UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Implementação e Avaliação de Máquinas de Comitê em um Ambiente com Múltiplos Processadores Embarcados em um Único Chip

Danniel Cavalcante Lopes

Natal, 30 de Julho de 2009.

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Lopes, Danniel Cavalcante.
Implementação e avaliação de máquinas de comitê em um ambiente com múltiplos processadores embarcados em um único chip / Danniel Cavalcante chip. – Natal, RN, 2009.
107 f.

Orientador: Jorge Dantas de Melo.
Co-orientador: Adrião Duarte Dória Neto.

Tese (Doutorado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e Computação.

1. Processamento paralelo – Tese. 2. Sistemas embarcados – Tese. 3. Redes neurais artificiais – Tese. I. Melo, Jorge Dantas de. II. Dória Neto, Adrião Duarte. III. Universidade Federal do Rio Grande do Norte. IV. Título.

RN/UF/BCZM CDU 004.272.2(043.2)

Implementação e Avaliação de Máquinas de Comitê em um Ambiente com Múltiplos Processadores Embarcados em um Único Chip.

Danniel Cavalcante Lopes

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do titulo de Doutor em Ciências.

Professor Orientador:
Dr. Jorge Dantas de Melo

Professor Co-orientador:
Dr. Adrião Duarte Dória Neto

Natal, 30 de Julho de 2009.

Implementação e Avaliação de Máquinas de Comitê em um Ambiente com Múltiplos Processadores Embarcados em um Único Chip.

Danniel Cavalcante Lopes

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN, como parte dos requisitos para a obtenção do grau de Doutor em Engenharia Elétrica e Computação.

Defendida e Aprovada em 30 de Julho de 2009.

Prof. Dr Jorge Dantas de Melo (Orientador)	Prof. Dr. Adrião Duate de Dória Neto (Co-orientador)
Prof. Dr. Manoel Eusebio Lima (Examinador Externo – UFPE)	Prof. Dr. Pedro Fernandes Ribeiro Neto (Examinador Externo – UERN)
Profa. Dra. Ana Maria Guimarães Guerreiro (Examinador Interno)	Prof. Dr. José Alberto Nicolau de Oliveira (Examinador Interno)

NATAL, RN

Dedico este trabalho
aos meus pais.

Agradecimentos

Aos meus pais, Raimundo Nonato de Paiva Lopes e Ana Maria Cavalcante Lopes, pela educação, valores e apoio dado durante toda essa trajetória.

Aos meus orientadores, Jorge Dantas de Melo e Adrião Duarte Dória Neto, pela dedicação, compromisso, ensinamento e amizade ao longo desses anos.

A todos os meus colegas do Laboratório de Sistemas Inteligentes (LSI) do DCA/UFRN por todas as brincadeiras, conversas, sugestões, aprendizado e apoio durante toda a nossa convivência diária.

Aos colegas do LSI, Rafael Marrocos Magalhães e Naiyan Hari Cândido Lima, que contribuíram diretamente nesse trabalho.

A todos os funcionários e professores do Departamento de Engenharia de Computação e Automação do Programa de Pós-Graduação em Engenharia Elétrica e Computação, que contribuíram com esse trabalho.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES, pelo suporte financeiro.

Resumo

O número de aplicações baseadas em sistemas embarcados cresce significativamente a cada ano. Isso se deve ao fato de que, apesar de sistemas embarcados possuírem restrições e unidades de processamento simples, o desempenho desses tem melhorado a cada dia. Entretanto a complexidade das aplicações também cresce, fazendo com que sempre exista a necessidade de um desempenho melhor. Portanto, apesar dessa evolução, existem casos, nos quais, um sistema embarcado com uma única unidade de processamento não é suficiente para realizar o processamento das informações em tempo hábil. Para melhorar o desempenho destes sistemas, pode-se analisar a implementação de soluções com processamento paralelo e assim utilizar-los em aplicações mais complexas que exigem um alto desempenho. A idéia é avançar além das aplicações que já utilizam sistemas embarcados, explorando a utilização de um conjunto de unidades de processamento cooperando entre si para execução de um algoritmo inteligente. O número de trabalhos existentes nas áreas de processamento paralelo, sistemas inteligentes e sistemas embarcados é grande. Entretanto, trabalhos que unam essas três áreas para a solução de algum tipo de problema são reduzidos. Diante deste contexto, esse trabalho teve como objetivo utilizar ferramentas disponíveis para arquiteturas FPGA, desenvolvendo uma plataforma com múltiplos processadores para utilização em problemas de processamento inteligente com redes neurais artificiais.

Palavras-Chave: Processamento Paralelo, Sistemas Embarcados, Redes Neurais Artificiais.

Abstract

The number of applications based on embedded systems grows significantly every year, even with the fact that embedded systems have restrictions, and simple processing units, the performance of these has improved every day. However the complexity of applications also increase, a better performance will always be necessary. So even such advances, there are cases, which an embedded system with a single unit of processing is not sufficient to achieve the information processing in real time. To improve the performance of these systems, an implementation with parallel processing can be used in more complex applications that require high performance. The idea is to move beyond applications that already use embedded systems, exploring the use of a set of units processing working together to implement an intelligent algorithm. The number of existing works in the areas of parallel processing, systems intelligent and embedded systems is wide. However works that link these three areas to solve any problem are reduced. In this context, this work aimed to use tools available for FPGA architectures, to develop a platform with multiple processors to use in pattern classification with artificial neural networks.

Keywords: Parallel Processing, Embedded Systems, Artificial Neural Networks.

“O mais competente não discute domina sua ciência e cala-se”

François-Marie Arouet – Voltaire, 1694 – 1778

Índice

Lista de Figuras

Lista de Tabelas

Lista de Símbolos e Abreviaturas

ASIC: *Application-Specific Integrated Circuit*
CLB: *Configurable Logic Block*
CLP: Controlador Lógico Programável
COMA: *Cache Only Memory Access*
DSP: Processador de Sinal Digital (*Digital Signal Processor*)
FIFO: *First In First Out*
FPGA: Dispositivo lógico programável em campo (*Field Programmable Gate Array*)
FSL: *Fast Simple Link*
HDL: *Hardware Description Language*
IA: Inteligência Artificial
IP: *Intellectual Property*
LE: *Logic Element*
LMB: *Local Memory Bus*
MIMD: *Multiple Instruction Multiple Data*
MISD: *Multiple Instruction Single Data*
µP: Microprocessador
µC: Microcontrolador
MLP: *Multi-Layer Perceptrons*
MPSoC: *Multiple Processor System on Chip*
NORMA: *No Remote Memory Access*
NUMA: *Non Uniform Memory Access*
PID: Proporcional, Integrativo e Derivativo
PWM: *Pulse Width Modulation*
OPB: *On-chip Peripheral Bus*
RBF: *Radial Basis Function*
RISC: *Reduced Instruction Set Computer*
RNA: Rede Neural Artificial
SIMD: *Single Instruction Multiple Data*
SISD: *Single Instruction Single Data*
SOPC: *System-On-a-programable Chip*
SPMD: *Single-Program Multiple Data*
SVM: Máquina de vetores de suporte (*Support Vector Machine*)
UC: Unidade de Controle
UMA: *Uniform Memory Access*
UP: Unidade de Processamento

Capítulo 1

Introdução

O número de aplicações baseadas em sistemas embarcados cresce significativamente a cada ano. Esses podem ser considerados como sistemas com tamanho reduzido, quando comparados aos sistemas com processadores de uso geral. Os sistemas embarcados possuem uma ou mais unidades de processamento dedicadas a uma aplicação específica, assim também são conhecidos como sistemas dedicados. Ao contrário dos sistemas com processadores de uso geral, os sistemas embarcados, ou dedicados, são projetados para uso em uma determinada situação, e normalmente possuem unidades de processamento mais simples.

Em [Shiqui 2004] é mostrado que pesquisadores têm utilizado diversos tipos de unidade de processamento, na execução dos algoritmos embarcados, por exemplo, microcontroladores (μC), controladores lógicos programáveis (CLPs), processadores de sinais digitais (DSP), *software processors* e *hardware processors.* Os dois últimos são implementados em dispositivos lógicos programáveis (FPGA).

Os μC são dispositivos considerados completos, pois possuem uma unidade de processamento, memória, entrada e saída no mesmo *chip*, diferente dos processadores de uso geral, que possuem componentes fisicamente separados [Byte 2002]. Os CLPs são microprocessadores (μP) dedicados utilizados em aplicações industriais [Frey 2000], enquanto os DSPs são μC normalmente usados no processamento de sinais multimídia [Furht 1997]. *Software processors* são processadores fornecidos pelo fabricante do FPGA como um programa escrito numa linguagem de descrição de *hardware* (HDL) [Coffer & Harding, 2005]. Os *hardware processors* são núcleos de processamento fornecidos em *hardware*, ao nível de portas lógicas, que podem ser encontrados em dispositivos FPGA, esses são criados e otimizados especificamente para um determinado FPGA [Coffer & Harding, 2005].

As unidades de processamento para uso embarcado são selecionadas levando em consideração fatores como: funcionalidade, preço, área e consumo de energia [Waldeck 2004][Benini 2002]. Mesmo com o aumento no desempenho dos

processadores dedicados, eles podem ter um desempenho menor que os processadores de uso geral.

Para melhorar o tempo de execução necessário nestes sistemas, pode-se analisar a implementação de soluções paralelas, para que assim possam ser utilizados em aplicações mais complexas e que exigem um alto desempenho, tais como: processamento de sinais digitais, criptografia e classificação de padrões. Neste raciocínio, ferramentas bem difundidas e utilizadas para solução de problemas relacionados a classificação de padrões são as redes neurais artificiais (RNA). Arquiteturas do tipo: percepetron de múltiplas camadas (MLP) [Haykin 2001] e máquinas de comitê [Anderson *et al* 1994, Dimitrakakis *et al* 2005], têm sido utilizadas em sistemas que utilizam processadores de uso geral, na solução de problemas relacionados a classificação de padrões.

Arribas [Arribas *et al* 2002] embarcou um FPGA em um robô para realizar o processamento da visão em tempo real. Os autores observaram que mesmo utilizando uma unidade de processamento com uma performance menor do que as utilizadas em computadores, foi possível executar essa aplicação com um bom desempenho.

Huerta [Huerta *et al* 2005] implementou um algoritmo de criptografia em um sistema embarcado, que utilizava um processador de *software* como unidade de processamento. Aplicações como transmissão e autenticação de dados utilizam esses algoritmos e estão sendo implementadas em diversos dispositivos semelhantes.

Em [Lim *et al* 2006] foi implementado um sistema para reconhecimento de voz utilizando o processador de *software MicroBlaze*. Os pesquisadores observaram que, mesmo com recursos limitados, o sistema utilizado tem um desempenho satisfatório para esse tipo de aplicação, podendo ser usado, por exemplo, em aparelhos celulares.

Um FPGA foi utilizado em [Gil *et al* 2007] para implementação de uma rede neural auto organizável para realização de diagnósticos em problemas urinários. A idéia era embarcar a plataforma em um aparelho móvel para que fosse possível realizar o diagnóstico dos problemas de forma automática.

Lee [Lee *et al* 2008] utilizou um sistema embarcado composto por duas unidades de processamento em paralelo, um DSP e um FPGA, na implementação de uma rede neural. O sistema foi desenvolvido para realizar o controle de sistemas não lineares e como exemplo de aplicação para esse sistema foi utilizado o problema do pêndulo invertido.

Em [Ganeshamoorthy *et al* 2008] foi implementada uma rede neural em um sistema de alto desempenho, com trezentas e noventa e seis unidades de processamento realizando processamento paralelo, para que fosse possível diminuir o tempo de execução da rede neural.

Esses são alguns exemplos de aplicações, nos quais sistemas embarcados, sistemas paralelos e sistemas inteligentes baseados em redes neurais podem ser utilizados.

1.1 Motivação

A quantidade e variedade de unidades de processamento existentes tem aumentado significativamente com o passar dos anos. O número de aplicações que utilizam sistemas com alguma forma de processamento é vasto, praticamente em qualquer lugar é possível encontrar um desses sistemas. Entre as diversas aplicações que realizam algum tipo de processamento, pode-se citar: forno de microondas, telefone celular, terminais bancários, automóveis e aparelhos médicos.

Assim como o número de aplicações têm aumentado, a complexidade das mesmas também têm aumentado, criando a necessidade de processadores com um desempenho maior. Portanto, em alguns casos, um único elemento de processamento não será capaz de executar a sua função em tempo hábil.

A idéia para executar uma aplicação que exige um desempenho maior do qual o processador suporta, é dividir essa aplicação em tarefas menores, e distribuí-las em mais de um processador. Por exemplo, em [Ou *et al* 2006] foi utilizado uma plataforma com vários processadores embarcados, no qual uma aplicação espacial complexa foi dividida em problemas mais simples.

Muitos engenheiros estão desenvolvendo sistemas embarcados, a escolha destes, ao invés da utilização dos processadores de uso geral, pode ser baseada em diversos fatores, tais como: custo, tamanho, desempenho e restrições ambientais.

O conhecimento adquirido anteriormente utilizando sistemas embarcados e as linhas de pesquisa em sistemas paralelos e sistemas inteligentes da Universidade Federal do Rio Grande do Norte (UFRN) motivou o desenvolvimento de aplicações unindo três campos da computação, como apresentado na Figura 1.1: os sistemas paralelos, embarcados e inteligentes.

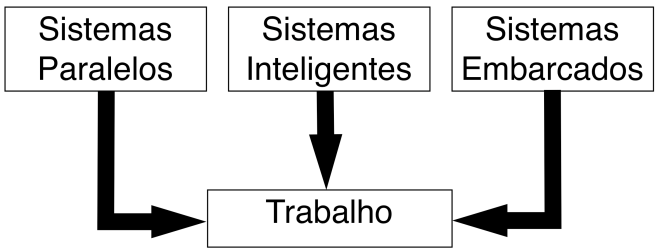

Figura 1.1 – Três grandes áreas da computação.

1.2 Objetivos

A idéia é avançar além das aplicações que já utilizam sistemas embarcados [Arribas 2002, Shiqui 2004, Huerta 2005, Lim 2006, Niu 2005, Gil 2007, Lee 2008], explorando a utilização de um conjunto de unidades de processamento cooperando entre si para execução de um algoritmo inteligente. O número de trabalhos existentes nas áreas de processamento paralelo, sistemas inteligentes e sistemas embarcados é grande. Entretanto trabalhos que unam essas três áreas para a solução de algum tipo de problema, são reduzidos.

Diante do contexto apresentado, esse trabalho teve como objetivo a implementação de redes neurais complexas. Para isso foi utilizado ferramentas disponíveis para o desenvolvimento de aplicações em arquiteturas baseadas em FPGA e técnicas de programação paralela para desenvolver uma plataforma com múltiplos processadores. Essa plataforma foi utilizada para execução de problemas que necessitavam realizar classificação de padrões. Para tanto foram estabelecidos os seguintes objetivos neste trabalho:

1. Embarcar mais de um processador em um único FPGA.
2. Realizar o mapeamento de uma arquitetura paralela em um FPGA, levando em conta aspectos ligados à definição do tipo de processador, organização da memória e formas de comunicação.
3. Documentar as informações necessárias, para utilizar uma arquitetura com vários processadores, de forma eficiente, em FPGA;
4. Implementar os algoritmos para realização da classificação de padrões.
5. Transformar os algoritmos seriais escolhidos em algoritmos paralelos, para que os mesmos fossem utilizados na arquitetura com mais de um processador.

6. Realizar testes de desempenho utilizando aplicações previamente selecionadas em função da sua complexidade.

Outra importante contribuição deste trabalho foi a implementação de estruturas de máquinas de comitê em um ambiente de desenvolvimento com restrições de desempenho como o FPGA. Os trabalhos relacionados observados limitavam-se a implementação de redes neurais mais simples como o perceptron de múltiplas camadas (MLP), enquanto nesse trabalho foram implementados dois tipos de comitês de máquinas, um composto por máquinas de vetor de suporte (SVM) e outro composto por especialistas constituídos de MLPs

1.3 Organização do Trabalho

Esse trabalho está dividido em seis capítulos, conforme descrito a seguir:

O capítulo dois apresenta conceitos sobre os sistema de computação paralela, onde serão mostradas algumas topologias de interconexão, formas de comunicação entre unidades de processamento, como transformar um algoritmo seqüencial em paralelo e como medir o desempenho do algoritmo paralelo.

No capítulo três são mostrados detalhes sobre sistemas embarcados. São apresentados conceitos básicos sobre FPGA, *software processor* Nios® II, barramento Avalon e como realizar a comunicação entre processadores no mesmo FPGA.

Já no capítulo quatro são abordados conceitos clássicos sobre redes neurais artificiais, paradigmas de aprendizagem, arquiteturas e trabalhos relacionados.

No capítulo cinco é apresentada a metodologia sugerida, os testes realizados e os resultados obtidos na plataforma proposta com múltiplos processadores.

O capítulo seis finaliza o trabalho, apresentando as considerações finais e sugestões de trabalhos que podem ser realizados como continuação deste trabalho.

As referências citadas, tais como: publicações, livros e sites da internet, encontram-se no final deste trabalho e estão classificadas em ordem alfabética.

Capítulo 2

Sistemas Paralelos

O desempenho dos sistemas computacionais aumenta a cada dia, da mesma forma que a complexidade das aplicações que são executadas nestes também aumenta. Assim, sempre existirá a necessidade de um desempenho melhor para execução de aplicações complexas. Portanto, apesar dessa evolução, existem situações, nos quais, um sistema computacional, com uma única unidade de processamento não é suficiente para realizar o processamento das informações em tempo hábil. Uma solução para resolver essas situações é a utilização de sistemas de computação paralela que têm como finalidade aumentar o poder de processamento, utilizando duas ou mais unidades de processamento trabalhando em conjunto para solucionar um problema complexo.

A idéia de processamento paralelo não é nova [Rose & Navaux 2003]. Von Neumann, por volta de 1940, sugere uma grade na qual os pontos são atualizados em paralelo para resolver equações diferenciais.

Nos anos oitenta, máquinas conhecidas como supercomputadores possuíam técnicas de sobreposição, permitindo que a execução em paralelo fosse realizada em *pipelines* diferentes (paralelismo temporal). Essas máquinas evoluíram e começaram a utilizar mais de uma unidade funcional para execução de tarefas simultâneas (paralelismo físico) [Rose & Navaux 2003].

No início da década de noventa começaram a surgir os *clusters*, que podem ser definidos como agregados de computadores, interligados através de uma rede local dedicada para realização de processamento de alto desempenho [Pitanga 2008].

Em [Rose & Navaux 2003] é comentado que, a partir do final da década de noventa, começa a acontecer uma gradativa passagem das técnicas de arquiteturas paralelas, empregadas nos supercomputadores, para dentro das arquiteturas dos microprocessadores.

Diante do contexto apresentado, este capítulo tem o objetivo de apresentar conceitos sobre processamento paralelo, discutidos na literatura, que são necessários para um melhor entendimento do trabalho realizado.

2.1 Arquiteturas paralelas

Michael Flynn em [Flynn 1966] caracterizou os diversos modelos existentes de arquiteturas de sistemas computacionais, segundo o fluxo de dados e de instruções. Essa classificação é conhecida como taxionomia de Flynn [Rose & Navaux 2003, Foster 1995, Hwang 1985] e define quatro classes de arquiteturas.

Arquiteturas SISD (*Single Instruction stream – Single Data stream*) são as mais simples, onde o sistema é considerado seqüencial, pois somente uma instrução é executada por vez em um dado instante de tempo. Para cada instrução, são lidos ou escritos na memória apenas os dados envolvidos na mesma. As máquinas baseadas nas arquiteturas de Von Neumann tradicionais, sem qualquer tipo de paralelismo, são exemplos desse modelo. A Figura 2.1 mostra um diagrama que representa esse tipo de arquitetura que é composta por uma unidade de controle (UC), uma unidade de processamento (UP) e uma memória (MEM).

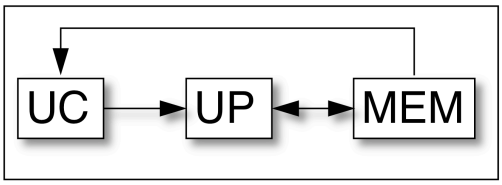

Figura 2.1 – Arquitetura SISD.

Arquiteturas MISD (*Multiple Instruction stream – Single Data stream*) seriam máquinas que executam várias instruções ao mesmo tempo sobre um único conjunto de dados ou operandos, entretanto não existem exemplos reais dessas máquinas.

Arquiteturas SIMD (*Single Instruction stream – Multiple Data stream*) implementam o equivalente ao paralelismo de dados, no qual uma simples instrução é executada paralelamente utilizando vários conjuntos de dados diferentes e de forma síncrona, como, por exemplo, as placas de aceleração de vídeo. Em um sistema SIMD uma única UC envia as mesmas instruções para várias UP, como na Figura 2.2.

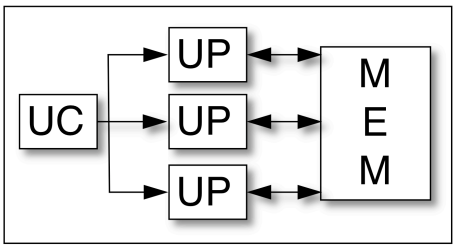

Figura 2.2 – Arquitetura SIMD.

Arquiteturas MIMD (*Multiple Instrucion stream – Multiple Data stream*) referem-se ao modelo de execução paralela no qual cada processador está essencialmente agindo de forma independente, havendo, portanto, múltiplos fluxos de instruções e múltiplos dados, como se fossem um conjunto de máquinas SISD, onde cada processador é capaz de executar um problema diferente, ou parte dele.

Entretanto, ainda na década de oitenta, a classificação de Flynn tornou-se ineficiente para classificar as variedades de arquiteturas MIMD existentes [Foster 1995, Hwang 1985]. Atualmente essa categoria engloba uma diversidade de modelos e podem ser classificadas, por exemplo, quanto a organização e acesso à memória.

Independente do modelo MIMD escolhido, ambos necessitam que as UP possam se comunicar e assim trocar informações para trabalhar em conjunto. Esses modos de comunicação são apresentados na seção a seguir.

2.2 Comunicação entre arquiteturas paralelas

Na seção anterior foram vistos as diferentes arquiteturas de sistemas computacionais. Nessa seção são apresentadas as formas na qual os dois tipos de arquiteturas MIMD podem trocar informações entre as diferentes UP existentes.

A Figura 2.3 ilustra os tipos de arquitetura MIMD, de acordo com a utilização da memória: (a) compartilhada e (b) distribuída. Na arquitetura MIMD com memória distribuída, cada UP possuí sua própria memória, sendo essa inacessível para as outras UP. Assim para que ocorra troca de informações entre as UP é necessário um quarto elemento, denominado de rede de interconexão (RED).

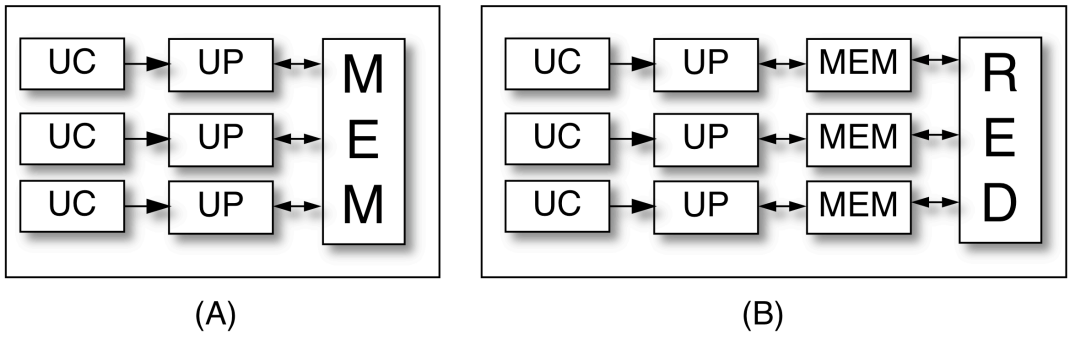

Figura 2.3 – Arquitetura MIMD, (a) Memória Compartilhada e (b) Memória Distribuída.

Portanto, existem dois paradigmas para realização da comunicação: o primeiro é a utilização da memória compartilhada e o segundo é a comunicação através da passagem de mensagens.

2.2.1 Memória compartilhada

A memória compartilhada consiste de um espaço de endereçamento global. Todas as UPs podem ler deste e escrever neste espaço. A memória compartilhada ainda pode ser subdividida em dois modelos [Rose & Navaux 2003, Hwang 1985, Tanenbaum 2007]: UMA (*Uniform Memory Access*) e NUMA (*Non Uniform Memory Access*).

