do paradigma de objetos. São Paulo:IME-USP, 1990. 335p.

[THI 86] THIS CPU does Floating Point Faster than Any Two-Chip Set. **Electronics**, [S.l.], p.51-55, Nov.1986.

[WIL 94] WILSON, G. **Timeline of Parallel Processing**. [S.l.:s.n.], 1994. Folheto.

Informática  *UFRGS*

Um Simulador Distribuído para Redes Neurais Artificiais.

por

Dinamérico Schwingel

Dissertação apresentada aos Senhores:

Prof. Dr. Cláudio Fernando Resin Geyer

Prof. Dr. Philippe Olivier Alexandre Navaux

Prof. Dr. Siang Wun Song (USP)

Vista e permitida a impressão.
Porto Alegre, 16/03/98.

Prof. Dr. Dante Augusto Couto Barone,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Curso de Pós Graduação
em Ciência da Computação
Instituto de Informática - UFRGS