Nos sistemas UMA, cada UP necessita da mesma quantidade de tempo para acessar qualquer um dos módulos de memória. Em outras palavras, cada palavra pode ser lida ou escrita tão rapidamente quanto qualquer outra, independente da sua posição [Tanenbaum 2007]. As máquinas UMA mais simples são baseadas em um único barramento de comunicação com duas ou mais UP e um ou mais módulos de memória, todos usando o mesmo barramento, como apresentado na Figura 2.4.

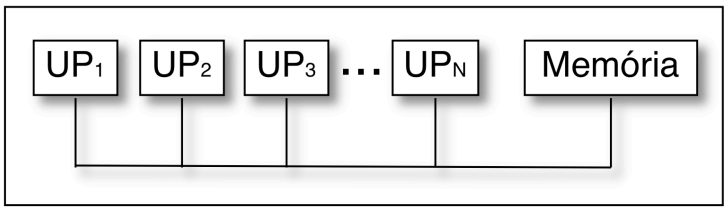

Figura 2.4 – Máquinas UMA.

Quando a UP necessita ler ou escrever uma palavra em um endereço da memória, em primeiro lugar essa UP verifica se o barramento está ocupado. Se o barramento estiver ocioso, a UP escreve o endereço da palavra que o mesmo deseja acessar, ativa alguns sinais de controle e espera até que a memória escreva ou leia a palavra desejada no barramento de dados. Se o barramento estiver ocupado quando uma outra UP desejar ler ou escrever na memória, essa espera até que o barramento esteja novamente ocioso e possa ser utilizado.

O problema desse modelo de arquitetura é justamente a utilização de um único barramento. Com duas ou três UP, a contenção no barramento é administrável, com várias a disputa pelo acesso ao barramento torna esse projeto impraticável. Dessa forma, o barramento se torna o gargalo do sistema.

Para conseguir utilizar várias UP alguma característica do modelo UMA deve ser modificada [Stallings 2002]. Usualmente, o que se muda é a idéia de que todos os módulos de memória necessitem da mesma quantidade de tempo para serem acessados. Portanto, essa propriedade não é mais válida para o segundo modelo, as máquinas NUMA. Nessas máquinas, existe um módulo de memória distinto para cada UP, conectado por um barramento exclusivo, como visto na Figura 2.5. O tempo necessário para uma UP acessar o seu próprio módulo de memória é menor do que o tempo necessário para utilizar os outros módulos.

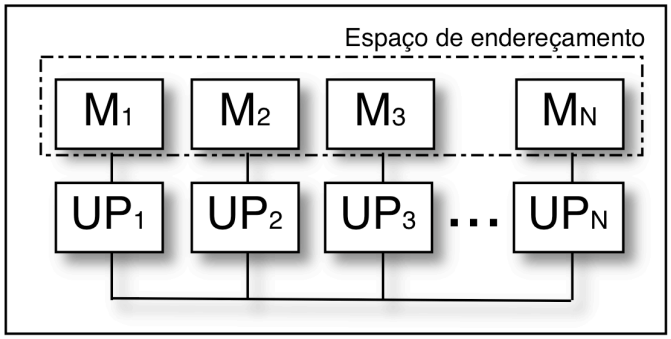

Figura 2.5 – Máquinas NUMA.

Apesar do modelo NUMA possuir diferentes módulos de memória, o espaço de endereçamento lógico desses módulos é o mesmo, ou seja, esses são vistos pelo sistema como um único módulo de memória compartilhada. Já que o acesso aos módulos de memória das outras UP é mais lento nas máquinas NUMA, essas máquinas são mais lentas, quando comparadas a máquinas UMA com a mesma freqüência de operação e quantidade de UP. Assim, programas desenvolvidos para

máquinas UMA podem ser executados em máquinas NUMA, sem a necessidade de modificação, embora o seu desempenho seja inferior.

A comunicação através do uso de uma memória compartilhada é simples de ser implementada, sendo realizada de forma eficiente com operações do tipo `load` e `store`. Entretanto, quando dois ou mais processadores desejam acessar simultaneamente uma mesma posição de memória, podem surgir problemas, como as condições de corrida.

As condições de corrida ocorrem quando duas ou mais instruções localizadas em UPs concorrentes acessam a mesma posição de memória e pelo menos uma das instruções envolve escrita. Como não há garantia de ordem de execução, o determinismo não existe [Tanenbaum 1999].

Por exemplo, na Figura 2.6, quando UP_1 deseja adicionar 1 ao valor da variável *X*, que encontra-se na memória associada ao valor 0. A UP_2 deseja, no mesmo instante, adicionar 2 à mesma variável. Qual o resultado?

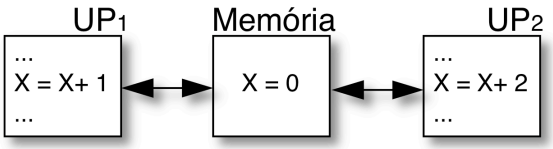

Figura 2.6 – Duas UP acessando simultaneamente a mesma posição de memória.

Três resultados podem ser obtidos. Se a UP_1 executa sua instrução de forma completa antes da UP_2 ler o valor de *X*, o resultado é 3. Da mesma forma, se a UP_2 executa sua instrução de forma completa antes da UP_1 ler o valor de *X*, o resultado também é 3. Se a UP_1 ou a UP_2 realizam suas leituras antes que o outro tenha completado a execução da sua instrução, o resultado depende de quem termina a execução por último: Se a UP_1 é a última a terminar, o resultado é 1. Se UP_2 é a última, o resultado é 2.

As condições de corrida podem ser evitadas não permitindo que mais de um processador leia e grave dados na memória compartilhada ao mesmo tempo, em outras palavras, garantindo a exclusão mútua [Tanenbaum 1999]. Assim, o problema do exemplo anterior é resolvido através da sincronização do uso dos dados compartilhados. As sessões de memória que possuem esses dados são chamadas regiões críticas. Para realizar a sincronização dessas regiões pode-se utilizar, por exemplo, semáforos ou ferrolhos.

O ferrolho, ou variável de bloqueio [Tanenbaum 1999], é uma variável compartilhada inicialmente com valor zero. Quando um processador quer entrar em uma região crítica, ele primeiro testa a variável de bloqueio. Se o bloqueio for 0 (`unlock`), o processador o define como 1 (`lock`) e entra na região crítica, se o bloqueio for 1 o processador apenas espera o bloqueio ser igual a 0. A Figura 2.7 mostra uma região crítica de *memória* que utiliza uma variável de bloqueio *X*.

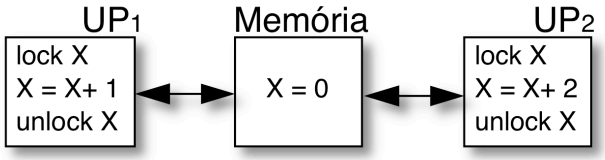

Figura 2.7 – Utilização de ferrolhos para garantir o acesso mutuamente exclusivo.

O problema dessa técnica é que se os dois processadores lerem a variável de bloqueio, ambos entrarão na região crítica ao mesmo tempo. Dijkstra em 1968 propôs uma solução geral para o problema da sincronização de processos paralelos [Tanenbaum 1999]. Em algum lugar da memória há algumas variáveis inteiras não negativas denominadas semáforos. Dijkstra propôs duas operações que operam nos semáforos, `up` e `down`.

A operação `down` em um semáforo verifica se o valor é maior que 0, se for ele diminui o valor e continua, caso contrário o processador é colocado em espera, sem completar a operação. A operação `up` incrementa o valor do semáforo, se um ou mais processadores estiverem em espera, incapazes de completar uma operação `down` anterior, um deles é escolhido e autorizado a completar a sua operação.

As operações `up` e `down` são realizadas como uma única e indivisível ação, garantindo que uma vez que uma operação de semáforo seja iniciada, nenhum outro processador acesse o semáforo até que a instrução tenha sido executada ou bloqueada. Assim o problema existente no uso dos ferrolhos para sincronização da utilização da memória compartilhada é solucionado.

Máquinas que implementam comunicações via memória compartilhada normalmente não permitem a utilização de um grande número de unidades de processamento como visto na seção anterior. Entretanto, a utilização de uma memória compartilhada é a forma mais simples de se implementar a comunicação entre processadores. Nesta classe incluem-se todas as máquinas com múltiplos

processadores que compartilham um espaço de endereços de memória comum, também conhecidas como multiprocessadores.

A sub-seção seguinte mostra as arquiteturas que utilizam a passagem de mensagens, como forma de comunicação, que é o caso das arquiteturas com memória distribuída.

2.2.2 Memória distribuída

Nesta classe incluem-se as máquinas formadas por várias unidades de processamento, cada uma com a sua própria memória. Cada unidade constituída de unidade de processamento, memória local e dispositivos de entrada e saída é denominada nó. Neste caso, não existe qualquer tipo de memória comum e a comunicação entre os diferentes nós do sistema é realizada através da troca de mensagens.

O modelo que rege os MIMD com memória distribuída é o NORMA (*No Remote Memory Access*). Nesse modelo, ilustrado na Figura 2.8, não existe uma memória primária compartilhada ao nível de arquitetura, em outras palavras uma UP não pode acessar a memória conectada a outra UP, apenas executando um `load`. Para que isso aconteça é necessário o envio explícito de uma mensagem e aguardar por uma resposta.

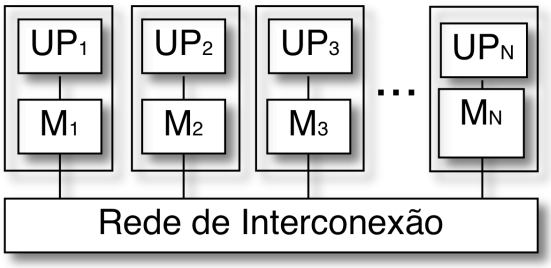

Figura 2.8 – Máquinas NORMA.

Como cada processador tem sua própria memória, surge a necessidade de alguma forma de interconexão entre as UP, que permita o acesso a todas as memórias por todas as UP. Se uma UP necessita de dados que não se encontram armazenados em sua memória local, estes podem ser acessados usando instruções próprias, tais como, `send` e `receive`, que implementam a comunicação. Pode-se construir máquinas desse tipo com milhares de unidades de processamento.

Uma possível desvantagem da utilização da memória distribuída é que sua programação é mais difícil, devido à necessidade da utilização de protocolos para comunicação. Entretanto, essa desvantagem é superada utilizando bibliotecas implementadas e disponibilizadas pelos fabricantes. Assim, a troca de mensagens é realizada através de primitivas do tipo `send` e `receive`. Como é necessário uma rede de interconexão externa para realizar a comunicação, o tempo de comunicação também é maior do que o da memória compartilhada. Portanto, se a comunicação entre os nós for intensa, o desempenho final pode acabar sendo comprometido.

A forma como as unidades de processamento de uma arquitetura estão conectadas é definida pela topologia de interconexão. Essa pode ser modelada como grafos [Tanenbaum 2007], sendo os caminhos representados por arcos e as unidades de processamento por nós. A topologia descreve como os nós e caminhos estão organizados e influencia diretamente na eficiência da troca das mensagens. Dentre os critérios existentes para a escolha da topologia de interconexão pode-se citar:

- Escalabilidade: capacidade de suportar o aumento de UP sem a necessidade de realizar grandes modificações;
- Desempenho: velocidade e quantidade de dados a serem transferidos.
- Custo: Quanto mais ligações existir entre os nós, maior é o custo da topologia;
- Conectividade: número de caminhos alternativos entre um par qualquer de UP. Essa característica está diretamente ligada à tolerância a falhas do sistema.
- Diâmetro: corresponde à distância existente entre os dois nós mais afastados. Essa é medida pelo número de arcos necessários para que a comunicação seja realizada. Quanto menor for o diâmetro, melhor será o desempenho no pior caso.

Sistemas de computação paralela possuem, em sua maioria, topologias regulares, ao contrário dos sistemas distribuídos que, por causa da posição geográfica e de sua integração, possuem topologias irregulares [Rose & Navaux 2003]. Entre os exemplos de topologias de interconexão pode-se citar: estrela, barramento, árvore, anel, grade, hipercubo, dentre outros.

A estrela é uma topologia de dimensão zero na qual os nós estariam ligados aos outros, como visto na Figura 2.9. Este nó central tem como função realizar a

comutação das mensagens. Embora seja um projeto simples, no caso de um grande sistema é provável que o nó central seja um gargalo para o mesmo. Além disso, essa topologia não apresenta uma boa tolerância a falhas, já que uma falha no nó central pode destruir completamente o sistema.

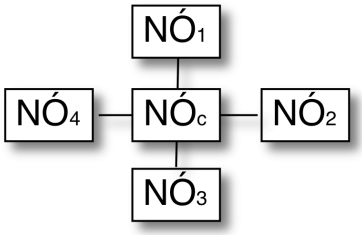

Figura 2.9 – Topologia de Interconexão Estrela.

O anel é uma topologia considerada unidimensional, pois cada mensagem pode ser enviada por um dos dois caminhos existentes, como na Figura 2.10. Uma desvantagem dessa topologia é que caso algum nó apresente falha, pode degradar o desempenho do sistema, já que o seu diâmetro aumentará.

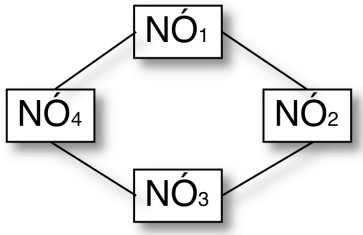

Figura 2.10 – Topologia de Interconexão Anel.

A grade ou malha, como visto na Figura 2.11(a) é um projeto de topologia bidimensional que tem sido usado em muitos projetos de sistemas distribuídos comerciais. Essa é de alta regularidade, fácil de ampliar para tamanhos maiores e tem um diâmetro que aumenta apenas com a raiz quadrada do número de nós. Uma variante da grade é o toro duplo, visto na Figura 2.11(b), que é uma grade cuja extremidades são conectadas. Além de ser mais tolerante a falhas do que a grade, seu diâmetro também é menor, porque as arestas opostas agora podem se comunicar.

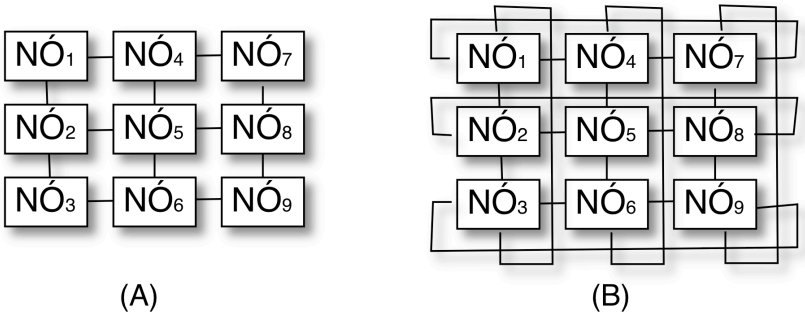

Figura 2.11 – Topologias de Interconexão: (a) grade e (b) toro duplo.

Uma outra topologia, vista na Figura 2.12(a) é a árvore. Um problema com esse projeto é que haverá muito tráfego perto do topo da árvore, assim os nós do topo se tornarão gargalos do sistema. Um modo de contornar esse problema é aumentar a quantidade de caminhos dos nós mais próximos do topo. Por exemplo, os caminhos das folhas nos níveis mais baixos poderiam ter uma quantidade de caminhos *b*, o próximo nível poderia *2b*, e os caminhos do nível superior *4b*. Esse projeto é denominado árvore gorda [Rose & Navaux 2003] e é visto na Figura 2.12(b).

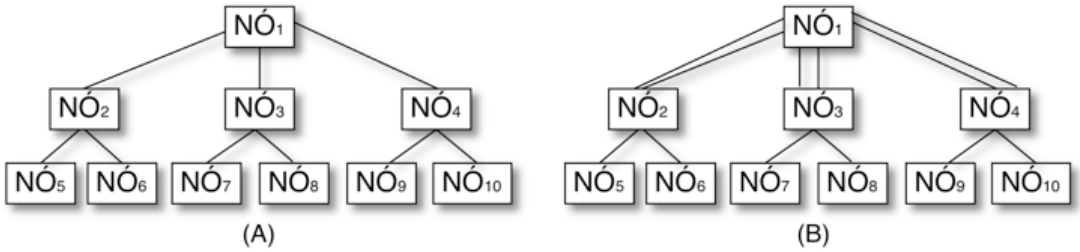

Figura 2.12 – Topologias de Interconexão: (a) árvore e (b) árvore gorda.

A última topologia mostrada é o barramento. Essa é uma alternativa de menor custo, porém por ser o único canal de comunicação, como visto na Figura 2.13, para todas as UPs existentes, têm baixa tolerância a falhas e é altamente bloqueante. Por isso sua escalabilidade acaba sendo comprometida, já que existirá uma disputa para utilizar o canal de comunicação. Essas deficiências podem ser amenizadas pela utilização de vários barramentos em paralelo.

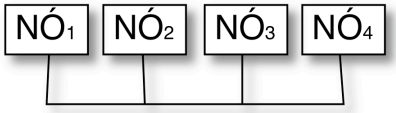

Figura 2.13 – Topologias de Interconexão barramento.

Um fator que pode ser decisivo na escolha de uma topologia é sua adequação a uma classe específica de algoritmos. No caso ideal, o padrão de interconexão da topologia corresponde exatamente ao padrão de comunicação da aplicação paralela que é executada na máquina. Por exemplo, a árvore favorece a execução de algoritmos de divisão e conquista. Enquanto malhas são adequadas aos problemas nos quais uma estrutura de dados bidimensional tem que ser processada de forma particionada como, por exemplo, na operação de matrizes. Na seção seguinte é mostrado como projetar algoritmos para serem executados em arquiteturas paralelas.

2.3 Algoritmos paralelos

O desenvolvimento de algoritmos paralelos não é uma tarefa trivial [Foster 1995]. Não existe uma receita para desenvolver um bom algoritmo paralelo, entretanto em [Foster 1995] é definida uma metodologia dividida em etapas que ao serem seguidas facilitam o desenvolvimento do algoritmo.

O primeiro passo no projeto de um algoritmo paralelo é a decomposição do problema global em problemas menores e mais simples. Isto pode ser feito de duas formas: o conjunto de dados de entradas é dividido em subconjuntos, ou a tarefa inicial é divida em pequenas tarefas. Tarefas podem ser consideradas como computações independentes do ponto de vista do algoritmo, ou programas independentes do ponto de vista da arquitetura.

Utilizando tarefas menores, ao invés de uma maior, é possível atribuir cada tarefa a uma unidade de processamento diferente para que essas possam ser executadas de forma simultânea. Da mesma maneira que a utilização de subconjuntos de entrada permite o processamento dos mesmos simultaneamente. Basicamente, existem dois tipos de decomposição [Foster 1995]: do domínio e funcional.

Na decomposição do domínio, o conjunto de dados de entrada é dividido em subconjuntos. Neste caso, cada UP processa apenas os dados do subconjunto que lhe foi atribuído. Os processos podem também ter a necessidade de se comunicarem periodicamente para a troca de informação. O modelo *Single-Program Multiple-Data* (SPMD) segue essa idéia, onde o código a ser executado é igual para todas as UP. Essa estratégia de decomposição nem sempre leva ao algoritmo mais eficiente para a

arquitetura paralela. Por exemplo, quando os conjuntos de dados atribuídos às diferentes UP requerem tempos de execução muito diferentes entre si.

A segunda forma de decomposição é a funcional, em outras palavras, o paralelismo de tarefas. Nesta, o problema é decomposto em um número de tarefas menores que o inicial e então essas tarefas são distribuídas para às UPs existentes [Foster, 1995].

Independente da decomposição escolhida, o objetivo ao se desenvolver algoritmos paralelos é explorar a concorrência e a escalabilidade das arquiteturas paralelas. Conforme descrito em [Foster 1995], o desenvolvimento de uma aplicação utilizando os princípios da programação paralela envolve as etapas, mostradas na Figura 2.14, de particionamento, comunicação, aglomeração e mapeamento.

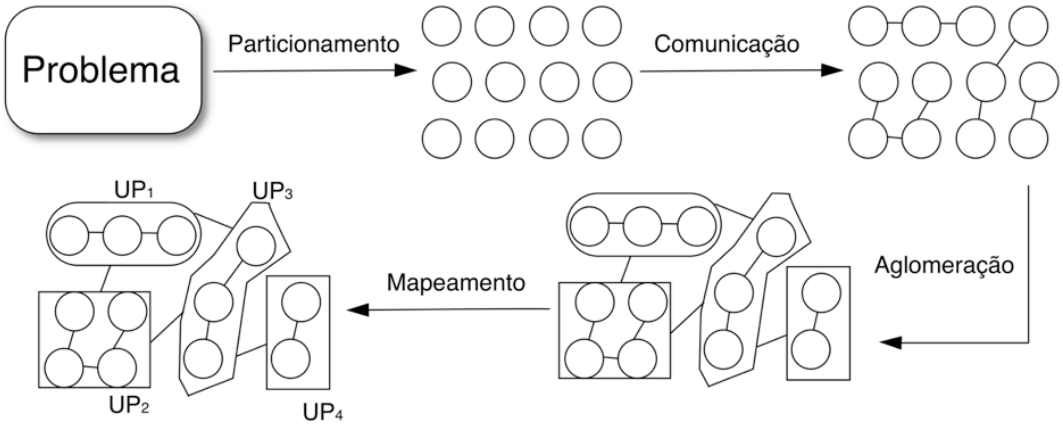

Figura 2.14 – Etapas de desenvolvimento de um algoritmo paralelo.

No particionamento, o problema é observado e procura-se a melhor forma de dividi-lo em sub-tarefas, de forma a aproveitar a maior possibilidade de exploração do paralelismo. Nesta etapa não há preocupação onde cada tarefa será executada nem por quantas unidades de processamento.

Na etapa de comunicação é estabelecida todas as conexões necessárias entre as tarefas para que o algoritmo possa ser executado. Todo o fluxo de informação deve ser analisado quanto à sua permanência, alteração e alcance.

A terceira etapa, à aglomeração, avalia a divisão das tarefas e comunicação. Esta etapa viabiliza o agrupamento de tarefas de forma a ponderar processamento e comunicação, tentando-se estabelecer o melhor custo/benefício entre ambos. Aqui é feito um estudo da granularidade, que diz respeito ao tamanho das tarefas, ou seja, a quantidade de processamento necessária para sua execução. Quanto maior a tarefa,

mais tempo de computação é consumido por essa, e quanto mais tarefas existir maior a sua granularidade.

Na última etapa, o mapeamento, cada tarefa é atribuída a uma unidade de processamento da arquitetura paralela, tendo o cuidado de acomodar de forma adequada tarefas que necessitam um maior processamento em UPs de melhor desempenho e comunicações mais intensas entre UPs mais próximas.

A análise de desempenho é adicionada em [Alves 2002] como mais uma etapa do método proposto por Foster, pois dependendo do resultado obtido nesta etapa, torna-se necessário o retorno a um dos passos anteriores.

2.4 Análise de desempenho

Um dos objetivos do desenvolvimento de algoritmos para arquiteturas paralelas é que esses executem mais rápidos do que em uma máquina com uma única unidade de processamento. Se o objetivo desejado não for alcançado não valerá a pena ter uma máquina paralela. O preço também é uma importante métrica, pois uma máquina paralela com o desempenho duas vezes melhor que uma com uma única unidade de processamento, mas que custa mais que o dobro dessa arquitetura, não é interessante.

Outras métricas, que podem ser levadas em consideração, são a quantidade de memória utilizada, portabilidade, escalabilidade, vazão, latência e os custos de: desenvolvimento, projeto, verificação e manutenção [Foster 1995]. Por exemplo, em sistemas de tempo-real as tarefas devem ser executadas em um tempo determinado, para que o sistema seja viável.

Análise de desempenho envolve técnicas que auxiliam ao projetista obter informações sobre o desempenho na execução de algoritmos em arquiteturas específicas. O desempenho depende de características como: topologia de interconexão, granularidade das tarefas e custo de comunicação.

O ideal seria que o ganho de velocidade fosse igual a quantidade de nodos computacionais empregados. No entanto isso é raro de acontecer pois, quando se desenvolve um algoritmo, necessita-se de ajustes na sincronização entre processos e troca de mensagens entre os nós o que representa atrasos, que são chamados de custo de paralelização.

O aumento no número de processadores deveria trazer um igual aumento de desempenho. Entretanto, além do custo de paralelização, existe um nível saturação no qual não adianta aumentar o nível de processadores porque o desempenho não vai melhorar. O ganho de velocidade ideal é difícil de ser obtido, já que, segundo [Amdhal, 1967] os algoritmos sempre possuem dois trechos de implementação. O primeiro é a parte seqüencial que é executada de forma seqüencial, e a segunda é o trecho de código que pode ser paralelizado.

Portanto, é importante definir a melhor quantidade de unidades de processamento a serem utilizadas em uma determinada tarefa. Esse valor é influenciado e varia de acordo com o algoritmo e a técnica de programação utilizada.

Se uma tarefa ao ser executada em uma máquina seqüencial leva *T* segundos, ao paralelisar, ela tem uma parte que obrigatoriamente é seqüencial [Amdahl, 1967]. Definimos como *p* a porção seqüencial do algoritmo. Assim, a parte paralela pode ser representada por *1-p*. Quando executada em uma máquina paralela de N nós, o tempo gasto total (T_t) para execução do algoritmo é dado por:

$$T_t = pT + (1-p)\frac{T}{N} \tag{2.1}$$

Da equação 2.1, pode ser observado que somente haverá melhoria na parte paralelizável do algoritmo. O desempenho do algoritmo varia de acordo com o tamanho da instância e do número de unidades de processamento, por isso é importante realizar uma extrapolação das observações. O ideal é que à aceleração no processamento fosse igual ao número de unidades de processamento. Esta aceleração também conhecida por ganho ou *speed-up* é calculada utilizando a equação 2.2.

$$aceleração_p(n) = \frac{T_{es}}{T_{ep}} \tag{2.2}$$

Onde T_{es} é o tempo de execução seqüencial e T_{ep} é o tempo de execução paralelo. O tempo de execução em paralelo é a soma dos tempos de processamento (T_P), comunicação (T_C) e espera (T_E). Assim pode-se dizer que o tempo de execução em paralelo pode ser obtido através da equação 2.3.

$$T_{ep} = T_P + T_C + T_E \tag{2.3}$$

Três fatores podem influenciar no valor desta aceleração: sobrecarga da comunicação, nível de paralelismo utilizado, e parte do algoritmo seqüencial. Portanto, para melhorar o tempo de execução paralelo é importante trabalhar a granularidade na etapa da aglomeração, tentando reduzir o tempo de comunicação e o tempo de espera do algoritmo. Quanto menores forem esses tempos mais próximo do ganho ideal o algoritmo estará.

A análise de Amdahl parte do princípio de que o trecho seqüencial do algoritmo é igual independente do número de processadores utilizados. Gustafson [Gustafson & Barsis 1988] argumentaram que, do ponto de vista prático, isso não é uma verdade absoluta, já que os algoritmos crescem com o aumento da capacidade computacional e a quantidade de código seqüencial não aumentam na mesma proporção. Assim, algoritmos reais poderiam ter seu trecho seqüencial reduzido com o aumento do número de processadores. Observando resultados práticos, em Gustafson [Gustafson & Barsis 1988] é sugerido uma alternativa à lei de Amdahl, postulando assim a lei de Gustafson-Barsis. Essa é utilizada para realizar o cálculo o *scaled speedup (SS(p))*, como descrito a seguir.

Seja T_s o tempo de execução do trecho seqüencial do algoritmo e $T_p(n)$ o tempo de execução do trecho paralelo do algoritmo para n unidades de processamento. Considerando o tempo de execução seqüencial e paralelo fixo, como na equação 2.4.

$$T(n) = T_s + T_P(n) = 1 \tag{2.4}$$

O *SS(p)* pode ser calculado através da equação expressa em 2.5:

$$SS(p) = \frac{T_s.n.T_p(n)}{T_s + T_p(n)} = T_s + p.T_p(n) = n + (1-n)T_s \tag{2.5}$$

Ao contrário da lei de Amdahl, onde a carga computacional é fixa, na lei de Gustafson considera-se o tempo fixo. Uma outra medida de desempenho é a eficiência que é calculada através da equação 2.6.

$$Eficiência = \frac{aceleração_p(N)}{P} \tag{2.6}$$

onde *P* é a quantidade de unidades de processamento utilizadas. O valor de eficiência ideal é 1, ou seja, para cada nova unidade de processamento adicionado, existe um ganho de processamento equivalente a nova unidade.

2.5 Tendências

Embora as unidades de processamento estejam com freqüências de operação cada vez mais rápidas, as demandas impostas estão crescendo no mínimo com a mesma rapidez [Tanenbaum 2007]. Mesmo a freqüência de operação aumentando a cada dia, essa não pode aumentar indefinidamente devido a fatores físicos, como dissipação de calor e miniaturização dos transistores.

Portanto para resolver problemas cada vez mais complexos, projetistas estão recorrendo cada vez mais a computação paralela, a tendência é a construção de *chips* com mais de uma unidade de processamento, tais como, os atuais processadores *dual core* e *quad core*, com dois e quatro núcleos, respectivamente.

Dentre as aplicações nas quais unidades de processamento com mais de um núcleo são utilizadas, estão os servidores de alta tecnologia e equipamentos eletrônicos, tais como DVD e organizadores pessoais. Os processadores com mais de um núcleo no mesmo *chip*, podem ser chamados de multiprocessadores em um *chip* e são classificados em homogêneos e heterogêneos.

Multiprocessadores em um *chip* são considerados homogêneos quando os núcleos existentes nele são iguais e podem desempenhar as mesmas funções, enquanto nos heterogêneos os núcleos são diferentes e são projetados para a execução de funções específicas. Enquanto os primeiros são utilizados em aplicações mais genéricas, os heterogêneos são utilizados, por exemplo, em equipamentos eletrônicos como reprodutores de DVD. Um núcleo pode ser utilizado no processamento da descompressão de vídeo, enquanto o outro na descompressão de áudio.

Como comentando em [Rose & Navaux 2003] é possível observar a tendência da utilização das técnicas de processamento nos sistemas paralelos, para dentro das arquiteturas de processadores. Desde 2001 é comum encontrar-se computadores pessoais com processadores com mais de um núcleo [Wolf 2003]. Esses são mais simples que processadores mais rápidos com um único núcleo.Assim, devido a essa simplicidade, estes sistemas consomem menos energia e conseqüentemente produzem menos calor [Gorder 2007]. Mesmo processando em uma freqüência mais baixa os

processadores *multicores* possuem em geral um desempenho melhor como mostrado na Figura 2.15.

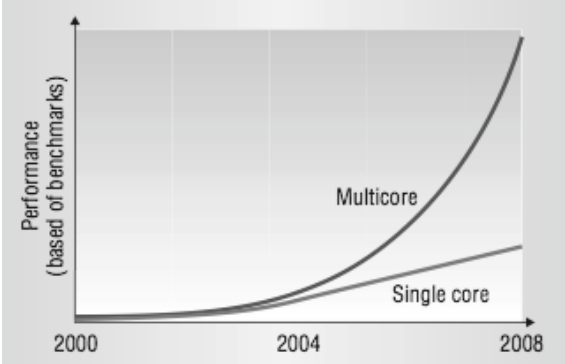

Figura 2.15 – Desempenho dos processadores com múltiplos núcleos [Geer 2005].

2.6 Conclusão

Neste capítulo foram apresentados os conceitos básicos referentes a computação paralela. As formas de comunicação entre processadores utilizando memória compartilhada e passagem de mensagens foram apresentas, assim como, a metodologia apresentada por Foster para transformação de um algoritmo seqüencial em um algoritmo paralelo.

O texto apresentado não tem a pretensão de abordar todo o assunto relacionado a essa área de pesquisa, e sim os conceitos necessários para o entendimento dos experimentos realizados e que serão apresentados no capítulo cinco. No próximo capítulo serão apresentados conceitos sobre sistemas embarcados, os quais, podem possuir uma ou mais unidades de processamento. Assim, quando existir mais de uma unidade de processamento os conceitos apresentados neste capítulo são utilizados.

Capítulo 3

Sistemas Embarcados

Sistemas embarcados, conforme apresentado no primeiro capítulo, são aqueles projetados para realizar uma tarefa específica, tornando possível a otimização dos seus projetos. Esses sistemas podem ser projetados utilizando apenas os componentes necessários para realização da tarefa em questão, possibilitando a redução de custos, tamanho e consumo de energia. A finalidade dos sistemas embarcados é diferente dos sistemas que utilizam processadores de uso geral. Enquanto os últimos a cada dia agregam mais recursos, aumentando seus processamentos e capacidades de armazenamento para realização das mais diversas tarefas. Os sistemas embarcados são utilizados justamente onde não é viável a utilização dos processadores de uso geral, tais como, celulares, sensores e automóveis.

Projetistas possuem diversos ambientes para desenvolver seus protótipos, com diferentes dispositivos como, por exemplo, μC, processadores de sinais digitais (DSP – *Digital Signal Processor*) e arranjo de portas programáveis em campo (FPGA – *Field Programmable Gate Array*).

Dentre os fatores existentes que influenciam na escolha da plataforma de desenvolvimento, pode-se citar a alta flexibilidade e o tempo de desenvolvimento reduzido que mostram o FPGA como uma boa escolha [Yiannacouras *et al* 2005]. Além desses fatores, o FPGA tem se tornado um dispositivo com maior desempenho e menor custo, quando comparado a plataformas semelhantes. Logo, essa plataforma de desenvolvimento tem sido utilizada no projeto de sistemas embarcados para as mais diversas aplicações [Niu *et al* 2005, Huerta *et al* 2005, Lim *et al* 2006, Arribas *et al* 2002].

Neste trabalho foi utilizada uma plataforma de desenvolvimento baseada em um dispositivo FPGA para realizar o projeto de uma arquitetura embarcada com múltiplos processadores. Nesse capítulo são apresentados os conceitos necessários, sobre essa tecnologia, para um melhor entendimento dos experimentos realizados.

3.1 FPGA

O FPGA é um circuito integrado composto basicamente por blocos lógicos dispostos em forma de matriz, blocos de memória, blocos de entrada e saída e uma lógica de interconexão programável [Castro 2007], como pode ser observado na Figura 3.1.

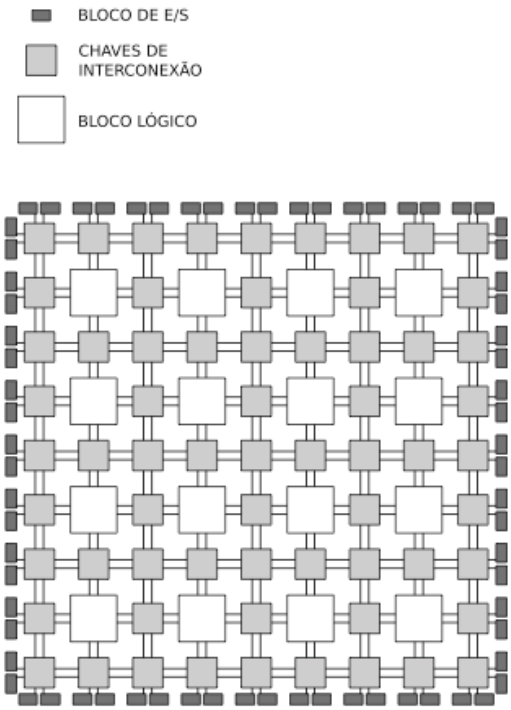

Figura 3.1 – Arquitetura típica de um FPGA [Castro 2007].

A funcionalidade do FPGA pode ser configurada através de programas desenvolvidos em linguagens de descrição de *hardware* (HDL – *Hardware Description Language*). As funções lógicas desejadas são implementadas nos blocos lógicos, também conhecidos por CLB (*Configurable Logic Block*) ou LE (*Logic Element*) [Tocci 2008], cuja nomenclatura varia de acordo com o fabricante. Neste trabalho, o termo adotado é o LE. A estrutura interna e a quantidade de LE existentes são características específicas de cada estrutura de FPGA.

Quando devidamente programado, um FPGA pode desempenhar a função dos mais diversos tipos de dispositivos lógicos, dos mais simples aos mais complexos, por exemplo, processadores. Os processadores descritos em HDL para implementação em FPGA são conhecidos como *software processors*. Alguns FPGAs possuem uma parte dedicada do seu circuito para ser utilizada como um processador físico, denominado

hardware processor, normalmente com desempenho melhor que o do *software processor*.

Entretanto, o número fixo de processadores físicos existente no FPGA pode não existir na quantidade desejada, e podem também não possuir o desempenho suficiente para a execução de aplicações. Os *software processors*, apesar de não possuírem o mesmo desempenho dos processadores físicos, possuem a vantagem de serem completamente configuráveis. Esses podem ser ajustados para gerar o melhor desempenho em uma aplicação específica [Yiannacouras *et al* 2005], além de permitirem a implementação de múltiplos processadores em um único FPGA.

Atualmente existem diversos *software processors* disponíveis em HDL, como por exemplo, o Leon 3, OpenSPARC [OpenSPARC 2008], OpenRISC [OpenRISC 2009], PicoBlaze, MicroBlaze, Nios® e Nios® II. As duas maiores fabricantes de FPGA, a Xilinx® e a Altera®, disponibilizam em conjunto com seus dispositivos, seus *soft processors*, desenvolvidos e validados por elas. Neste trabalho foram utilizados múltiplos *soft processors*. Nas seções seguintes são descritos dois desses processadores: o Microblaze® [Xilinx 2009] e o Nios® II.

3.2 Microblaze

A fabricante Xilinx distribui em conjunto com algumas famílias de FPGA o Microblaze®, o *software processor* desenvolvido por ela. Esse é um processador RISC de 32 bits, que pode ser configurado pelo projetista, já que o mesmo é apresentado como um componente de *software* [Ouelette *et al* 2005].

O Microblaze® possui três barramentos para conexão: LMB (*Local Memory Bus*), OPB (*On-Chip Peripheral Bus*) e o FSL (*Fast Simple Link*); unidade de ponto flutuante; freqüência de operação superior a 150MHz; *pipeline* de três estágios; *hardware* para realização de multiplicações e divisões; e memória *on-chip* 64Kb. [Ouelette *et al* 2005].

A freqüência e o desempenho do *Microblaze* variam de acordo com a configuração utilizada, a adição de blocos adicionais, tal como uma unidade de ponto flutuante aumenta drasticamente o desempenho do mesmo, como pode ser visto na Figura 3.2. O desempenho utilizando uma unidade de ponto flutuante aumentou seis

vezes em uma codificação JPEG, cinqüenta vezes no cálculo da FFT, chegando a cento e vinte vezes no filtro FIR.

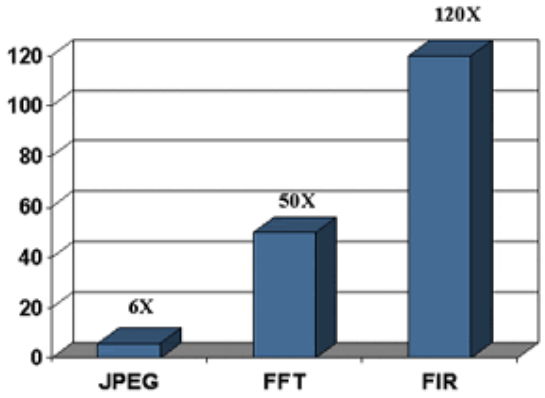

Figura 3.2 – Aumento do desempenho utilizando uma unidade de ponto flutuante.

O tamanho ocupado pelo Microblaze® no FPGA varia dependendo da família de dispositivos na qual o mesmo é utilizado e dos componentes que forem sintetizados. Sua freqüência de operação e desempenho variam de acordo com a configuração utilizada [Xilinx 2009]. Por exemplo, no *Virtex II Pro*®, o Microblaze® pode utilizar 827 LE quando usado em uma freqüência de 150 MHz, ou 1.225 LE a uma freqüência de 170 MHz.

O Microblaze® pode gerenciar a memória interna, 64 Kbyte configuráveis, ou uma memória externa acoplável utilizando ou não um sistema operacional. Existem diversos sistemas operacionais para esse processador, entre eles o µCLinux. Cada Microblaze® possui dezesseis conexões FSL, permitindo a comunicação com até oito Microblazes diferentes. Esses canais de comunicação são unidirecionais de 32 bits, síncronos, do tipo FIFO (*First In First Out*) [Xilinx 2009].

Na literatura observou-se trabalhos [Huerta *et al* 2005, Lim *et al* 2006, Huerta *et al* 2007] utilizando esse *software processor*. Nestes trabalhos, o Microblaze® é mostrado como uma boa plataforma de desenvolvimento, validando as características apresentadas.

3.3 Nios® II

Pode-se dizer que os sistemas que utilizam o Nios® II são equivalentes a sistemas com um µC, pois incluem em um único circuito integrado: uma unidade de processamento (UP), memória e periféricos [Altera Avalon 2009, Plavec *et al* 2005].

O núcleo do Nios® II é baseado na arquitetura RISC (*Reduced Instruction Set Computer)*, ou seja, possui um conjunto de instruções reduzido. Possui uma unidade para realização de operações com ponto flutuante, *pipeline* configurável de até seis estágios, memória interna de até 64Kb e sua programação é realizada utilizando a linguagem de programação C/C++. O núcleo do Nios® II é distribuído em três versões: econômica, básica e rápida. O desenvolvedor deve optar por uma versão adequada à sua aplicação. A Tabela 3.1 [Altera Nios II 2009] mostra diferenças entre as versões.

Tabela 3.1 – Características das versões do *Nios® II*.

Característica	**Versões do Nios® II**		
	Econômica	**Básico**	**Rápido**
Freqüência máxima	200MHz	165MHz	185MHz
Tamanho ocupado no FPGA	700 LE	1400 LE	1800 LE
Pipeline	1 estágio	5 estágios	6 estágios
Endereçamento de memória externa	Até 2Gbytes	Até 2Gbytes	Até 2Gbytes
Memória *on-chip*	-	64Kbytes	64Kbytes
Previsão de desvios	-	Estática	Dinâmica
Multiplicação em *hardware*	-	3 ciclos	1 ciclo

A versão rápida foi desenvolvida para aplicações que necessitam de um alto desempenho. A mesma possui memória *cache* para dados e instruções, que melhoram, por exemplo, o desempenho de aplicações com uma grande quantidade de dados.

A versão básica não possui memória *cache* para dados e o seu desempenho é inferior a versão rápida. Esta versão é cerca de 40% menor do que a versão rápida. O mesmo pode ser usado em aplicações onde alto desempenho não seja um requisito fundamental.

A versão econômica do Nios® II necessita da metade dos blocos lógicos necessários para utilização da básica. Essa possui apenas as funções mínimas para que se possa utilizar o conjunto de instruções existente no Nios® II. Esse núcleo é utilizado em aplicações onde é requerido um controle lógico simples.

Por ser um *software processor*, pode-se adicionar novas funções e interfaces para periféricos ao núcleo do Nios® II, de acordo com a necessidade da aplicação [Altera Nios II 2009]. Por exemplo, interfaces de comunicação, conexões para dispositivos de entrada e saída, e controladores de memórias externas. Desenvolvedores também podem criar seus próprios periféricos e integrá-los ao Nios® II. Por exemplo, pode-se criar um *hardware* específico para implementar uma função que ao ser executada em *software* exige muitos ciclos do processador [Altera Nios II

2009]. Essa modificação melhora o processamento de duas maneiras: a CPU é liberada para executar uma outra função, enquanto o *hardware* desenvolvido processa outras informações em paralelo. A Figura 3.3 mostra um exemplo de arquitetura em que o Nios® II pode ser programado.

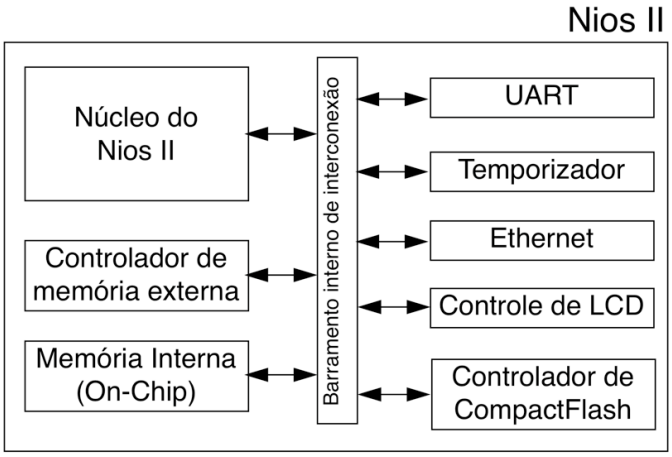

Figura 3.2 – Exemplo de uma arquitetura configurada com o Nios® II.

Para que o Nios® II possa se comunicar com os outros componentes do sistema, é necessária a utilização de um *barramento interno de interconexão*, como na Figura 3.3. Assim foi criado o barramento Avalon, que é mostrado em detalhes na próxima seção.

3.4 Barramento Avalon

O barramento Avalon foi desenvolvido para conectar dispositivos, como os *software processor* Nios® e Nios® II a periféricos em um mesmo sistema programável em *chip* (*System-On-a-programable Chip* – SOPC). As principais metas no desenvolvimento do barramento foram a utilização de um protocolo simples de ser entendido e otimização do uso dos LE que compõem o FPGA [Altera Avalon 2009].

Dispositivos lógicos, que usam esse barramento para comunicar-se com outros na realização de uma tarefa, são chamados de periféricos Avalon. Esses são divididos, podendo ser instanciados como mestre ou escravo, onde o primeiro é capaz de iniciar transferências de dados e o segundo só transfere dados quando é requisitado. O Processador Nios® II é um exemplo de periférico mestre; já a memória é um exemplo de periférico escravo. A Figura 3.4 mostra um exemplo de sistema utilizando o barramento Avalon para realizar as comunicações.

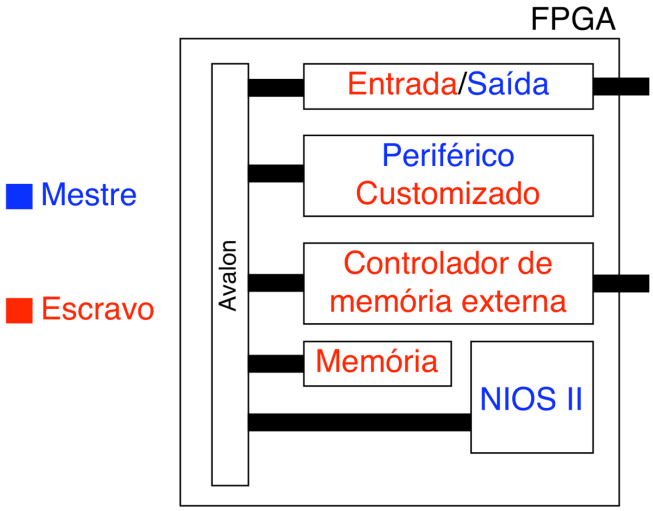

Figura 3.4 – Exemplo de sistema com Barramento Avalon.

O barramento especifica as portas de conexão entre dispositivos mestre e escravo e o tempo para se comunicarem. Toda a lógica interna existente no barramento é construída de forma transparente pelo SOPC *Builder*, um dos *softwares* integrantes do ambiente de desenvolvimento da Altera®.

Diversos periféricos mestres são suportados, podendo realizar transmissões simultâneas desde que não seja para o mesmo escravo [Altera Avalon 2009]. Quando um escravo pode ser acessado por dois ou mais mestres é necessário um árbitro e uma técnica de arbitragem para saber qual dos mestres terá acesso ao escravo.

A Figura 3.5 mostra um outro exemplo de sistema com barramento Avalon. Nesse sistema, diferente do mostrado na Figura 3.4, existe um periférico escravo *memória de dados*, que é acessado por dois periféricos mestres *Nios® II*.

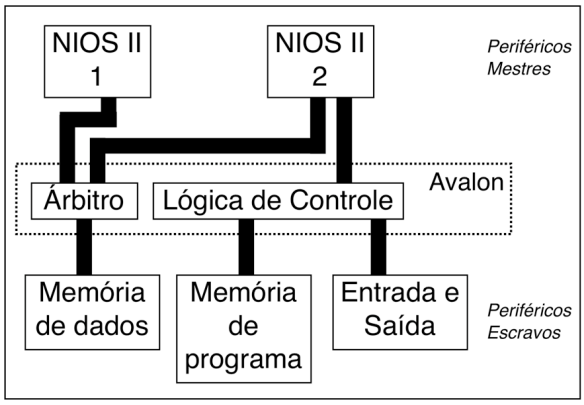

Figura 3.5 – Dois periféricos mestre com acesso ao mesmo periférico escravo.

O árbitro é formado por dois elementos: lógica de requisição e a lógica arbitrária. A primeira avalia os sinais de endereço e controle existentes em cada um dos mestres e gera um sinal de requisição, que alimenta a lógica arbitrária essa é onde o esquema de arbitragem está localizado. A arbitragem é transparente aos periféricos. Cada mestre atua como se fosse único e quando mais de um mestre requisita o mesmo escravo, o árbitro decide qual mestre acessará o escravo.

O esquema de arbitragem utilizado pelo árbitro é o *round-robin*, também conhecido *weighted-round-robin* ou *fairness-based* [Altera Simultaneous 2009]. O esquema de arbitragem é a técnica necessária para que o árbitro possa eleger o mestre que acessará o escravo. O *round-robin* é um dos primeiros e mais simples algoritmos para escalonamento de processos [Tanembaum 1999]. Em seu funcionamento, um intervalo de tempo qualquer, denominado *quantum* é definido, determinando o tempo de cada interrupção. Os processos são armazenados em uma fila circular, como na Figura 3.6, onde cada processo é retirado do processador ao término do *quantum*. Se o processo finalizar em um tempo menor que o *quantum*, o processador é liberado para executar um novo processo, caso o processo não seja finalizado ele volta para a fila.

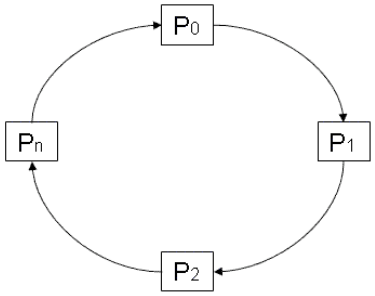

Figura 3.6 – Fila circular.

De forma análoga pode-se considerar os dispositivos mestres como os processos e o barramento Avalon como o processador. Sempre que mais de um mestre tenta acessar o mesmo escravo, o esquema de arbitragem se encarrega de determinar o tempo que cada mestre tem posse do escravo.

Portanto, ao utilizar o barramento Avalon como interface de comunicação entre os periféricos no FPGA, é possível utilizar vários periféricos mestres conseqüentemente vários Nios® II. Assim, é possível o desenvolvimento de sistemas com múltiplos processadores em um único *chip*, conhecidos por MPSoC (*Multiple Processor System on Chip*) que são mostrados na seção seguinte.

3.5 Sistemas com múltiplos processadores em um único chip

De acordo com Jerraya [Jerraya *et al* 2005] um dos maiores desafios nos projetos de MPSoC é o desenvolvimento do *software* dos ambientes com múltiplos processadores. Aplicações e sistemas operacionais devem ter um bom desempenho para que possam cumprir os requisitos necessários em tempo real. A Altera® disponibiliza boas ferramentas para trabalhar com os MPSoC que utilizam o Nios® II como unidade de processamento. Os sistemas que utilizam mais de um núcleo Nios® II podem ser divididos em dois tipos. O primeiro, visto na Figura 3.7(a), é considerado independente, pois nenhum recurso é compartilhado, enquanto o segundo, visto na Figura 3.7(b), possui uma memória compartilhada.

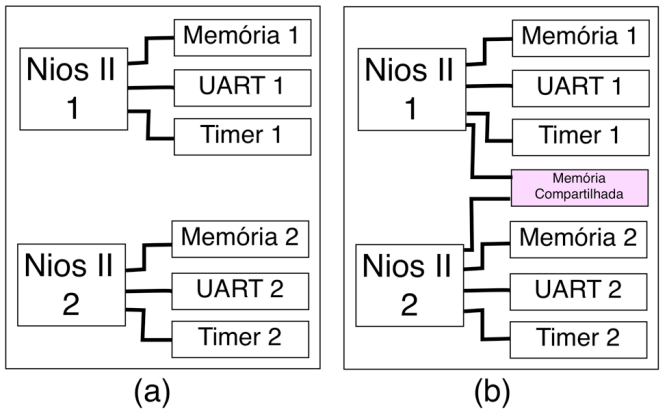

Figura 3.7 – Sistema com múltiplos processadores: (a) independentes; (b) com recursos compartilhados.

Em [Huerta 2007] foi verificado que ao utilizar o processador Microblaze em um FPGA da Xilinx, o fator limitante da quantidade de processadores embarcados é a quantidade de blocos lógicos do FPGA e a quantidade de blocos de memória existente. Assim, enquanto ainda houver blocos lógicos disponíveis no FPGA é possível aumentar a quantidade de processadores no FPGA. Essa afirmação, como foi verificado nesse trabalho, também é válida para os processadores Nios® II embarcados em FPGAs Altera®.

Como visto no capítulo anterior, existem duas formas diferentes para comunicação entres os processadores: a memória compartilhada e a troca de mensagens. Para que a comunicação possa ocorrer entre múltiplos Nios® II é necessário adicionar ao sistema um dos dois periféricos mostrados adiante: o *mutex*

para comunicação através da memória compartilhada ou o *mailbox* para troca de mensagens.

3.5.1 Mutex

É um periférico utilizado para coordenar os acessos de leitura e escrita em um recurso comum. Este possui um protocolo para garantir exclusão mútua ao recurso compartilhado e foi projetado para uso dos processadores Nios® II que utilizam o barramento Avalon [Altera Mutex 2009].

As operações realizadas no *mutex* são atômicas, ou seja, indivisíveis, garantindo em condições normais o término das operações iniciadas, mesmo que o periférico escravo seja requisitado por um outro mestre. Cada mestre possui um registrador com um valor de identificação único (ID), enquanto o *mutex* possui dois registradores: valor e proprietário.

O registrador valor do *mutex* sempre pode ser lido por um mestre. O seu conteúdo é quem garante a disponibilidade do *mutex*, caso seja nulo, seu estado está disponível e o mesmo pode ser adquirido por um mestre. Após a aquisição do *mutex*, como visto na Figura 3.8, o dispositivo escreve o seu identificador no registrador proprietário, e um valor não nulo no registrador valor para que nenhum outro dispositivo acesse o *mutex*. Ao término do processo, o dispositivo é reiniciado e o valor do registrador *valor* volta a ser nulo para que um outro dispositivo possa adquiri-lo.

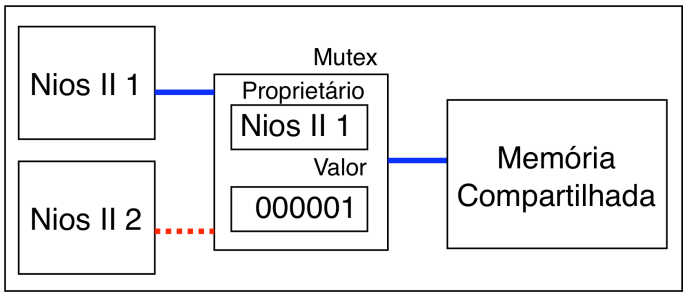

Figura 3.8 – Processador adquirindo mutex para escrever em uma memória compartilhada.

3.5.2 Mailbox

O *mailbox*, ou caixa postal, permite a troca de mensagens entre processadores utilizando o barramento Avalon [Altera Mailbox 2009]. Esse deve ser utilizado em

conjunto com uma memória compartilhada, separada da memória interna dos processadores, para o armazenamento das mensagens. Internamente, o *mailbox* contêm dois *mutexes*, como visto na Figura 3.9, um para leitura e outro para escrita, garantido que apenas um processador modificará o conteúdo de sua memória em um determinado instante.

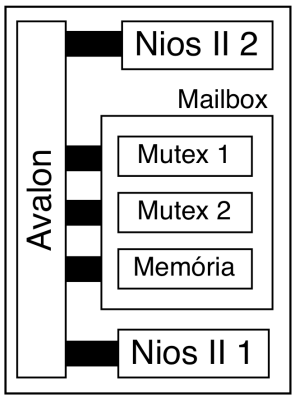

Figura 3.8 – Utilização de *mailbox* para comunicação por troca de mensagens.

As mensagens enviadas pelo *mailbox* são de 32 bits, que são armazenadas em um espaço e endereço pré-definido, denominado pilha, na memória compartilhada. O tamanho desse endereço define o número máximo de mensagens que podem ser armazenadas na fila da pilha. As primeiras mensagens armazenadas são as primeiras a sair, pois a fila é do tipo FIFO – *First In First Out*.

Tanto a escrita como a leitura das mensagens existentes no *mailbox* não podem ser realizadas simultaneamente por mais de um processador. Quando necessário, o processador pode enviar mensagens sucessivas até que a fila do *mailbox* esteja cheia. A leitura também pode ser realizada sucessivamente até que a fila do *mailbox* esteja vazia, pois quando uma mensagem é lida pelo processador é retirada da fila.

Os parâmetros do *mailbox* são ajustados pelo programa fornecido pela Altera®, o SOPC *Builder*. Nesse *software* são definidos, quais processadores vão compartilhar o *mailbox*, qual módulo de memória será utilizado para o armazenamento das mensagens e qual será o tamanho da fila. A janela de ajuste dos parâmetros é vista na Figura 3.10.

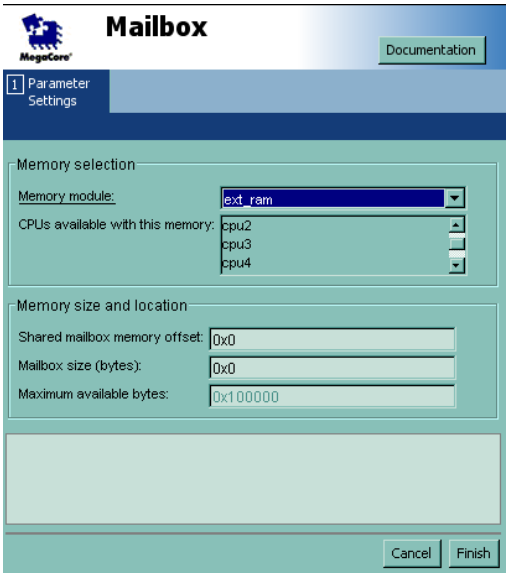

Figura 3.10 – Janela de ajustes do *mailbox*.

Como visto no segundo capítulo é necessário realizar uma análise para saber qual o desempenho de um determinado sistema. Conhecendo o desempenho, pode-se realizar comparações entre uma aplicação executada em um sistema seqüencial e em um sistema paralelo. Neste trabalho, a análise foi feita realizando a medição do tempo de execução dos algoritmos desenvolvidos. Essa medição pode ser realizada de duas formas distintas: utilizando uma biblioteca de *software* ou um periférico específico [Tong *et al* 2005].

3.6 Medição do tempo de processamento no FPGA

Existem duas maneiras de mensurar o tempo de execução em um sistema, cuja unidade de processamento é o Nios® II. Pode-se utilizar uma biblioteca implementada em *software* chamada de *time_stamp(),* ou um periférico chamado de *performance counter*, esses são mostrados a seguir. Pesquisadores testaram em [Tong *et al* 2007] as medidas desempenho existentes e decidiu utilizar o *performance counter* por apresentar um resultado mais preciso quando comparado as outras medidas.

A biblioteca *time_stamp()* é utilizada em conjunto com um temporizador de alta resolução. Esta é uma biblioteca, que quando chamada retorna a quantidade de ciclos executados até o momento. O procedimento *time_stamp* apresenta o pseudo-código de como utilizar essa biblioteca.

```
Procedimento time_stamp
01:    Define S1, S2, S3
02:    Inicializa time_stamp
03:    S1 ← time_stamp()
04:            Faça: ...
05:    S2 ← time_stamp()
06:            Faça: ...
07:    S3 ← time_stamp()
08:    Mostra Tempo1 ← S2 – S1
09:    Mostra Tempo2 ← S3 – S2
10: Fim time_stamp
```

O *performance counter* é um bloco lógico composto por vários contadores para mensurar o tempo de execução de trechos selecionados do código. Cada trecho selecionado é chamado de seção e pode mensurar o tempo de até quatro seções. O procedimento *performance_counter* mostra um exemplo de pseudo-código, com as instruções necessárias para utilização do *performance counter*. Para cada trecho de código selecionado, o *performance counter* possui dois contadores, um de sessenta e quatro bits para contar o tempo executado no trecho, e um de trinta e dois bits, para contar quantas vezes o trecho foi executado.

```
Procedimento performance_counter
01:    Define Secao1, Secao2, Secao3
02:    Reset Performance_Counter
03:    Inicio Medicao
04:            Inicio Secao1
05:                    Faça: ...
06:            Fim Secao1
07:            Inicio Secao2
08:                    Faça: ...
09:            Fim Secao2
10:            Inicio Secao3
11:                    Faça: ...
12:            Fim Secao3
13:    Fim Medicao
14:    Mostra relatorio
15: Fim performance_counter
```

De acordo com [Tong *et al* 2007] a utilização do *performance counter* é o único método utilizado pelo Nios® II para medir tempo com praticamente nenhum aumento no tempo de execução do código. A desvantagem é que ele aumenta o número de blocos lógicos necessários do projeto no FPGA. Os resultados obtidos pelo *performance counter* são mostrados como vistos na Tabela 3.2, esse retorna o tempo

necessário para execução de cada função, o número de ciclos necessários e quantas vezes a função foi executada.

Tabela 3.2 – Exemplo de como o *performance counter* apresenta os resultados.

```
Total Time: 0.102086 seconds (5104278 clock-cycles)
+----------------------+-----+-----------+---------------+-----------+
| Section              |  %  | Time (sec)|  Time (clocks)|Occurrences|
+----------------------+-----+-----------+---------------+-----------+
|Cálculo Especialistas| 67.4|    0.06882|        3441227|          1|
+----------------------+-----+-----------+---------------+-----------+
|Cálculo Rede Passagem| 26.8|    0.02738|        1369163|          1|
+----------------------+-----+-----------+---------------+-----------+
|Cálculo da Rede Final|  5.7|    0.00588|         293858|          1|
+----------------------+-----+-----------+---------------+-----------+
```

As duas formas de medição do desempenho foram testadas e também foi observado nos experimentos uma precisão maior na utilização do *performance counter*. O conhecimento dessas medições é necessário para mensurar o tempo de execução necessário dos algoritmos testados na plataforma utilizada.

3.7 Conclusão

Este capítulo mostrou uma breve explicação do que é um FPGA e as características que o tornam uma boa escolha para ser utilizado em sistemas embarcados. Foi mostrado como, a partir das idéias vistas no segundo capítulo, criar sistemas para processamento paralelo utilizando um FPGA.

Também foi apresentado o processador Nios® II, que foi escolhido como unidade de processamento da arquitetura paralela utilizada. Também foram apresentados periféricos e ferramentas de apoio necessários para que um sistema com múltiplos processadores em um FPGA Altera® possa ser implementado.

A utilização do Nios® II como unidade de processamento tem sido bastante explorada pela literatura em diversas aplicações, apresentando-se como uma boa escolha para desenvolvimento de projetos, quando comparada a plataformas semelhantes. Assim, o Nios® II foi escolhido, para que arquiteturas de redes neurais em paralelo fossem implementadas. As redes neurais artificiais são apresentadas no próximo capítulo.

Capítulo 4

Redes Neurais Embarcadas

Esse capítulo traz uma breve introdução sobre sistemas inteligentes. Nesse são apresentados conceitos de aprendizagem de máquina, redes neurais artificiais (RNA), máquinas de comitê e máquinas de vetor de suporte (SVM – *Support Vector Machine*). O capítulo finaliza com comentários sobre algumas redes neurais implementadas em dispositivos embarcados, sua importância e como podem ser aplicadas.

O objetivo principal dos sistemas inteligentes é o desenvolvimento de algoritmos que utilizem máquinas para realizar tarefas cognitivas, nas quais os humanos atualmente são melhores. Um sistema inteligente deve ser capaz de realizar três tarefas básicas [Haykin 2001]: armazenar, aplicar e e adquirir novo conhecimento através da experiência, em outras palavras, aprender. Os paradigmas de aprendizagem são vistos na próxima seção.

4.1 Paradigmas de aprendizagem de máquina

A Aprendizagem de Máquina trata de um campo do saber científico preocupado em projetar e criar modelos computacionais capazes de aprender e/ou criar conhecimento e habilidades encontradas no ser humano e na natureza [Magalhães 2007]. Pode-se dizer que construir sistemas inteligentes, consiste em desenvolver algoritmos computacionais que façam com que as máquinas aprendam e melhorem o seu desempenho com o passar do tempo e de forma automática.

Existe uma variedade de algoritmos criados para a tarefa de aprendizagem de máquina, dentre estes pode-se citar: Aprendizagem por Reforço, Máquinas de Vetor de Suporte e Redes Neurais Artificiais.

A interação desses algoritmos com o ambiente pode ser classificada em dois modelos: aprendizagem sem professor, ou auto-supervisionada e aprendizagem com professor, também conhecida como aprendizagem supervisionada [Haykin 2001].

4.1.1 Aprendizagem auto-supervisionada

Na aprendizagem auto-supervisionada não existe um tutor para supervisionar o processo de aprendizagem, ou seja, não existem exemplos da função a ser aprendida. Esse paradigma ainda pode ser divido em dois tipos: aprendizagem auto-organizada e aprendizagem por reforço.

1. Aprendizagem auto-organizada. Não existe crítico ou professor externo para supervisionar a tarefa de aprendizagem, como pode ser visto na Figura 4.1. Para realizar a aprendizagem auto-organizada, pode-se utilizar diferentes algoritmos, por exemplo, aprendizagem competitiva ou os mapas auto-organizáveis. Esses algoritmos são detalhados em [Haykin 2001].

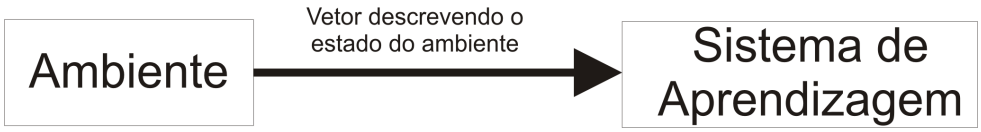

Figura 4.1. – Diagrama de blocos da aprendizagem auto-organizada.

2. Aprendizagem por reforço. Neste tipo de aprendizagem, o aprendizado é realizado através da interação continua com o ambiente. Aprende-se o que fazer, através de situações e ações, maximizando-se o sinal de retorno, a recompensa.

Na aprendizagem por reforço não é dito para o agente quais ações realizar, como nos outros métodos de aprendizagem, no qual é necessário um professor. Ao invés disso o agente tenta descobrir que ações ele deve tomar para receber uma recompensa melhor. Em alguns casos, as ações tomadas não irão afetar apenas a próxima ação, mas também as próximas ações. A Figura 4.2 mostra o diagrama de blocos da aprendizagem por reforço.

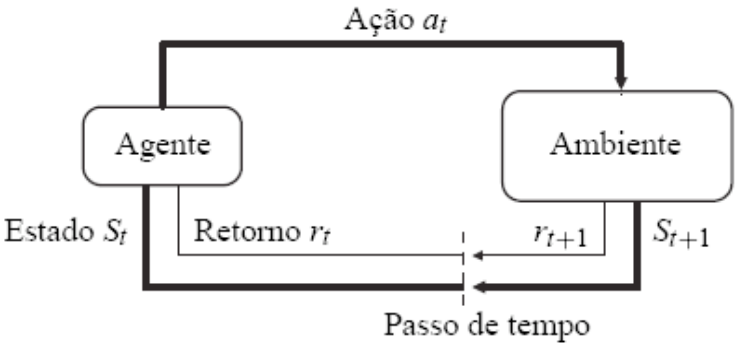

Figura 4.2 – Diagrama de blocos da aprendizagem por reforço.

A aprendizagem por reforço não é definida caracterizando os métodos de aprendizagem, e sim caracterizando o problema da aprendizagem [Sutton 1998]. Qualquer método que possa ser utilizado para resolver o problema proposto é considerado como um método de aprendizagem por reforço.

4.1.2 Aprendizagem supervisionada

Na aprendizagem supervisionada, ou com professor, o professor tem o conhecimento sobre o ambiente. As RNA, por exemplo, são algoritmos que podem utilizar esse paradigma. As RNA serão descritas na próxima seção. O diagrama de blocos da aprendizagem com professor é visto na Figura 4.3. Nesse caso é o professor quem tem o conhecimento sobre o ambiente. Este conhecimento é representado por um conjunto de exemplos de entrada e saída.

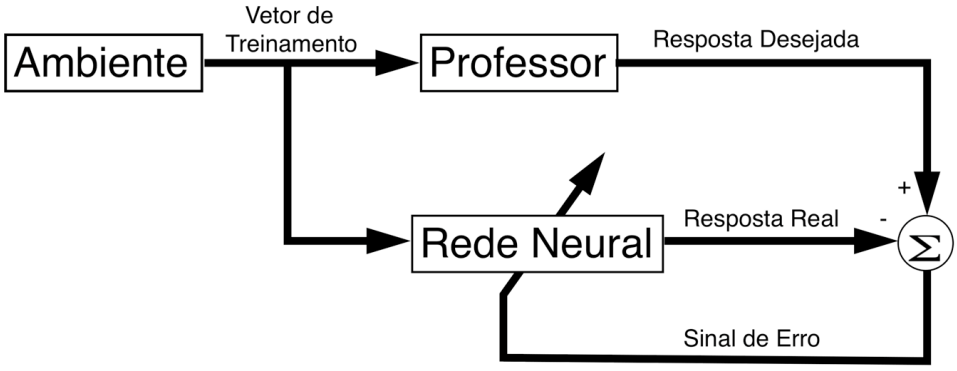

Figura 4.3 – Diagrama de blocos da aprendizagem supervisionada.

Um vetor de treinamento retirado do ambiente é a entrada aplicada ao professor e a RNA. Devido ao conhecimento prévio, o professor é capaz de fornecer a saída desejada do sistema para aquela entrada. A resposta do professor representa a saída ótima que deve ser aprendida pela rede neural. Os parâmetros da rede são ajustados a cada iteração para minimizar o sinal do erro.

O sinal de erro é definido como a diferença entre a resposta desejada e a resposta da rede. Esse ajuste é realizado passo a passo, iterativamente, com o objetivo de fazer a rede neural emular o professor. Assim, o conhecimento do professor é transferido para a rede neural, através do treinamento. Quando a rede neural é capaz de imitar o comportamento do professor, esse é descartado e a rede neural assume o

comando do sistema. Essa forma de aprendizagem é denominada de aprendizagem por correção de erro que é detalhada em [Haykin 2001].

4.2 Redes neurais artificiais

A rede neural artificial (RNA) tem sido motivada, desde o início, pelo reconhecimento de que o cérebro humano processa informações de uma forma inteiramente diferente do computador digital convencional. O cérebro pode ser considerado um computador altamente complexo, não linear e paralelo.

Uma rede neural artificial é uma unidade de processamento maciçamente paralela e distribuída, na qual é realizado um processamento simples, e tem a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se assemelha ao cérebro em dois aspectos:

1. O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem.
2. As conexões entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido.

A RNA possui a habilidade de aprender e, portanto, de generalizar. A generalização se refere ao fato que essa produz saídas adequadas para entradas que não estavam presentes durante o treinamento.

A unidade de processamento de informação que é fundamental para a operação de uma rede neural é denominada de neurônio. A Figura 4.4, mostra o modelo de um neurônio de Mc-Culloch-Pitts [Haykin 2001].

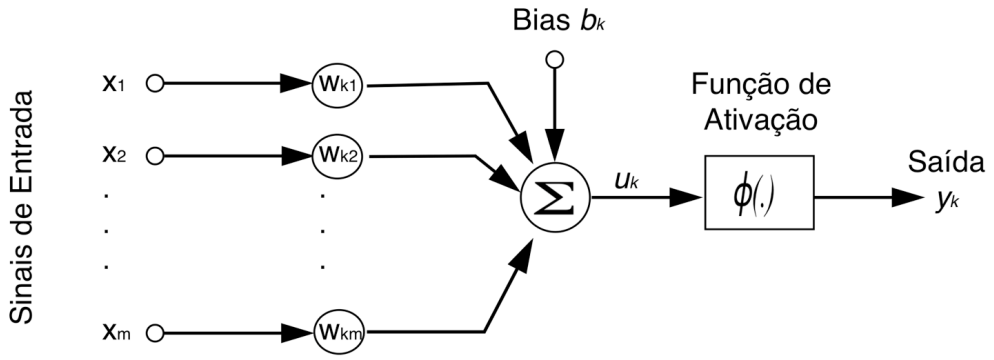

Figura 4.4 – Modelo de um neurônio não-linear.

O neurônio pode ser dividido em três partes básicas:

1. Um conjunto de pesos w_{ki}, $i = 1, 2... n$.
2. Um somador dos sinais de entrada, ponderados pelas respectivas sinapses do neurônio.
3. Uma função de ativação fornecedora neurônio $\phi(.)$.

O modelo mostrado, na Figura 4.4, inclui também o bias aplicado externamente, representado por b_k, no qual tem o efeito de aumentar, se seu valor for positivo, ou diminuir, caso o valor seja negativo, a entrada da função de ativação.

Em termos matemáticos, pode-se descrever a relação entre a entrada e a saída de um neurônio k através das seguintes equações:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{4.1}$$

$$y_k = \phi(u_k + b_k) \tag{4.2}$$

Onde x_1, x_2, ..., x_m são sinais de entrada; w_{k1}, w_{k2}, ..., w_{km} são os pesos sinápticos do neurônio k; u_k é a saída do combinador linear devido aos sinais de entrada; b_k é o bias; ϕ a função de ativação e y_k é o sinal de saída do neurônio.

A função de ativação define a saída de um neurônio em função da combinação linear dos pesos aplicados as entradas. Os tipos básicos de funções de ativação são: limiar, linear por partes, sigmóide e tangente sigmóide.

A propriedade mais importante para uma rede neural é a sua habilidade de aprender, a partir do ambiente e assim melhorar o desempenho. A melhoria do desempenho ocorre com o tempo, de acordo com alguma medida pré-estabelecida. Uma rede neural aprende acerca do seu ambiente através de um processo iterativo de ajustes aplicados a seus pesos sinápticos. A rede se torna mais instruída sobre o seu ambiente após cada iteração do processo de aprendizagem.

O conceito de aprendizado no contexto das RNA foi definido em [Mendel & McLaren 1970] como sendo: um processo nos quais os parâmetros livres (pesos sinápticos), da rede neural, são modificados devido à um processo de estimulação no ambiente que a rede está inserida. Essa definição implica no seguinte algoritmo:

1. A rede neural é estimulada por um ambiente;
2. A rede neural sofre modificações nos seus parâmetros livres;
3. A rede neural responde de uma maneira nova ao ambiente.

Uma rede neural pode possuir *x* entradas, *n* neurônios, *c* camadas e *y saídas* organizados de diferentes modos. A forma como uma rede neural está organizada é chamada de arquitetura da rede. Exemplos de arquiteturas de redes são: perceptron com camada única, perceptron com múltiplas camadas (MLP – *Multi-Layer Perceptrons*) e redes de base radial. A Figura 4.5 mostra um exemplo de arquitetura de RNA do tipo MLP.

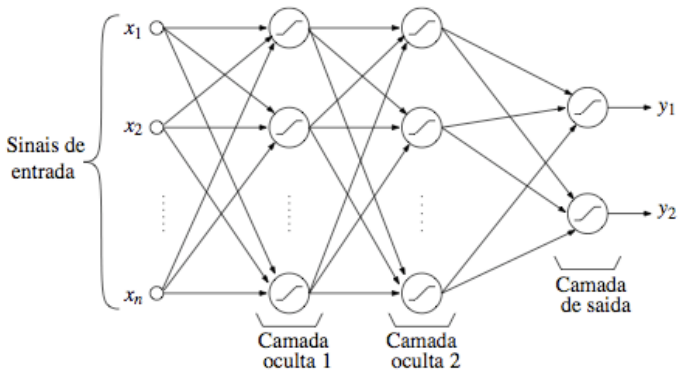

Figura 4.5 – Exemplo de arquitetura de uma RNA do tipo MLP.

As MLP são um dos tipos mais difundidos de arquitetura de RNA. Os dados iniciam da camada mais a esquerda, denominada camada de entrada, atravessam cada uma das camadas seguintes, conhecidas como camadas ocultas, e seu fluxo termina na última camada à direita, a camada de saída. Em cada passagem por entre as camadas, a entrada é ponderada por um peso sináptico e acumulada em conjunto com o bias formando o campo local induzido que é então utilizado pela função de ativação em cada neurônio das várias camadas da rede.

As MLP costumam interagir com o ambiente na fase de treinamento, como visto na Figura 4.3, ou seja utilizando uma aprendizagem supervisionada. Um algoritmo comumente utilizado e já bem definido na literatura é o da retropropagação do erro (*backpropagation*). Esse consiste em ciclos compostos por: apresentações aleatórias de vetores de entrada, obtenção da diferença entre a resposta fornecida pela rede e a resposta desejada. Essa diferença é utilizada como argumento para realização das alterações nos pesos sinápticos e assim treinar a rede.

Além dos parâmetros já comentados, pode-se agrupar *n* redes neurais como uma única máquina de aprendizagem. Uma arquitetura, no qual *n* sub-máquinas de aprendizagem são agrupadas para formar uma única máquina é chamada de máquinas de comitê, que são descritas a seguir.

4.3 Máquinas de comitê

Máquinas de comitê são estruturas que fazem uso de um conceito comumente utilizado na computação paralela; dividir para conquistar. De acordo com essa abordagem, uma tarefa computacional complexa é resolvida dividindo-a em um número de tarefas computacionais simples, que são resolvidas separadamente, e então essas respostas individuais são reagrupadas para solucionar o problema.

Assim, as máquinas de comitê podem ser definidas como um conjunto de máquinas de aprendizagem (especialistas) em que as decisões são combinadas para obter uma resposta global. A idéia é unir o conhecimento adquirido pelos especialistas para chegar a uma decisão global que é supostamente superior à encontrada por qualquer uma das estruturas isoladamente.

A simplicidade computacional é alcançada distribuindo-se tarefas de aprendizagem entre uma quantidade determinada de especialistas. Considerando cada especialista como um processo separado é possível constatar que esses podem ser implementados em unidades de processamento diferentes. As máquinas de comitê são divididas em duas categorias [Haykin 2001]:

1. Estruturas estáticas: nesse caso as respostas de vários especialistas são combinadas por meio de um mecanismo que não envolve o sinal de entrada. As estruturas estáticas ainda possuem duas classificações: média de *ensemble* onde as saídas de diferentes especialistas são combinadas linearmente para produzir uma saída global, enquanto o reforço utiliza algoritmos de baixa precisão que, em conjunto, alcançam uma alta precisão.
2. Estruturas dinâmicas: nessas, o sinal de entrada é utilizado para obtenção de uma resposta e também pode ser classificado em dois tipos. No primeiro denominado de mistura de especialistas, as respostas dos especialistas são combinadas por meio de uma rede de passagem, que tem a função de ponderar as respostas dos especialistas para gerar a solução final. Enquanto no segundo,

mistura hierárquica de especialistas, as respostas dos módulos são combinadas em diversas redes de passagem, organizadas em uma forma hierárquica.

A Figura 4.6 mostra uma máquina de comitê estática do tipo média de *ensemble*. As saídas dos especialistas (y_1, y_2 ... y_k) são combinadas linearmente para gerar a saída global y. A entrada x recebida por cada um dos especialistas é a mesma, entretanto cada um deles pode ser treinado independentemente.

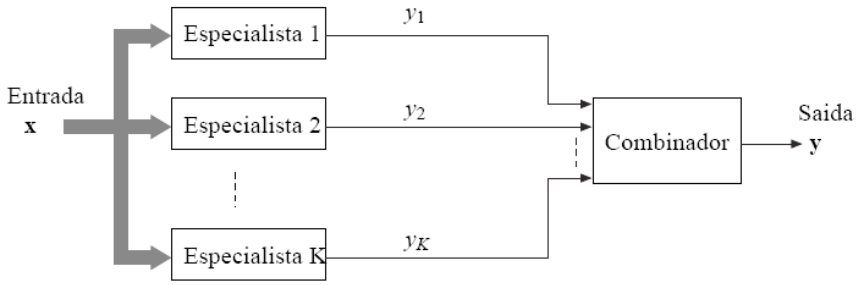

Figura 4.6 – Máquina de comitê estática do tipo media em *ensemble*.

A vantagem dessa arquitetura em comparação a uma rede neural única é que devido à sua organização, essa possui um número de parâmetros ajustáveis reduzido. Cada especialista pode se especializar em um sub-conjunto da entradas, e portanto necessita menos parâmetros, enquanto uma única rede neural tem que aprender todo o conjunto de entrada. Como os especialistas podem ser treinados em paralelo o tempo de treinamento também é menor.

A Figura 4.7 apresenta o diagrama de blocos de uma rede dinâmica, ou modular. Nesta, existem k módulos, cada um representando um especialista, e uma rede que faz a integração entre os módulos, denominada rede de passagem. y_1, y_2 ... y_k são as saídas dos especialistas e g_1, g_2 ... g_k as ponderações da rede de passagem para cada especialista.

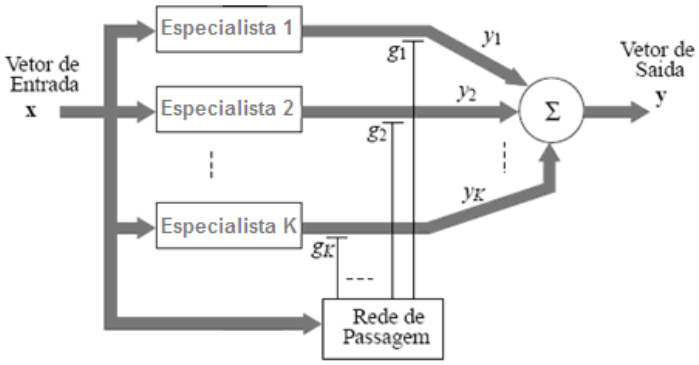

Figura 4.7 – Diagrama de blocos de uma máquina de comitê dinâmica.

Segundo observado em [Haykin 2001], uma RNA pode ser chamada de modular se a computação realizada pela rede pode ser decomposta em dois ou mais módulos, que operam sobre entradas distintas sem comunicação entre eles. As saídas dos módulos são mediadas por uma unidade que não pode alimentar a informação de volta aos módulos. Essa unidade decide como as saídas dos módulos devem ser combinadas para gerar a saída global, e determina os padrões que os módulos devem aprender.

Uma vantagem das redes modulares, em relação as redes mais simples, como a MLP, é que a aprendizagem é mais rápida em problemas no qual existe uma decomposição natural dos dados em funções mais simples. Isto se deve ao fato da capacidade que a rede modular tem em dividir o espaço de entrada, enquanto na MLP o conjunto todo deve ser aprendido.

4.4 Máquinas de vetores de suporte

As Máquinas de Vetores de Suporte (SVM – *Support Vector Machines*) constituem uma técnica de aprendizagem fundamentada na teoria de aprendizado estatístico. Essa estratégia foi introduzida por Vapnik em [Vapnik 1992]. Na literatura são encontradas aplicações em diversos domínios com resultados superiores aos obtidos por outro algoritmo de aprendizado, como as RNA [Haykin 2001].

As redes perceptron, que são RNAs com uma única camada de neurônios podem atuar, por exemplo, como classificadores lineares. Em sua concepção não é levado em consideração a otimização da superfície de separação. Em uma classificação como a da Figura 4.8(a). Uma rede perceptron apresentaria como resposta o hiperplano h_2, h_3, ou h_4. Já, as MLP são mais expressivas, podendo representar funções não-lineares gerais, mas são difíceis de treinar, devido à abundância de mínimos locais e a elevada quantidade de pesos. As SVM, foram propostas procurando solucionar esses problemas utilizando um algoritmo de treinamento eficiente para representar funções não-lineares complexas.

Considerando as Figuras 4.8(b) e (c) é possível descrever o funcionamento das SVMs da seguinte forma: dado um conjunto de pontos de entrada pertencente a duas classes, uma SVM determina qual o melhor hiperplano que separa esses pontos. O hiperplano gerado pela SVM é determinado por um subconjunto dos pontos das duas classes, chamado vetores de suporte.

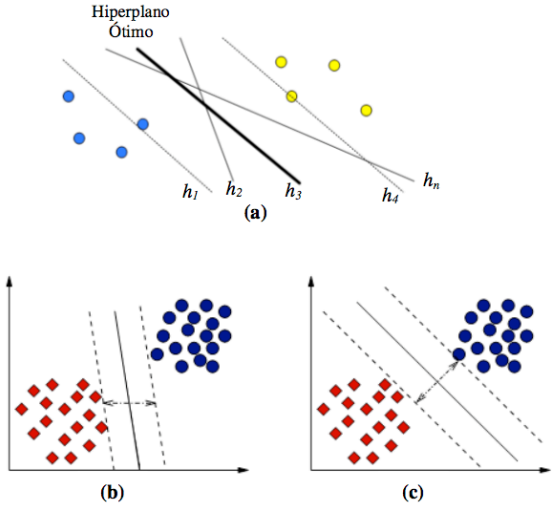

Figura 4.8 – Classificação entre duas classes.

Entre as características que justificam a utilização das SVMs pode-se citar:

- Boa capacidade de generalização;
- Robustez diante de projetos de alta dimensionalidade;
- Capacidade de lidar com dados ruidosos;
- Uma base teórica bem estabelecida na matemática e estatística.

As SVMs também apresentam uma característica atrativa, a existência de um único mínimo global, facilitando a classificação de padrões próximos. Essas lidam com problemas não lineares mapeando o conjunto de treinamento de seu espaço original, referenciado como espaço de entradas, para um novo espaço de maior dimensão, denominado espaço de características, visto na Figura 4.9(c).

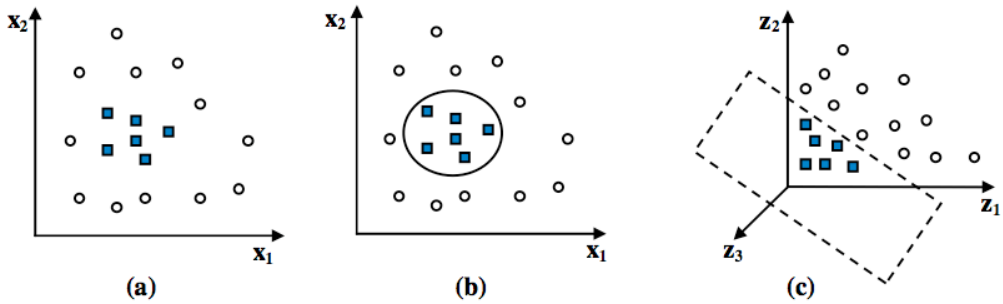

Figura 4.9 – (a) Amostras não lineares, (b) Classificação não linear no espaço de entradas, (c) Classificação linear no espaço de características.

As SVM podem ser consideradas classificadores robustos e de baixo processamento [Haykin 2001]. Esses classificadores fazem uso de funções *kernel* para padrões não linearmente separáveis, tornando o algoritmo eficiente, pois permite a construção de simples hiperplanos em um espaço de características. Um *kernel* é uma função que recebe dois pontos do espaço de entradas e calcula o produto escalar desses dados no espaço de características, esse é calculado de acordo com a equação 4.3.

$$k[x_i, x_j] = \phi(x_i).\phi(x_j) \tag{4.3}$$

Uma vez que o mapeamento das SVM é realizado por uma função *kernel*, e não diretamente por $\phi(x)$, nem sempre é possível saber exatamente qual mapeamento é efetivamente realizado, pois as funções de *kernel* realizam um mapeamento implícito dos dados. Assim, nas SVM os exemplos de treinamento nunca aparecem isolados, mas sempre em forma de um produto interno, que pode ser substituído por uma função de *kernel*. As funções, mostradas na Tabela 4.1, Polinomial, RBF (*Radial Basis Function*) e MLP são exemplos de *kernel* que podem ser utilizados.

Tabela 4.1 – Resumo dos *kerneis* que podem ser utilizados [Haykin, 2001, p. 366]

Tipos de máquina de vetor de suporte	Núcleo do produto interno	Comentários
Máquina de aprendizagem polinomial	$(x_i.x_j + 1)^P$	A potencia *p* é especificada *a priori* pelo usuário
Rede de Função de Base Radial	$e^{\frac{-\lVert x_i - x_j \rVert^2}{2\sigma^2}}$	A largura σ^2, comum a todos os núcleos, é especificada *a priori* ao usuário
Perceptron de duas camadas	$\tanh\left(kx_1.x_j - \delta\right)$	O teorema de Mercer é satisfeito apenas para alguns valores de β_0 e β_1

Independente de como uma máquina de vetor de suporte é implementada, ela difere da abordagem de projeto de uma MLP de uma forma fundamental. Na MLP, a complexidade do modelo é controlada mantendo-se o número de características, neurônios, pequeno. Por outro lado, a máquina de vetor de suporte oferece uma solução para o projeto de uma máquina de aprendizagem controlando a complexidade do modelo independentemente da dimensionalidade. Maiores detalhes sobre esse método pode ser encontrando em [Haykin 2001].

4.5 Redes neurais artificiais implementadas em sistemas embarcados

Redes neurais artificiais podem ser implementadas usando sistemas embarcados analógicos ou digitais. As implementações digitais são mais populares devido a vantagens como exatidão, repetibilidade e baixa sensibilidade ao ruído, dentre outras.

As implementações digitais podem ser divididas quanto a tecnologia: FPGA, DSP, e ASIC (*Application-Specific Integrated Circuit*). Implementações em DSP são seqüenciais e não tiram proveito do paralelismo das redes neurais, ASIC não podem ser reconfiguradas pelo usuário. Portanto o FPGA torna-se uma boa opção pois oferece um grau de paralelismo e podem ser reconfiguradas pelo usuário [Muthuramalingam *et al* 2008].

Existem vários trabalhos na literatura utilizando algum tipo de rede neural como ferramenta, mas poucos que utilizam alguma forma de paralelismo na sua implementação ou são implementados em FPGA. Os trabalhos comentados a seguir, mostram um pouco do que já tem sido feito nessa área.

No trabalho [Muthuramalingam 2008] é comentado que ainda é um desafio o desenvolvimento de redes neurais complexas, estritamente em *hardware* no FPGA, devido à quantidade de neurônios necessários. O desempenho da rede neural está relacionada com a eficiência na concepção de um único neurônio, e quanto maior for a quantidade de neurônios maior é o numero de multiplicações realizadas. Assim o custo de implementação torna-se elevado, já que é necessário um *hardware* mais complexo. Entretanto, os autores também citam três importantes características que os fazem acreditar que as redes neurais e os FPGA podem trabalhar juntos: paralelismo, modularidade e adaptação dinâmica.

Em [Taright & Hubin 1998] foi implementado um neurônio e mostrado a possibilidade de implementação de uma rede *perceptron* de múltiplas camadas em um FPGA. A rede neural foi utilizada em um nariz eletrônico que classificava diversos níveis de poluição do ar em tempo real. O processo pode ser dividido em duas etapas: o treinamento e o reconhecimento, sendo os mesmos distintos e seqüenciais. A fase de treinamento foi realizada em um computador, onde foram calculados todos os parâmetros da rede, tais como: pesos sinápticos, número de entradas e saídas e a quantidade de camadas ocultas. A segunda fase foi realizada no FPGA levando em

consideração as suas restrições. Os autores limitam-se a falar que foram implementados poucos neurônios, mas não dizem o número exato. O problema dessa aplicação, que por ser implementada fisicamente e descrita em HDL, é a necessidade de se modificar o seu código sempre que se deseja mudar a arquitetura da rede neural.

Em [Wei *et al* 2004] é comentado que os processadores de uso geral atuais possuem processamento suficiente para detecção de fase em tempo real utilizando máquinas de vetores de suporte. Entretanto, uma implementação em sistema embarcado é de grande interesse e tem sido pouco explorado. Em seu trabalho [Wei 2004] mostra uma implementação preliminar de uma SVM em FPGA. Alguns experimentos foram simulados em computador com bons resultados. Entretanto, ao realizar a implementação em *hardware*, alguns fatores, como o tamanho da imagem, tiveram que ser alterados devido as limitações do mesmo.

Uma implementação de um controlador neural proporcional, integrativo e derivativo (PID) em um FPGA, utilizando apenas um Nios® II é apresentada em [Wang *et al* 2008]. A vantagem de utilizar essa ferramenta é que mesmo trabalhando em uma baixa freqüência, esse conseguiu um desempenho melhor que os sensores com controladores PID tradicionais. Já em [Cao *et al* 2008] é mostrado a implementação de uma rede neural RBF implementada no Nios® II em conjunto com uma técnica de modulação em largura de pulso (PWM – *Pulse Width Modulation*). Enquanto em [Kwon *et al* 2007] é utilizando controlador neural implementado em Nios® II, que pode ser citado foi o realizado em [Kwon *et al* 2007]. Pode-se concluir que o Nios® II é uma boa plataforma em diversas áreas, incluindo a de controle neural.

A arquitetura implementada e descrita em [Muthuramalingam *et al* 2008] é comentada a seguir para exemplificar parte das dificuldades existentes na concepção de redes neurais em FPGA.

Os autores utilizaram um FPGA da Xilinx para a implementação de uma rede neural com 8 bits de precisão e 5 camadas, sendo a quantidade de neurônios por camadas a seguinte: 1 – 6 – 6 – 6 – 3.

Redesenhando a Figura 4.4, o neurônio pode ser mostrado como na Figura 4.10. Para a implementação de um neurônio como este são necessários blocos de somadores e multiplicadores para a ponderação dos pesos w nas entradas p e uma lógica complexa para realizar o cálculo da função de ativação não linear $f(x)$.

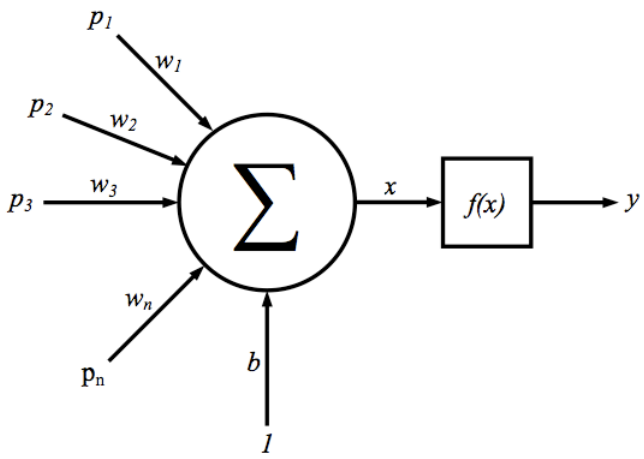

Figura 4.10 – Representação individual de um neurônio [Muthuramalingam 2008].

A implementação de multiplicadores ponto flutuante com sinais e a lógica para o cálculo da função não linear da rede utiliza uma grande quantidade de recursos. A realização de computação em paralelo de *n* neurônios também requer um grande número de recursos, tornando a sua implementação de alto custo. Para reduzir custos, os cálculos dos neurônios foram realizados de uma forma seqüencial, o que diminuí a velocidade da computação.

A decisão de quantos bits serão utilizados é uma escolha importante, já que alta precisão significa menos erros de quantização. Por outro lado com menos bits é possível desenvolver um *hardware* mais simples, ocupando menos espaço e consumindo menos energia. A estrutura do neurônio foi dividida em dois blocos, o primeiro calcula a expressão 4.4 e o segundo uma das expressões: 4.5, 4.6 ou 4.7.

$$x = \sum_{i=1}^{n} p_i w_i + b \tag{4.4}$$

$$f(x) = x \tag{4.5}$$

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.6}$$

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.7}$$

A expressão 4.5 é a função de ativação linear, a 4.6 a log sigmóide e a 4.7 a tan sigmóide. O primeiro bloco possui somadores, subtratores, multiplicadores e uma lógica de controle, já o segundo possui uma lógica complexa no qual não foi apresentada em detalhes. O tamanho do espaço requerido para 1 neurônio com 3 entradas são mostrados na Tabela 4.2.

Tabela 4.2 – Espaço requerido por um neurônio com três camadas.

Função de ativação	Espaço em *slices* necessário		
	Bloco 1	Bloco 2	Total
Linear	258	-	258
Log Sigmóide	258	705	963
Tan Sigmóide	258	811	1069

Os autores utilizaram um FPGA, Xilinx para desenvolver o projeto, portanto o espaço requerido foi mensurado em número de *slices*. De acordo com [Altera Apex 2009] um *slice* equivale à aproximadamente dois LE dos FPGA Altera®. Uma importante contribuição dos autores foi substituir a lógica existente no bloco dois por uma tabela. A utilização de tabelas aumenta a velocidade de operação, mas necessita de muita memória para uma boa precisão.

Com a utilização de uma tabela, ao invés de realizar o cálculo da função não linear, os autores obtêm o resultado correspondente a função na tabela, que é composta por valores pré-calculados. Dessa forma, foi possível diminuir o espaço requerido, como mostrado na Tabela 4.3, na concepção de um neurônio.

Tabela 4.3 – Comparação do espaço requerido entre implementações distintas de neurônio.

Função de ativação	Espaço em *slices* necessário		
	Tabela	Computação	Redução
Log Sigmóide	281	953	79,33%
Tan Sigmóide	282	1096	83,22%

Com essa modificação foi possível realizar uma economia de 79,33% dos recursos do FPGA para a função *Log Sigmóide* e 83,22% para função *Tan Sigmóide*.

A medida que o número de neurônios cresce, a quantidade de recursos necessário no *hardware* também aumenta. Com base nos experimentos realizados os autores definiram duas expressões para prever a quantidade de recursos necessários na implementação física de uma rede neural qualquer. A expressão 4.8 calcula o espaço necessário para a rede neural no qual é feito o cálculo da função de ativação, e a expressão 4.9 para a rede neural que utiliza a tabela com valores pré-calculados.

$$S \approx \sum_{i=1}^{n} a_{1i} S^{i}(250 + 6S^{i-1}) + \sum_{i=1}^{n} a_{2i} S^{i}(950 + 6S^{i-1}) + \sum_{i=1}^{n} a_{3i} S^{i}(1050 + 6S^{i-1}) \tag{4.8}$$

$$S \approx \sum_{i=1}^{n} S^{i}(255 + 6S^{i-1}) \tag{4.9}$$

Onde S^0, S^1, S^2 ... S^n, são os números de neurônios de cada camada. Se a camada i^{th} utilizar uma função linear, então $a_{1i}=1$, $a_{2i}=0$, $a_{2i}=0$, para função log sigmóide $a_{1i}=0$, $a_{2i}=1$, $a_{2i}=0$ e para função tan sigmóide $a_{1i}=0$, $a_{2i}=0$, $a_{2i}=1$.

Para a arquitetura implementada pelos autores, a quantidade de *slices* necessários na implementação foi 6.132, ao utilizar a expressão 4.9 o valor obtido foi 5.931, ou seja um erro de 3,38%.

Nos trabalhos citados, observar-se que os pesquisadores ao implementaram redes neurais artificiais estritamente em *hardware*, utilizando os recursos físicos do FPGA, tiveram algum tipo de dificuldade devido às restrições existentes. Desse modo, foram implementadas redes neurais mais simples.

Ao utilizar a equação 4.9 para prever o espaço necessário na implementação, por exemplo, de uma rede modular. Sendo essa composta por 3 especialistas, cada um formado por 10 neurônios na camada de entrada, 3 na camada oculta e 4 na camada de saída; e 1 rede de passagem, formada por 10 neurônios na camada de entrada, 5 na camada oculta e 4 na camada de saída. A quantidade de espaço necessário para cada especialista é calculada na equação 4.10.

$$S \approx 3(255 + 6 \times 10) + 4(255 + 6 \times 3) = 2037 \tag{4.10}$$

Assim, de acordo com a expressão, para cada especialista são necessários 2.085 *slices*. O cálculo para rede de passagem é mostrado em 4.11.

$$S \approx 5(255 + 6 \times 10) + 4(255 + 6 \times 5) = 2715 \tag{4.11}$$

Portanto, a quantidade de *slices* necessários para a implementação da rede modular completa com 3 especialistas e 1 rede modular é calculado na equação 4.12.

$$S \approx 2037 + 2037 + 2037 + 2715 = 8826 \tag{4.12}$$

Como um *slice* nos FPGA Xilinx equivale à aproximadamente dois LE nos FPGA Altera®, essa rede caso fosse implementada estritamente em *hardware* ocuparia em torno de 17.652 LEs, desconsiderando o erro de 3,38%.

Uma outra forma de abordagem, a qual foi utilizada neste trabalho, é a utilização dos *soft processor*, como o Nios® II, na implementação das redes neurais. A quantidade de LEs necessários para utilização do Nios® II, varia de acordo com a versão utilizada, como mostrado na Tabela 4.4.

Tabela 4.4 – Quantidade de LE necessária para utilização do Nios® II

Nios® II	**Versão**		
	Econômica	**Básico**	**Rápido**
Espaço Necessário	700 LE	1400 LE	1800 LE

Apesar da diferença de desempenho de um processador em *software* em relação a uma implementação em *hardware*, esse permite uma programação mais simples, aproveitando os recursos do processador, assim pode-se construir redes mais complexas. A idéia deste trabalho é aproveitar os recursos existentes no FPGA, embarcando não só um, mas quantos Nios® II forem possíveis, e assim aproveitar a vantagem do processamento de vários Nios® II executando em paralelo.

Para uma discussão mais detalhada, e outros exemplos de implementações de redes neurais em FPGA consultar [Omondi & Rajapakse 2006].

4.5 Conclusão

Redes neurais artificiais tem se apresentado como uma boa ferramenta para soluções de problemas relacionados a aproximação de funções e classificação de padrões. Isso incentiva cada vez mais pesquisadores a buscarem novas aplicações para as RNA, utilizando essas em algoritmos mais complexos. Com o aumento da complexidade das aplicações, a arquitetura da RNA também aumenta. Portanto, parâmetros como o número de neurônios e a quantidade de camadas também aumentam, dificultando a implementação dessas em sistemas embarcados.

Neste capítulo foi apresentado a utilização de FPGAs na implementação de RNAs. Foi mostrada a dificuldade de se implementar uma rede neural em *hardware*, e propõe como solução alternativa, a utilização de um *software processor*. Além disso sintetizou a teoria de redes neurais artificiais necessária, que em conjunto com os conhecimentos apresentados nos capítulos anteriores, ajudam na compreensão do trabalho mostrado no capítulo seguinte. O objetivo deste trabalho foi implementar estruturas complexas de redes neurais embarcadas em um FPGA com múltiplos processadores, unindo três áreas da computação, os sistemas paralelos, embarcados e inteligentes. Assim, no capítulo cinco são mostrados as implementações e as avaliações de duas máquinas de comitê, compostas por MLPs e SVMs, em um ambiente embarcado com múltiplos processadores.

Capítulo 5

Plataforma, Experimentos e Resultados

Os capítulos anteriores apresentaram o embasamento teórico necessário para à compreensão dos experimentos realizados nas plataformas utilizadas. Esse capítulo visa mostrar esses experimentos e a discussão dos resultados obtidos.

Os experimentos foram divididos em etapas que são mostradas em detalhes a seguir. A primeira foi testar e documentar possíveis opções de comunicação entre os Nios® II e assim eleger a opção com melhor desempenho de comunicação. Eleita a forma no qual os processadores realizariam as comunicações, o passo seguinte foi adaptar a fase de execução dos algoritmos de redes neurais implementados em computadores pessoais para utilização na arquitetura embarcada.

As redes neurais implementadas foram as máquinas de comitê estática e dinâmica. A estática é baseada em *ensembles* de SVM, enquanto a dinâmica utiliza MLP na arquitetura dos seus especialistas.

Além da adaptação para utilização na arquitetura embarcada, foi necessário transformar o algoritmo seqüencial em um algoritmo paralelo, seguindo a metodologia proposta por Foster. Assim, com o algoritmo paralelo foi possível aproveitar as vantagens da arquitetura paralela com os múltiplos processadores.

5.1 Metodologia

Os procedimentos e as etapas necessárias para execução do trabalho são descritas a seguir:

1. Definição da quantidade máxima de processadores de *software* Nios II® que podem ser instanciados em cada um dos FPGA utilizados. Essa etapa foi importante para conhecer quantos Nios II® poderiam ser utilizados em cada uma das plataformas utilizadas.

2. Definição do paradigma de comunicação entre os Nios II®. Foram realizados experimentos com algoritmos de multiplicação de matrizes utilizando a comunicação através de passagem de mensagens e através da memória compartilhada. Essa etapa foi necessária para definir a melhor forma de comunicação entre os múltiplos Nios II®.
3. Implementação da máquina de comitê dinâmica, utilizando MLPs como especialistas, em um sistema com processador de uso geral e no FPGA. A rede neural foi implementada em um computador e os resultados obtidos foram comparados com os da rede implementada e executada no FPGA. O algoritmo desenvolvido no computador foi reescrito utilizando as bibliotecas da linguagem C utilizadas para compilação e execução de algoritmos no *software processor* Nios II® instanciados no FPGA. Comparando os resultados foi possível verificar e validar o algoritmo implementado.
4. Implementação da máquina de comitê estática baseada em SVMs em um sistema com processador de uso geral e no FPGA. Os procedimentos utilizados nessa etapa foram idênticos aos da máquina de comitê dinâmica descritos na terceira etapa. O objetivo das etapas 3 e 4 foi mostrar a possibilidade de implementar redes neurais complexas em dispositivos com recursos limitados como os FPGAs.
5. Implementação dos algoritmos paralelos: para realizar a implementação dos algoritmos de redes neurais em paralelo foi utilizado a metodologia proposta e utilizada por Ian Foster em [Foster 1995].
6. Análise dos resultados: Foi verificado o ganho e eficiência obtidos com a utilização de vários processadores em relação com um único processador no mesmo FPGA. Os resultados obtidos na plataforma não foram comparados com outros trabalhos já que não foi observado implementações das redes, implementadas, em dispositivos semelhantes.
7. Foram realizadas medições do consumo de energia dos FPGA. Foi utilizada uma ferramenta disponibilizada pelo fabricante para realizar a verificação do consumo, sendo tais dados disponibilizados para comparações futuras com dispositivos semelhantes.

As etapas apresentadas são discutidas em detalhes nas próximas seções.

5.2 Plataformas utilizadas para realização dos experimentos.

As plataformas de desenvolvimento utilizadas foram baseadas na família de FPGA *Cyclone*® fabricada pela Altera®. Essas família foi projetada para atender os requisitos de desenvolvedores que necessitavam de uma plataforma de baixo custo e pouco consumo. A cada nova geração, a Altera® procurou resolver os desafios técnicos encontrados na geração anterior, tais como, consumo, desempenho e maior integração do circuito integrado. A Tabela 5.1 mostra algumas características de cada geração.

Tabela 5.1 – Características da família de FPGA *Cyclone*®.

	Cyclone® I	Cyclone® II	Cyclone® III
Ano de lançamento	2002	2004	2006
Tecnologia de fabricação	130 nm	90 nm	65 nm
Elementos lógicos (até)	20.600	68.416	198.464
Bits de memória	294.912	1.152.000	8.211.000
Velocidade memória (até)	200 MHz	216 MHz	260 MHz
Suporte a mem. externa	SRAM	DDR2	DDR2
Tensão no núcleo	1.5V	1.2V	1.2V

O *Cyclone*® *I* foi o primeiro FPGA da família de baixo custo da Altera® e ainda tem sido utilizado por desenvolvedores que não necessitam de muitos recursos. Devido a maior integração dos componentes na fabricação do *Cyclone*® *II*, nesse é possível obter uma quantidade maior de elementos lógicos e bits de memória, assim como, integrá-lo a um DSP. O *Cyclone*® *II* possui um consumo de energia menor que plataformas de desenvolvimento similares, sendo o seu consumo 50% menor que o da geração anterior. Já a terceira geração lançada em 2006 consome menos da metade de energia necessária pela geração anterior oferecendo duas vezes mais elementos lógicos e oito vezes o número de bits de memória. Para obtenção dos resultados foram utilizados o *Cyclone*® *Nios*® *II Development Kit*, o *Cyclone II*® *DSP Development Kit* e o *Cyclone III*® *DSP Development Kit.*

O *Cyclone*® *Nios*® *II Development Kit* possui um FPGA Altera® *Cyclone*® EP1C20F400C7 com 20.600 elementos lógicos e 294.912 bits de memória. Também fazem parte da placa de desenvolvimento, um oscilador de 50 MHz, um *compact flash* com dezesseis megabytes, para armazenamento de arquivos se necessário, um megabyte de memória SRAM, oito megabytes de memória SDRAM, que podem ser utilizadas como memória de programa e dados dos algoritmos implementados no

Nios® II. Além do próprio FPGA, também estão disponíveis as interfaces de comunicação ethernet e serial.

O FPGA existente no *Cyclone® II DSP Development Kit* é um Altera® *Cyclone® II* EP2C35F672 que possui 33.216 elementos lógicos e 483.840 bits de memória. Dentre outros elementos a placa de desenvolvimento possui: um oscilador de 50 MHz, um megabyte de memória SRAM, duzentos e cinqüenta e seis megabytes de memória DDR2, conversor analógico digital e uma interface de comunicação serial.

O terceiro *kit* utilizado foi o *Cyclone® III DSP Development Kit*, que possui um FPGA Altera® *Cyclone® III* EP3C120F458 com 119.088 elementos lógicos e 3.981.312 bits de memória. A placa de desenvolvimento, além do FPGA, possui duzentos e cinqüenta e seis megabytes de memória DDR2, oito megabytes de SRAM, sessenta e quatro megabytes de memória *flash* e um oscilador de 75 MHz.

Para implementação e a execução dos experimentos realizados as características consideradas mais importantes são a quantidade de elementos lógicos e bits de memória existentes no FPGA, assim como a freqüência de operação do núcleo do Nios® II. A Tabela 5.2 mostra um resumo das informações dos FPGA apresentados.

Tabela 5.2 – Recursos disponíveis em alguns FPGA.

	Cyclone® EP1C120	Cyclone® II EP2C35	Cyclone® III EP3C120
Elementos Lógicos	20.600	33.216	119.088
RAM Bits	294.912	483.840	3.981.312
Freqüência Nios® II	50MHz	50MHz	75MHz

Na próxima seção são detalhados os experimentos realizados para definição do modelo de comunicação utilizado entre os processadores.

5.3 Experimentos para definição da comunicação

Os sistemas com múltiplos processadores Nios® II necessitam de outros blocos funcionais para sua correta operação, tais como: temporizadores, *mutex*, *mailbox*, *performance counter* e memória. Assim o primeiro passo foi testar a quantidade de processadores que o FPGA suportaria, e assim definir o número máximo de processadores. Algumas das configurações geradas para saber a quantidade de blocos lógicos necessários, para utilização de mais de um processador e a porcentagem ocupada pelo projeto no FPGA, são mostradas na Tabela 5.3.

Tabela 5.3 – Configurações geradas.

Configuração			Suportado[2]	Quantidade de LE necessários	Porcentagem Ocupada[2]		
Nios® II	Core[1]	Mailbox			C1	C2	C3
4	E	4	C1, C2, C3	11.134	54,04	33,51	9,34
4	E	6	C1, C2, C3	12.170	59,07	36,63	10,22
4	E	3	C1, C2, C3	11.151	54,13	33,57	9,36
6	E	4	C1, C2, C3	17.116	83,07	51,52	14,37
6	E	6	C1, C2, C3	17.845	86,62	53,72	14,98
8	E	2	C2, C3	21.146	-	63,67	17,75
4	B	0	C1, C2, C3	17.035	82,69	50,69	14,30
6	B	0	C2, C3	25.536	-	76,87	21,44
8	B	0	C3	33.699	-	-	28,29
12	B	0	C3	45.593	-	-	38,28
16	B	0	C3	57.820	-	-	48,55
16	R	0	C3	72.537	-	-	60,91

[1]E→ Econômico, B→ Básico e R → Rápido [2]C1 → Cyclone I, C2 → Cyclone 2 e C3 → Cyclone 3

Para gerar as configurações da Tabela 5.3 foi utilizado um IBM-PC Pentium 4 3.0 GHz com 2 Gbytes de memória RAM, e os programas *Quartus II* e *SOPC builder* versões 7.2. (*Cyclone® I* e *II*) e 8.0 (*Cyclone® III*).

O fabricante Altera® [Altera Nios II 2009] realizou testes de desempenho com as três versões do Nios® II. Os experimentos foram realizados utilizando o Nios® II econômico operando a 200 MHz, o básico a 165 MHz e o rápido a 185 MHz, ambos foram implementados em um FPGA *Statrix® II*. Nessas configurações, a versão básica apresentou um desempenho quatro vezes maior que a econômica e de aproximadamente 65% da rápida.

Nos experimentos realizados neste trabalho também foi observado um desempenho superior da versão básica relacionada a econômica. Devido a quantidade de blocos lógicos existentes nos FPGAs disponíveis optou-se por utilizar a versão básica no *Cyclone® I* e *II*. Testes com a versão rápida foram realizados apenas no *Cyclone® III*. Assim, como apresentado em [Huerta 2007] foi observado que o fator limitante para quantidade de processadores é o tamanho do FPGA. No *Cyclone® I* foi possível utilizar 4 Nios® II básicos, enquanto no *Cyclone® II* o número máximo de processadores na versão básica foi 6. No *Cyclone® III* foram realizados experimentos com 16 processadores tanto na versão básica como na rápida.

Definida a quantidade de processadores que seria utilizada, foram realizados testes com as duas formas de comunicação existentes: a passagem de mensagens e a

memória compartilhada. No paradigma da passagem de mensagens foram testadas as topologias de interconexão: mestre-escravo, anel e barramento.

5.3.1 Comunicação dos Múltiplos processadores através da passagem de mensagens.

Para que os Nios® II possam realizar comunicação através da troca de mensagens é necessária a utilização de uma caixa postal (*mailbox*) detalhada no capítulo três. As topologias de interconexão implementadas são mostradas na Figura 5.1, onde P significa processador, MB significa *mailbox* e ARB significa árbitro.

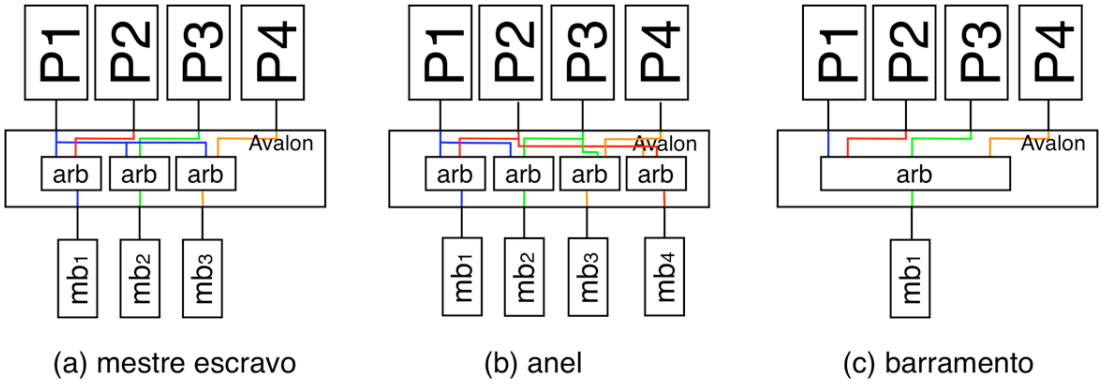

Figura 5.1 – Topologias de interconexão implementadas.

Existem duas funções disponibilizadas pela Altera® para ler o conteúdo da caixa postal. A primeira é a `get()`, sendo essa uma função não bloqueante, ou seja, o processador leitor tenta ler a mensagem na caixa postal, independente da disponibilidade dessa mensagem e continua executando o fluxo de instruções; a segunda é o `pend()`, que bloqueia o fluxo de instruções, até que a mensagem requerida esteja disponível para leitura. Para escrever a mensagem é utilizada a função `post()` que quando executada, o processador verifica se existe espaço disponível e escreve uma mensagem na caixa postal, caso contrário retorna um erro.

No *Cyclone® Nios® II Development Kit* foram realizados dois experimentos utilizando o paradigma de passagem de mensagens. O primeiro foi a implementação de um algoritmo gerador de tráfego, para verificar a velocidade de transmissão alcançada e a consistência dos dados. O segundo foi a implementação de algoritmos de multiplicação de matrizes; seqüencialmente em um único processador e a versão paralela em dois e quatro processadores. Esses experimentos são descritos a seguir.

No algoritmo gerador de tráfego existem três tipos de comunicação, a primeira *todos-para-um* mostrada na Figura 5.2(a), a segunda *um-para-todos* mostrada na Figura 5.2(b) e a última, vista na Figura 5.2(c), é uma comunicação *um-para-um* na qual apenas dois processadores se comunicam.

Para comunicação *um-para-todos*, foram enviadas 65.536 mensagens de um processador, denominado mestre, para cada um dos outros existentes, chamados de escravos. Da mesma forma, na comunicação *todos-para-um*, os processadores escravos enviam, cada um, 65.536 mensagens para o processador mestre. Na comunicação *um-para-um* dois processadores quaisquer se comunicam. Deve ser observado que tais mensagens já representam uma comunicação intensa para esse tipo de arquitetura. Cada um dos testes de comunicação realizados foi repetido trinta vezes para verificar a variabilidade dos resultados.

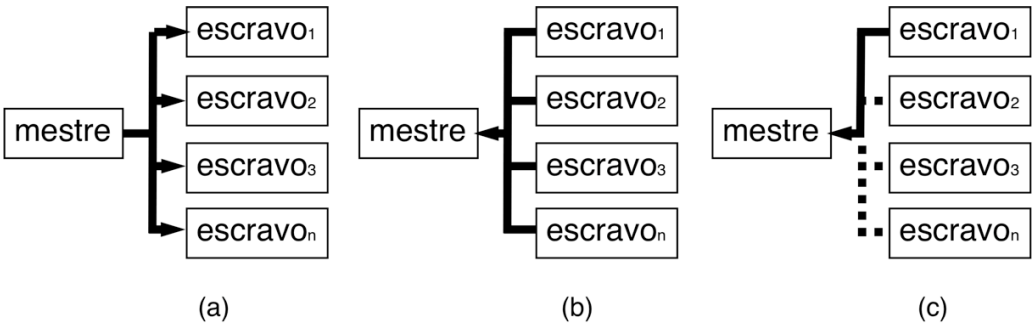

Figura 5.2 – Comunicação (a) *um-para-todos*, (b) *todos-para-um* (c) *um-para-um*.

Para que os receptores possam ler as mensagens dos transmissores, é necessário que eles escrevam as mensagens nas caixas postais, utilizando uma das seguintes funções: `get()`, `pend()` ou `sinc()`. A última foi criada como alternativa as rotinas já existentes, sua implementação foi realizada segundo o pseudo código apresentado na Figura 5.3.

```
1. Procedimento Sinc_Receiver
2.    Receive request
3.    Send ack
4.    Wait msg
5. Fim Sinc_Receiver
```

```
1. Procedimento Sinc_Sender
2.     Send request
3.     If ack is true
4.         Send msg
5.     Else
6.         Do null
7. Fim Sinc_Sender
```

Figura 5.3 – Pseudo código da função *sinc()*.

Na topologia mestre-escravo, cada escravo, denominados P_2, P_3, e P_4, possui uma caixa postal independente para realizar a comunicação com o processador mestre

denominado P_1. Na topologia anel, o processador P_4 não possui comunicação direta com P_1, para que essa ocorra, P_4 envia os dados para P_2 ou P_3, e esses realizam a comunicação com P_1. Na última topologia testada, a do tipo barramento, os processadores se comunicam através de uma única caixa postal.

O valor enviado por cada um dos processadores é conhecido, portanto quando o processador receptor lê qualquer outro valor diferente deste, considera a mensagem lida incorretamente. Em todos os experimentos realizados, a quantidade de mensagens lidas corretamente foi de 100%.

A velocidade de comunicação foi testada utilizando o algoritmo gerador de tráfego, para isso foi calculado quanto tempo foi necessário para que todas as mensagens, de 32 bits, fossem transmitidas, ou seja, escritas e lidas da caixa postal, que possui um *buffer* para armazenamento de cinco mensagens.

Na comunicação *um-para-um* o procedimento adotado foi o seguinte: para as topologias mestre-escravo e barramento, um processador aleatório envia dados para o receptor, enquanto na anel é o processador mais distante que envia os dados. Os resultados obtidos são mostrados na Figura 5.4.

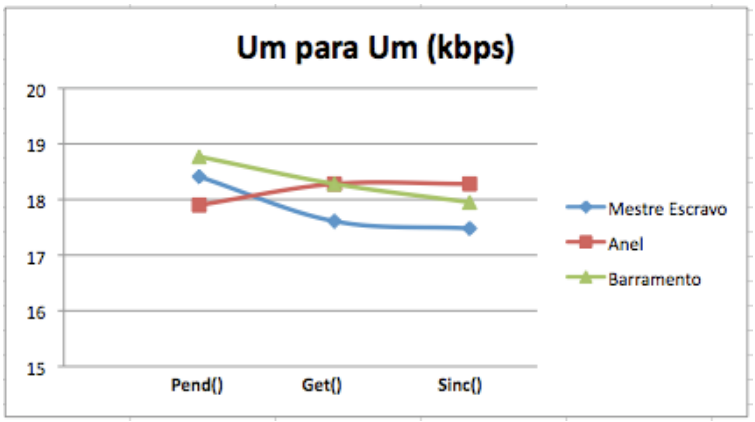

Figura 5.4 – Velocidade da comunicação *um-para-um*.

Na comunicação *todos-para-um* foi realizado o mesmo procedimento para as três topologias. Três processadores enviam simultaneamente mensagens para um processador, o receptor. Os resultados são mostrados na Figura 5.5.

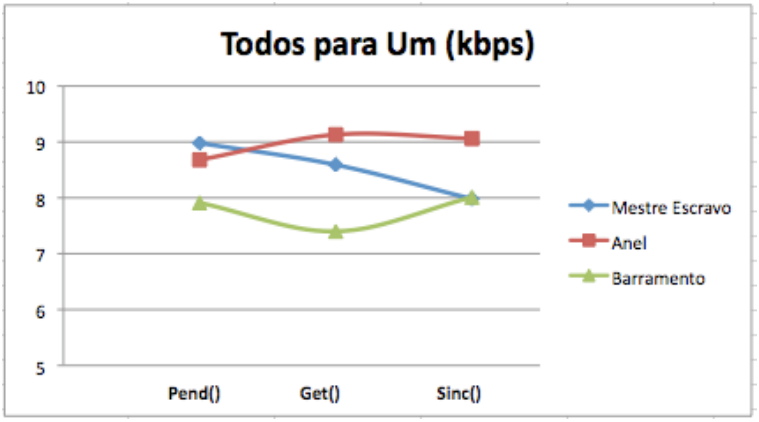

Figura 5.5 – Velocidade da comunicação *todos-para-um*.

O procedimento para medição da velocidade, no caso da comunicação *um-para-todos* é o inverso da *todos-para-um*, ou seja, nas três topologias apenas um processador envia mensagens para os restantes, os receptores. A Figura 5.6 apresenta os resultados obtidos.

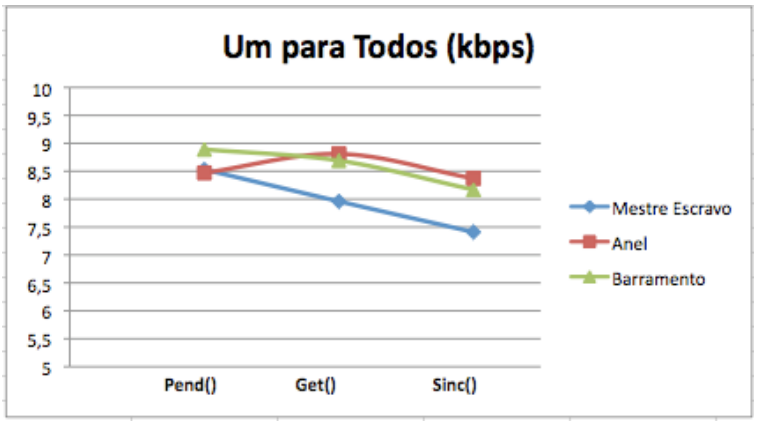

Figura 5.6 – Velocidade da comunicação *um para todos*.

Os valores mostrados, nas três figuras anteriores, são as médias obtidas das trintas execuções de cada uma das comunicações, sendo essa quantidade escolhida aleatoriamente, para que fosse observado a variabilidade dos resultados. Observa-se que os resultados obtidos são semelhantes nas três topologias, assim como a variação entre as funções utilizadas para realizar a comunicação.

A comunicação com o melhor desempenho foi a *um-para-um* justificado pelo fato que apenas um processador requer a caixa postal, conseqüentemente, o barramento para executar a operação de escrita.

Na *todos-para-um* os canais de comunicação são utilizados simultaneamente, acarretando uma degradação no desempenho. Isso se deve ao fato de que

internamente todos os componentes utilizam o mesmo barramento Avalon para realizar a comunicação, gerando uma disputa por esse.

A queda de desempenho da comunicação *um-para-todos* é previsível, pois um processador, envia de forma seqüencial, as mensagens para cada um dos processadores receptores.

Como as variações obtidas entres os experimentos realizados foram mínimas, foi escolhido utilizar a topologia mestre-escravo para implementação dos algoritmos propostos, pois essa é semelhante a necessidade de comunicação dos algoritmos neurais.

Para testar o desempenho da arquitetura na execução de um problema real, foi implementado um algoritmo de multiplicação de matrizes com sessenta e quatro linhas e sessenta e quatro colunas. Esse algoritmo foi escolhido, pois em sua execução existem diversas operações de adição e multiplicação, sendo essas as operações básicas dos algoritmos neurais implementados. O tamanho reduzido das matrizes se deve ao fato que o sistema possui limitações de memória.

Para comparar o desempenho da arquitetura paralela, um algoritmo de multiplicação de matrizes, seqüencial, também foi implementado e foi executado em um único Nios® II. Assim, a multiplicação de matrizes, de números inteiros, foi realizada em um, dois e quatro processadores. O algoritmo paralelo foi implementado de acordo com o procedimento *multiplica_matriz_caixa_postal*.

Procedimento multiplica_matriz_caixa_postal

```
01:     Inicialização variáveis
02:       Inicializa Matriz C [64][64]
03:     Fim inicialização variáveis
04:     Ler quantidade  processadores
05:     Ler identificação processador
06:     Se processador = 1 então
07:       Inicializa Matriz A[64][64]
08:       Inicializa Matriz B[64][64]
09:       Para i=2 até quantidade processadores faça
10:           Envia bloco A[identificação processador]
11:           Envia Matriz B[64][64]
12:       Multiplica bloco A[i]xB[64][64]
13:       Aguarda recebimento de todos os blocos
14:       Monta Matriz C[64][64]
15:       Mostra a Matriz C[64][64]
16:     Senão
17:       Aguarda recebimento do bloco A[i]
18:       Aguarda recebimento da Matriz B[64][64]
19:       Multiplica bloco A[i]xB[64][64]
20:       Envia bloco C[i]
21: Fim multiplica_matriz_caixa_postal
```

O processador mestre inicializa as matrizes A e B e envia para cada um dos processadores escravos toda a matriz B e algumas linhas da matriz A. Cada processador realiza a multiplicação nos elementos, os quais foram destinados. No final, os escravos enviam os resultados parciais para o mestre e esse monta o resultado total. As comunicações existentes são mostradas na Figura 5.7, sendo que as setas verdes representam a comunicação *um-para-todos* e as azuis a comunicação *todos-para-um*.

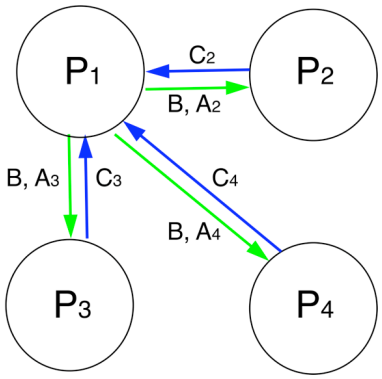

Figura 5.7 – Comunicação entre os processadores na multiplicação de matrizes.

Os desempenhos obtidos na multiplicação de matrizes são mostrados na Figura 5.8. Como pode ser observado, a comunicação através das caixas postais mostrou-se inviável, pois a quantidade de ciclos necessários para a execução do algoritmo paralelo é maior que do algoritmo seqüencial.

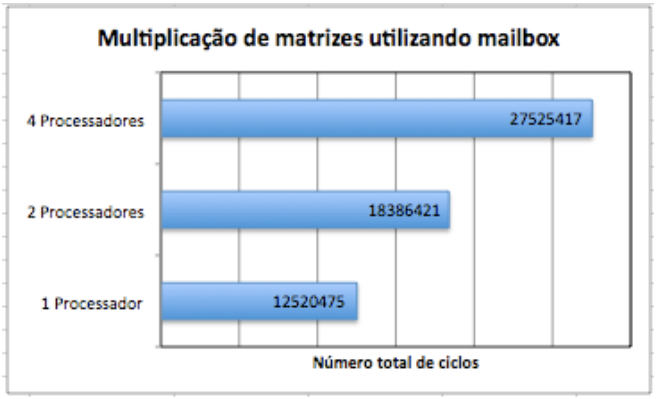

Figura 5.8 – Desempenho da multiplicação de matrizes utilizando caixas postais.

Para execução do algoritmo utilizando quatro processadores foram necessários mais que o dobro de ciclos utilizados pela execução em apenas um processador. Isso ocorre porque apesar do tempo de processamento ter diminuído com o acréscimo de mais processadores, o tempo de comunicação aumentou. Como apresentado no

capítulo dois, o tempo de execução de um algoritmo paralelo pode ser calculado através da equação 5.1.

$$T_{ep} = T_P + T_C + T_E \tag{5.1}$$

No qual T_P é o tempo de processamento, T_C é o tempo de comunicação e T_E o tempo de espera, ou ociosidade do processador. Como T_C aumentava com o acréscimo de novos processadores, esse contribuiu para o aumento do T_{ep}, tornando essa arquitetura impraticável. Assim, novos experimentos foram realizados utilizando o segundo paradigma de comunicação entre processadores, a memória compartilhada.

5.3.2 Comunicação dos Múltiplos processadores utilizando memória compartilhada.

Como visto no capítulo três, os processadores Nios® II, também podem realizar comunicação através da utilização de um *mutex* em conjunto com uma memória compartilhada. Esse dispositivo irá garantir a exclusão mútua e à atomicidade das operações executadas na memória compartilhada. A Figura 5.9(a) mostra como é organizado o sistema com memória compartilhada. A Figura 5.9(b) mostra como essa arquitetura é mapeada no FPGA.

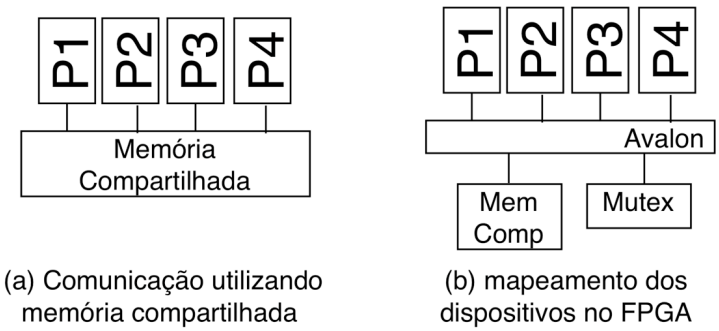

Figura 5.9 – Sistema utilizando memória compartilhada.

O algoritmo de multiplicação de matrizes paralelo utilizando caixas postais implementado anteriormente foi modificado para que a comunicação fosse realizada através da memória compartilhada. O procedimento *multiplicação_matriz_paralelo _usando_memoria_compartilhada* apresenta o pseudo código do algoritmo utilizado, sendo esse executado em todos os processadores.

Procedimento multiplicação_matriz_paralelo_usando_memoria_compartilhada

```
01:     Inicialização variáveis
02:         Inicializa Matriz A [64][64] em memória compartilhada
03:         Inicializa Matriz B [64][64] em memória compartilhada
04:         Inicializa Matriz C [64][64] em memória compartilhada
05:         Inicializa  flag = 0 em memória compartilhada
06:     Fim inicialização variáveis
07:     Ler quantidade  processadores
08:     Ler identificação processador
09:     Inicio Multiplicação
10:         Ler identificação processador
11:         Para (i =(((64/quantidade processadores)*(identificação processador -1))+1); i <
((64/quantidade processadores) * identificação processador); i++) faça
12:                 Para (j =1;  j < 64; j++) faça
13:                     Para (k=1; k < 64; k++) faça
14:                         Matriz C[i][j] = Matriz C[i][j] + Matriz B[i][k]*Matriz C[k][j]
15:     Fim Multiplicação
16:     Se identificação processador != 1 então
17:         flag =flag +1
18:     Senão enquanto (flag != 3)
19:         Espere
20:     Mostra Matriz C
21: Fim multiplicação_matriz_paralelo_usando_memoria_compartilhada
```

No algoritmo da multiplicação de matriz em paralelo, o primeiro passo é identificar em qual processador o mesmo está sendo executado, para então realizar a multiplicação de um conjunto específico de dados da matriz. Como as matrizes são armazenadas em uma memória compartilhada, na qual todos os processadores tem acesso, o tempo de comunicação é praticamente nulo, sendo esse igual ao tempo de transferência do barramento. Quando os processadores escravos finalizam a execução um *flag* de sinalização é modificado, para que o mestre possa ler o valor correto da matriz resultante. A Figura 5.10 mostra o tempo de execução, em número de ciclos, necessário para execução do algoritmo com um, dois e quatro processadores.

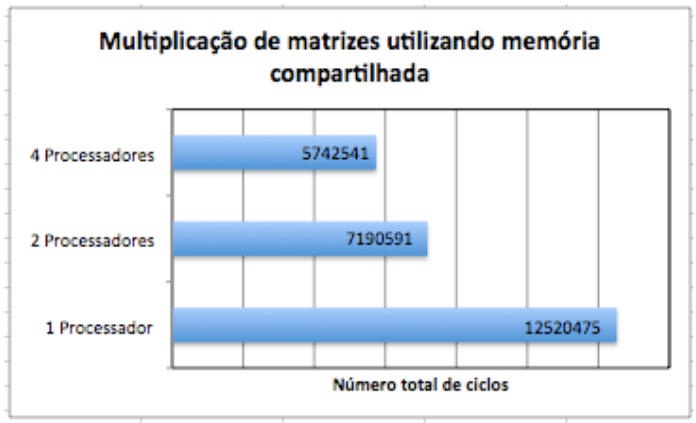

Figura 5.10 – Desempenho da multiplicação de matrizes com memória compartilhada.

Observando os resultados obtidos no algoritmo de multiplicação de matrizes utilizando memória compartilhada é possível perceber a diminuição no tempo de execução, quando são utilizados mais processadores. Essa melhoria deve-se a diminuição do tempo de processamento e do tempo de comunicação reduzido.

Os experimentos com a multiplicação de matrizes utilizando caixas postais e memória compartilhada foram realizados pois as operações de soma e multiplicação presentes nesse algoritmo são similares as operações básicas das redes neurais implementadas, além disso as comunicações são semelhantes as necessárias aos algoritmos neurais que foram implementados.

O tempo de execução do algoritmo que utiliza caixas postais aumentou a medida que acrescentava mais processadores, enquanto na memória compartilhada esse tempo diminuiu. Assim, devido ao fraco desempenho observado na utilização das caixas postais, optou-se pela utilização da memória compartilhada para realização da comunicação na plataforma com os algoritmos neurais. Devido aos resultados obtidos, optou-se por realizar experimentos apenas com a comunicação através da memória compartilhada nos algoritmos neurais que foram implementados.

5.4 Implementação das arquiteturas de redes neurais

Neste trabalho foram implementadas duas arquiteturas de redes neurais. A primeira foi uma máquina de comitê dinâmica utilizando MLPs como especialistas, enquanto a segunda é uma máquina de comitê estática que utiliza SVMs como *ensembles*. As aplicações, os experimentos realizados, assim como os resultados obtidos são descritos nessa seção.

5.4.1 Máquina de comitê dinâmica para classificação de distúrbios da rede elétrica.

De acordo com Cândido [2008], a atual complexidade do sistema elétrico, aliada às novas demandas por parte dos consumidores e à privatização do setor elétrico, tornaram o mercado de energia cada vez mais competitivo e exigente. Portanto é importante realizar uma análise eficiente das perturbações que podem ocorrer no sistema elétrico, para melhorar o índice da qualidade da energia. O

conceito de qualidade da energia está relacionado a um conjunto de alterações que podem ocorrer no sistema elétrico. Podendo ser caracterizado como qualquer problema manifestado na tensão, corrente ou desvio de freqüência, que resulta em falha ou má operação de algum equipamento dos consumidores [Oleskovicz 2004]. Assim, o objetivo dessa analise é observar o conjunto de alterações que podem ocorrer na rede elétrica, ocasionando falha ou má operação em equipamentos.

Diante dessa problemática é evidente a importância de uma análise e diagnóstico da qualidade da energia elétrica, na intenção de determinar as causas e conseqüências dos distúrbios no sistema. Nesse contexto, foi implementada uma máquina de comitê dinâmica para ser aplicada no problema da classificação dos distúrbios da rede elétrica. A base de dados é composta por trezentos e quarenta e quatro vetores de dez entradas. Os dados representam quatro distúrbios que ocorrem na rede elétrica classificados tais como: transitórios, harmônicos elevação e afundamento de tensão.

A máquina de comitê implementada foi a rede modular estendida proposta em [Magalhães 2007]. Ele observou que a rede modular apresentada em [Haykin 2001] era eficiente para alguns problemas simples, entretanto, para problemas mais complexos não apresentou boas soluções, provavelmente por causa da simplicidade dos especialistas, que eram formados por uma única camada. Portanto, uma arquitetura ampliada foi desenvolvida, adicionando camadas ocultadas, como ocorre nas redes MLP, e funções de ativação não lineares tanto na rede de passagem como nos especialistas. A rede implementada é vista na Figura 5.11, onde cada módulo representa um especialista, que são formados por uma MLP de K camadas e N neurônios.

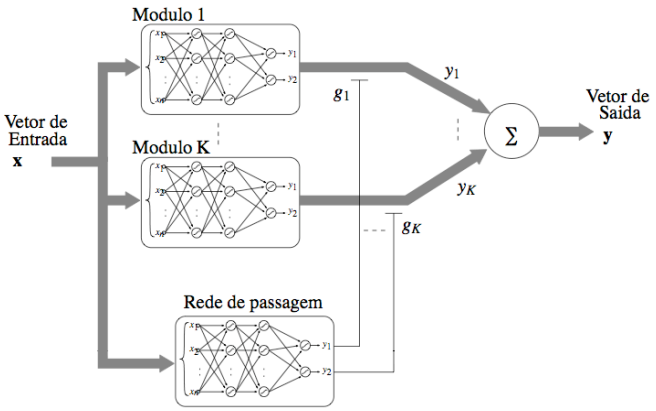

Figura 5.11 – Rede modular implementada no FPGA.

Quatro configurações diferentes de máquinas de comitê para classificar os distúrbios da rede elétrica foram treinadas em [Magalhães 2007], variando apenas a quantidade de neurônios. Nos treinamentos foram obtidos uma classificação de até 100%, o resumo das características de cada uma das redes é mostrado na Tabela 5.4.

Tabela 5.4 – Resumo das redes modulares implementadas.

Rede Modular	**MOD-0**	**MOD-1**	**MOD-2**	**MOD-3**
Numero de especialistas	3	3	3	3
Arquitetura dos especialistas (neurônios por camada)	10:3:4	10:5:4	10:10:4	10:15:4
Arquitetura da rede de passagem (neurônios por camada)	10:5:4	10:5:4	10:10:4	10:10:4
Classificação	98,46%	99,48%	100%	100%

Para implementar as redes modulares da Tabela 5.4 na arquitetura paralela foram analisados os aspectos de comunicação e a memória necessária. A estratégia de paralelização adotada foi a decomposição funcional, no qual cada processador, executa uma tarefa diferente.

Como o próprio nome indica, as redes modulares foram desenvolvidas e inspiradas na idéia de aprender a solucionar o problema de forma modular e dividida. Assim, cada especialista, por exemplo, pode ser associado a uma unidade de processamento. Variações dessa abordagem podem ser implementadas de acordo com a quantidade de especialistas necessários e unidades de processamento.

Os elementos constituintes da rede modular são os especialistas que aprendem sobre o problema e fornecem respostas desejadas, a rede de passagem que aprende a ponderar as saídas dos especialistas e o somatório que gera a saída final do sistema.

Ao implementar a rede modular paralela em N unidades de processamento, observa-se a necessidade de três comunicações. Os dados de entrada devem ser enviados para a rede de passagem e para todos os especialistas (comunicação *um-para-todos*). As saídas dos especialistas devem ser enviadas para que a rede de passagem (comunicação *todos-para-um*) que realiza a ponderação das saídas. A rede de passagem deve enviar as saídas ponderadas (comunicação *um-para-um*) para o combinador gerar a saída final da rede.

Computacionalmente, o trabalho de agrupar as respostas das saídas individuais dos especialistas através de uma combinação é reduzido e pode, por exemplo, ser realizado na mesma unidade de processamento que a rede de passagem, diminuindo assim uma comunicação no sistema.

Os cálculos realizados pelas redes modulares são as saídas dos especialistas, as ponderações da rede de passagem nas saídas dos especialistas e a combinação dessas gerando a saída final. A saída do especialista k é um vetor de saídas Y_k, que é obtido através dos neurônios da sua camada de saída. A organização dos parâmetros e a estrutura das redes modulares implementadas são mostradas na Figura 5.12. O algoritmo, as estratégias de paralelização e resultados obtidos são descritos em seqüência.

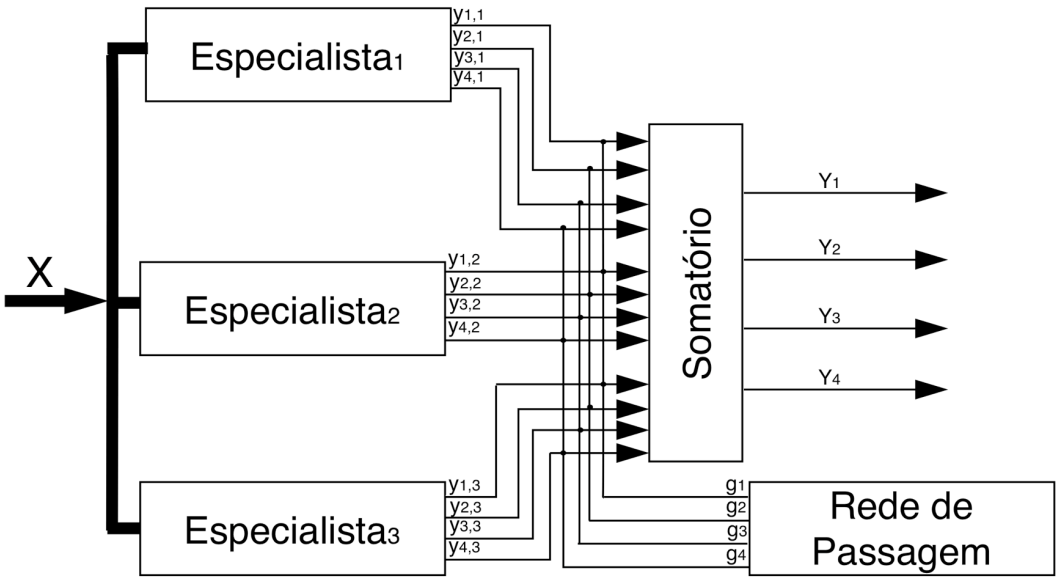

Figura 5.12 – Parâmetros da rede modular implementada no FPGA.

Os experimentos iniciais foram realizados no *Cyclone® Nios® II Development Kit*. Devido à limitação de memória existente nessa plataforma somente a rede modular *MOD-0* foi implementada. A *MOD-0* é composta por três especialistas que possuem três camadas, sendo que a camada de entrada possui dez neurônios, a oculta três e a de saída quatro. Já a rede de passagem, também com três camadas, possui dez na primeira, cinco na segunda e a última camada é composta por quatro neurônios.

Para analisar o desempenho desse e dos outros algoritmos paralelos implementados foram utilizadas as métricas apresentadas no capítulo dois. Portanto, é necessário observar o desempenho do algoritmo quando executado em uma única unidade de processamento. Assim, além da implementação na arquitetura paralela, a rede modular foi executada em uma arquitetura com um único processador. O pseudo-código da implementação seqüencial é visto no procedimento *rede_modular_ sequencial*.

```
Procedimento rede_modular_sequencial
01:     Repita
02:         Leia entrada e escreva na memória
03:         Para ( i=1; i<= numero_especialistas; i++) faça
04:                 Calcule saída do especialista(i)
05:         Calcule saída da rede de passagem
06:         Calcule saída geral da rede
07:         Mostre saída da rede
08:     Enquanto existir entrada
09: Fim rede_modular_sequencial
```

Pode-se observar que a saída geral da rede, executada na linha 06 do procedimento *rede_modular_sequencial*, só pode ser calculada depois da obtenção da saída da rede de passagem, já que essa vai realizar as ponderações necessárias nas saídas dos especialistas para obtenção do resultado final.

Na execução do procedimento *rede_modular_sequencial* foi calculado a quantidade de ciclos necessários para cada uma das funções principais: cálculo das saídas dos especialistas, da rede de passagem e final da rede. Os resultados obtidos para o conjunto de entradas utilizado são apresentados na Tabela 5.5.

Tabela 5.5 – Tempo necessário para execução da rede *MOD-0* em um processador.

Função	Ciclos	Tempo	%
Cálculo da saída dos especialistas	3.850.298	70ms	63,08
Cálculo da saída da rede de passagem	1.902.565	35ms	31,17
Cálculo da saída da rede	350.971	6,4ms	5,75

A função na qual mais demandou tempo foi o cálculo da saída dos especialistas, então foi decidido, paralelizar essa função no algoritmo paralelo, como pode ser observado no pseudo código do procedimento *rede_modular_dois_processadores*.

```
Procedimento rede_modular_dois_processadores
01:     Repita
02:         Leia identificação do processador
03:         Escreva flag = 2
04:         Se identificação do processador = 1 então
05:                 Leia entrada e escreva na memória compartilhada
06:                 Escreva flag = 0
07:                 Calcule saída da rede de passagem
08:                 Enquanto flag = 0
09:                         Espere
10:                 Calcule a saída da rede
11:                 Mostre saída da rede
12:         Senão
13:                 Para ( i=1; i<= numero_especialistas; i++) faça
14:                         Enquanto flag = 2
15:                                 Espere
16:                         Calcule saída do especialista(i)
17:                 Escreva flag = 1
18:     Enquanto existir entrada
19: Fim rede_modular_dois_processadores
```

De acordo com o procedimento *rede_modular_dois_processadores*, o processador 1, denominado mestre, calcula a saída da rede de passagem e aguarda que o processador 2, o escravo, calcule as saídas dos especialistas. Assim, a saída final da rede somente é calculada quando o processador 2 finaliza o seu processamento.

Os resultados obtidos quando a rede *MOD-0* é executada em dois processadores são mostrados na Tabela 5.6. O tempo considerado é quando o processador denominado mestre, termina sua execução, já que o algoritmo é finalizado por esse.

Tabela 5.6 – Tempo necessário para execução da rede *MOD-0* em dois processadores.

Função	**Ciclos**	**Tempo**	**%**
Cálculo da saída da rede de passagem	2.073.565	38ms	46,81
Espera da saída dos especialistas	2.045.505	37ms	46,17
Cálculo da saída da rede	310.990	5,7ms	7,02

Analisando os resultados, percebe-se que o processador mestre passa quase metade do tempo ocioso aguardando que a saída dos especialistas, que são calculadas pelo processador escravo, sejam escritas na memória compartilhada. Isso se deve ao fato que a tarefa realizada pelo escravo necessita mais tempo que a do mestre. Então o algoritmo paralelo foi reescrito como mostrado no procedimento *rede_modular_ quatro_processadores*.

Procedimento rede_modular_quatro_processadores

```
01:     Repita
02:        Leia identificação do processador
03:        Escreva flag = 0
04:        Se identificação do processador = 1 então
05:                Leia entrada e escreva na memória compartilhada
06:                Escreva flag = 1
07:                Calcule saída da rede de passagem
08:                Enquanto flag != 4
09:                        Espere
10:                Calcule a saída da rede
11:                Mostre saída da rede
12:        Senão
13:                Enquanto flag = 0
14:                        Espere
15:                Calcule saída do especialista(identificação do processador - 1)
16:                Escreva flag = flag + 1
17:     Enquanto existir entrada
18: Fim rede_modular_quatro_processadores
```

Com o novo algoritmo, sendo executado em quatro processadores, cada escravo recebe como tarefa o cálculo da saída de apenas um especialista, enquanto o mestre

calcula a saída da rede de passagem e a saída final da rede. Os resultados obtidos podem ser vistos na Tabela 5.7.

Tabela 5.7 – Tempo necessário para execução da rede *MOD-0* em quatro processadores.

Função	Ciclos	Tempo	%
Cálculo da saída da rede de passagem	2.033.123	37ms	67,76
Espera da saída dos especialistas	651.585	11,8ms	21,72
Cálculo da saída da rede	315.937	5,8ms	10,52

Os tempos de execução total da rede *MOD-0* implementada no *Cyclone® I* são vistos no gráfico da Figura 5.13. Nesse é possível observar a melhoria alcançada no tempo total, em número de ciclos, da execução do algoritmos paralelos em relação ao algoritmo seqüencial.

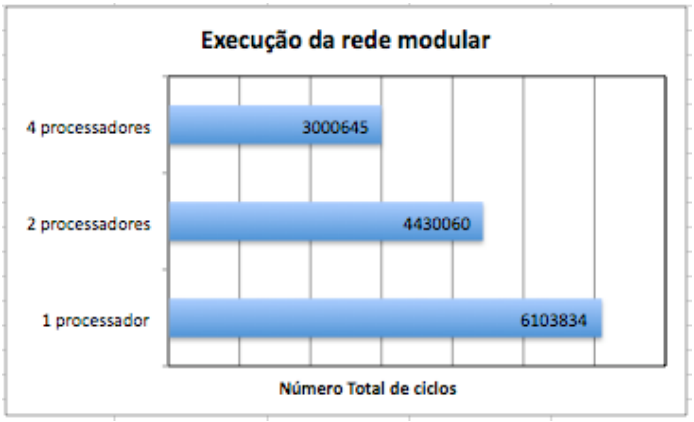

Figura 5.13 – Tempos obtidos para execução da rede *MOD-0* no *Cyclone® I*.

Experimentos idênticos aos realizados com a rede *MOD-0* no *Cyclone® I* foram realizados no *Cyclone® II*, entretanto os resultados obtidos são semelhantes aos do *Cyclone® I*, não justificando a utilização da plataforma para essa aplicação. Essa pode ser utilizada em aplicações mais complexas, que por exemplo, necessite de um maior número de especialistas. Os experimentos foram então repetidos no *Cyclone® III*.

No *Cyclone® III* foi possível realizar a implementação das redes *MOD-0*, *MOD-1*, *MOD-2* e *MOD-3*. Nessa plataforma o número de processadores poderia ser aumentado, entretanto como as redes mencionadas já obtém até 100% de classificação. Essa classificação é igual a obtida quando a rede é executada em um computador, assim optou-se por realizar os experimentos com no máximo 4 processadores. O tempo de processamento, o ganho e a eficiência das redes sendo executadas nesse FPGA com a versão básica do Nios® II a 75MHz são mostrados na Tabela 5.8.

Tabela 5.8 – Resultados obtidos na execução da redes modulares no *Cyclone® III*.

Rede Modular	Tempo (ms)			Ganho		Eficiência	
	1 Proc.	2 Proc.	4 Proc.	2 Proc.	4 Proc.	2 Proc.	4 Proc.
MOD-0	79,57	49,11	30,61	1,62	2,6	0,81	0,65
MOD-1	81,31	47,82	29,03	1,7	2,8	0,85	0,7
MOD-2	83,11	50,36	29,16	1,65	2,85	0,82	0,71
MOD-3	87,21	50,41	29,96	1,73	2,91	0,86	0,72

A segunda aplicação implementada foi uma máquina de comitê estática utilizada para classificar padrões em exames de tiróide. Os experimentos realizados são descritos na próxima sub-seção.

5.4.2 Máquina de comitê estática para auxílio no diagnóstico de disfunções na tireóide.

O funcionamento e os problemas que ocorrem na tireóide apresentados detalhadamente em [Termutas, 2007] são descritos sucintamente a seguir.

Os hormônios produzidos pela glândula tireóide que é vista na Figura 5.14, o *levothyroxine* (T4) e *triiodothyronine* (T3) ajudam a controlar o metabolismo do corpo. Esses são importantes produtores de proteínas que estabilizam a temperatura corporal. Assim, desordens nessa glândula não devem ser ignoradas, pois podem levar a morte. Em geral, os problemas da tireóide podem ser divididos em dois grupos. O primeiro que afeta a funcionalidade da glândula e o segundo que forma tumores na glândula.

As desordens são relativamente comum na população sendo a maioria tratadas com sucesso. Essas anormalidades são relacionadas usualmente a pouca produção dos hormônios (*hipotireoidismo*) ou o excesso desses (*hipertireoidismo*). Esses problemas podem ocasionar uma inflamação na tireóide.

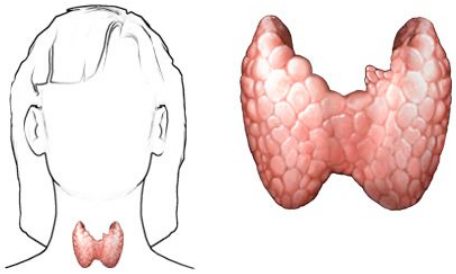

Figura 5.14 – Glândula Tireóide.

Para um correto diagnostico é importante realizar uma investigação e interpretação precisa dos dados existentes nos exames da tireóide. Portanto, a classificação correta dos dados para diagnostico de problemas relacionados a tireóide é uma tarefa crucial. De acordo com Temurtas [Termutas 2007] vários métodos têm sido utilizados para realizar essa classificação. Nesse contexto, a segunda rede neural implementada foi uma máquina de comitê estática, como visto no capítulo quatro, para auxiliar no diagnóstico do problema da tireóide.

A máquina de comitê estática implementada é composta por classificadores independentes; sendo que as saídas calculadas por esses são combinadas para gerar a saída global do sistema. Foi implementada a fase de execução de um *ensemble* de máquinas de vetores de suporte. O *ensemble* implementado é visto na Figura 5.15, e é composto por cinco classificadores, cuja arquitetura são máquinas de vetores de suporte.

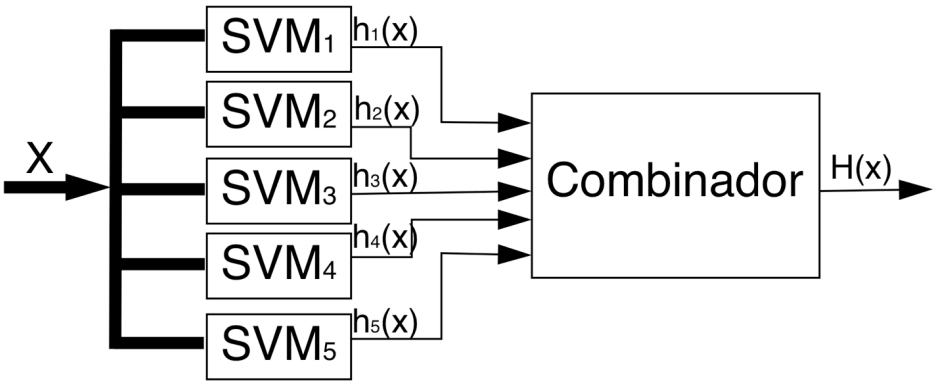

Figura 5.15 – *Ensemble* de vetores de suporte implementado no FPGA.

O cálculo da saída das SVM (h_1,h_2,h_3,h_4,h_5) é dado pela equação 5.2 [Haykin, 2001].

$$h(x) = \sum_{j=1}^{m_i} w_j \varphi_j(x) + b \tag{5.2}$$

onde m é a dimensão do espaço de características, w é o vetor de peso ótimo da SVM, encontrado no treinamento e φ é o mapeamento entre o espaço de entrada e o espaço de características, e depende do *kernel* utilizado, que também é definido no treinamento. Como o treinamento não foi escopo deste trabalho, para mais detalhes consultar [Lima *et al* 2009].

O *kernel* utilizado para cada SVM desta máquina de comitê estática foi o RBF gaussiano. Quando essa função *kernel* é utilizada para gerar o mapeamento entre os espaços de entrada e de características, o treinamento da SVM ao invés de gerar um simples hiperplano, gera uma rede de função de base radial.

Essa rede RBF tem seus centros gerados automaticamente pela SVM. Utilizando o valor da largura de gaussiana fornecido no treinamento, a quantidade de centros é a quantidade de vetores de suporte e as localizações dos centros são os valores dos vetores de suporte.

A equação 5.2 é a que demanda mais tempo na execução da arquitetura implementada, por isso foi a parte do código que foi paralelizada, a outra etapa do algoritmo é realizada pelo combinador. Esse recebe as saídas produzidas pelas SVM e realiza uma combinação linear para gerar a saída final *H(x)* da rede de acordo com a equação 5.3.

$$H(X) = \begin{cases} 1, & se \quad \sum_{t=1}^{t_i} \beta_t h_t \geq 0 \\ -1, & se \quad \sum_{t=1}^{t_i} \beta_t h_t < 0 \end{cases} \tag{5.3}$$

no treinamento do comitê, em cada passo, uma SVM é treinada, seu erro é computado e então é calculado o β, que é uma medida da importância do classificador para o *ensemble*. Por sua vez, h_t representa a saída da SVM t. Portanto a saída final do *ensemble*, ou seja, a saída do combinador é uma votação das saídas das SVMs ponderadas por seus respectivos β. O tempo de computação da equação 5.3 é pouco significativo quando comparado ao tempo de computação da equação 5.2.

Os dados de entrada utilizados são setenta e cinco exemplos divididos em dois padrões, tireóides com ou sem desordem, que compõem uma das realizações do conjunto *thyroid disease diagnoses* disponível em [Benchmark Repository]. Esse conjunto tem sido utilizado como *benchmark* para validação de classificadores. O *ensemble* implementado realiza a classificação desse conjunto de dados.

Cada modelo de SVM é diferente um do outro, devido à distribuição de probabilidade dos dados de entrada, ou seja, cada SVM trabalha com um conjunto de entradas diferentes, que faz parte do conjunto de entrada geral. O primeiro passo foi calcular quanto tempo é necessário para executar cada um dos classificadores individualmente, esse tempo é visto na Tabela 5.9.

Tabela 5.9 – Tempo necessário para execução dos classificadores no *Cyclone*® *I.*

Classificador	Ciclos	Tempo
1	834.876.317	16,70s
2	733.388.855	15,58s
3	821.974.962	16,44s
4	778.113.886	15,56s
5	833.460.012	16,67s

Em seguida foi realizada a execução do *ensemble* completo, com cinco classificadores e um combinador em um único processador. Após a execução em um processador, a mesma arquitetura foi implementada em dois processadores. A combinação da equação 5.3, somente pode ser completada, quando todos os classificadores tiverem calculado suas respectivas saídas. Portanto, foram escolhidos os três modelos mais rápidos (2, 3 e 4) para serem executados no processador mestre e os dois mais lentos no processador escravo. Assim, o tempo ocioso no escravo é reduzido e o mestre não precisa esperar as saídas dos classificadores calculados no escravo, porque quando o mestre finaliza seu processamento, o escravo já tem terminado a execução. Com quatro processadores o mesmo procedimento foi utilizado, entretanto o mestre executa os dois classificadores mais rápidos, enquanto cada um dos escravos executa um dos outros classificadores. Os resultados dos experimentos são mostrados na Tabela 5.10.

Tabela 5.10 – Tempo total para execução da máquina de comitê estática o *Cyclone*® *I.*

Número de Processadores	Ciclos	Tempo
1	4.012.266.152	80,25s
2	2.493.087.474	49,87s
4	1.664.254.711	33,29s

Na Figura 5.16 é possível observar o tempo, em porcentagem, necessário para execução em dois e quatro processadores, relacionado com o tempo requerido para executar em um processador. O ganho alcançado é visto claramente, pois a medida que o número de processadores aumenta o tempo de execução diminui.

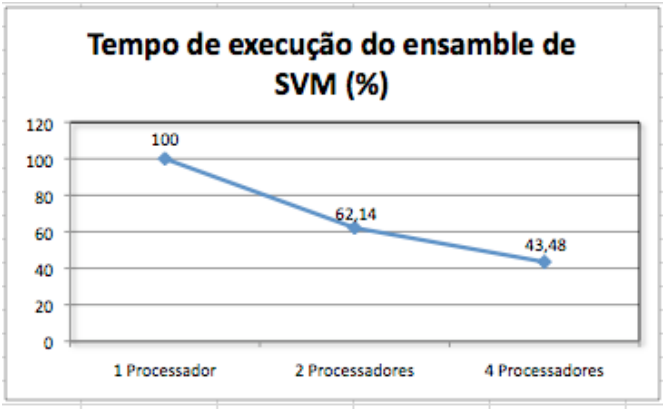

Figura 5.16 – Resultados obtidos para execução do ensemble de SVMs.

Com o tempo de execução de todos os algoritmos implementados no *Cyclone*® *I* tanto seqüencial, como em paralelo é possível calcular, como visto no capítulo dois, o ganho da aplicação paralela e a sua eficiência. Esses são vistos na Tabela 5.11.

Tabela 5.11 – Ganho e eficiência dos algoritmos implementados no *Cyclone*® *I*.

Algoritmo	Número de processadores	Ganho	Eficiência
Multiplicação de Matriz utilizando memória compartilhada	2	1,74	0,87
	4	2,18	0,55
Máquina de comitê dinâmica (rede modular)	2	1,38	0,69
	4	2,03	0,51
Máquina de comitê estática (*ensemble* de SVMs)	2	1,60	0,80
	4	2,41	0,60

Assim, como foi feito nas redes modulares, novos experimentos foram realizados com o *ensemble* de SVMs no *Cyclone*® *III*. Como nesse a quantidade de elementos lógicos é maior, foi possível embarcar mais processadores. Assim, além dos experimentos com o comitê de 5 SVMs no *Cyclone*® *I*, foram realizados testes com um comitê de 8 SVMs e outro de 16 no *Cyclone*® *III*.

Os algoritmos foram paralelizados de duas formas. A primeira, no qual foi realizada uma decomposição funcional, em que, cada processador, executa a mesma quantidade de SVMs, por exemplo, com 4 processadores foram executadas 2 SVMs em cada um. Na segunda foi realizada uma decomposição de dados, no qual os dados de entradas foram divididos em partes semelhantes, para serem executados em processadores diferentes.

Observou-se que, ao aumentar o número de processadores a eficiência da arquitetura paralela diminuía, assim ao invés de aumentar ainda mais o número de processadores, optou-se por utilizar a versão rápida do Nios® II. Dessa forma a maior quantidade de processadores utilizada foi 16. Os resultados obtidos para os

experimentos com a versão básica do Nios® II são apresentados na Tabela 5.12 e com a versão rápida na Tabela 5.13.

Como o objetivo foi avaliar o desempenho de uma SVM mais complexa, não foi considerado a taxa de acertos na classificação. Entretanto as classificações obtidas para os dados de entrada utilizados são: 95,53% para 5 SVMs, 95,6% para 8 SVMs e 95,47% para 16 SVMs. A precisão na classificação dos dados de entradas no FPGA é igual a classificação obtida quando a rede neural é executada em um computador.

Tabela 5.12 – Tempo, ganho e eficiência dos SVMs implementados no *Cyclone® III* utilizando Nios® II Padrão.

Proc.	8 SVMS						16 SVMS					
	Funcional			Dados			Funcional			Dados		
	T(S)	G	E	T(S)	G	E	T(S)	G	E	T(S)	G	E
1	77,88	-	-	77,88	-	-	156,58	-	-	156,58	-	-
2	41,91	1,85	0,9	49,94	1,55	0,77	82,09	1,90	0,9	95,02	1,64	0,82
4	21,56	3,61	0,9	36,06	2,15	0,53	43,98	3,56	0,89	51,98	3,01	0,75
6	-	-	-	31,65	2,46	0,41	-	-	-	42,49	3,67	0,61
8	13,55	5,74	0,7	29,35	2,65	0,33	23,59	6,64	0,83	32,55	4,81	0,60
16	-	-	-	-	-	-	16,70	9,38	0,58	21,53	7,27	0,45

Tabela 5.13 – Tempo, ganho e eficiência dos SVMs implementados no *Cyclone® III* utilizando Nios® II Rápido.

Proc.	8 SVMS						16 SVMS					
	Funcional			Dados			Funcional			Dados		
	T(S)	G	E	T(S)	G	E	T(S)	G	E	T(S)	G	E
1	39,82	-	-	39,51	-	-	78,90	-	-	78,89	-	-
2	24,18	1,64	0,82	32,38	1,22	0,61	43,67	1,80	0,9	47,65	1,65	0,82
4	12,15	3,28	0,82	17,95	2,2	0,55	29,38	2,68	0,67	36,23	2,17	0,54
6	-	-	-	12,91	3,06	0,51	-	-	-	27,21	2,89	0,48
8	7,22	5,51	0,68	10,50	3,76	0,47	15,17	5,2	0,65	21,91	3,6	0,45
16	-	-	-	-	-	-	9,35	8,44	0,52	11,45	6,88	0,43

Para verificar o consumo de energia do FPGA foi utilizada a ferramenta disponibilizada pela Altera® *PowerPlay Analyzer Tool*. Com essa ferramenta é possível estimar a potência dos projetos desenvolvidos no FPGA. Na Tabela 5.14 é apresentado o consumo de energia dos sistemas utilizados e de outras unidades de processamento. Pode-se observar que a potência necessária do sistema em FPGA é menor que a potência dos processadores de uso geral e em alguns casos menor que a de unidades de processamento semelhantes, como o DSP.

Tabela 5.14 – Potência necessária

Cyclone® I com 1 Nios II® Básico	0,63 W
Cyclone® I com 4 Nios II® Básicos	0,7 W
Cyclone® III com 1 Nios II® Básico	0,46 W
Cyclone® III com 1 Nios II® Básico	1,23 W
Pentium M 1.2 Ghz [Intel 2009]	22 W
Mobile Pentium 4 2.4 [Intel 2009]	24,92 W
Core 2 Solo ULV U2100 [Intel 2009]	5,5 W
Atom N270 [Intel 2009]	2,5 W
Athlon XP 1700+ [Amd 2009]	64 W
Athlon XP 64 X2 4050e [Amd 2009]	45 W
DSP TMS320C6412 [Texas 2009]	1,02 W

5.5 Conclusão

Neste capítulo foram apresentados os algoritmos implementados nas plataformas embarcadas. Três algoritmos foram implementados, uma multiplicação de matrizes, uma máquina de comitê dinâmica, composta por SVMs, e uma estática composta por MLPs. O objetivo foi analisar o desempenho dos algoritmos quando embarcados em uma plataforma paralela e com recursos limitados, como as que foram utilizadas. Todos os algoritmos tiverem um ganho de desempenho quando o paralelismo foi implementado. As Figura 5.17 e Figura 5.18, mostram um resumo dos resultados obtidos utilizando o *Cyclone® I*. As Figura 5.19, Figura 5.20, Figura 5.21 e Figura 5.22 mostram o resumo dos resultados obtidos utilizando o *Cyclone® III*.

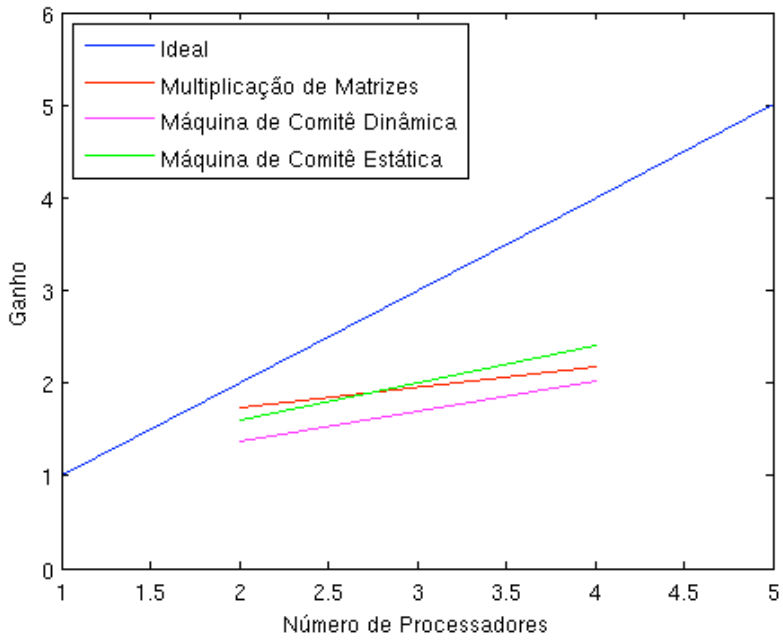

Figura 5.17 – Ganho dos algoritmos executados no *Cyclone® I*, comparados com o ideal.

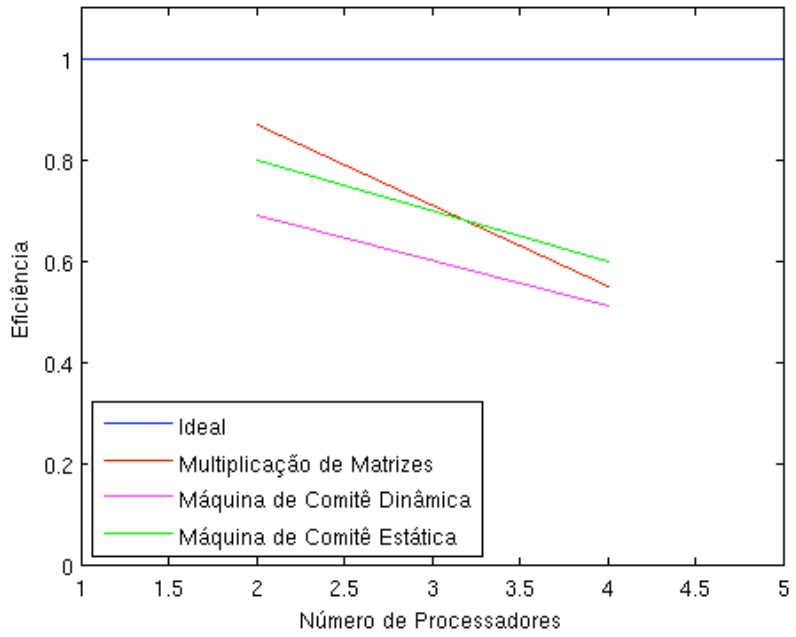

Figura 5.18 – Eficiência dos algoritmos executados no *Cyclone*® *I*, comparados com o ideal.

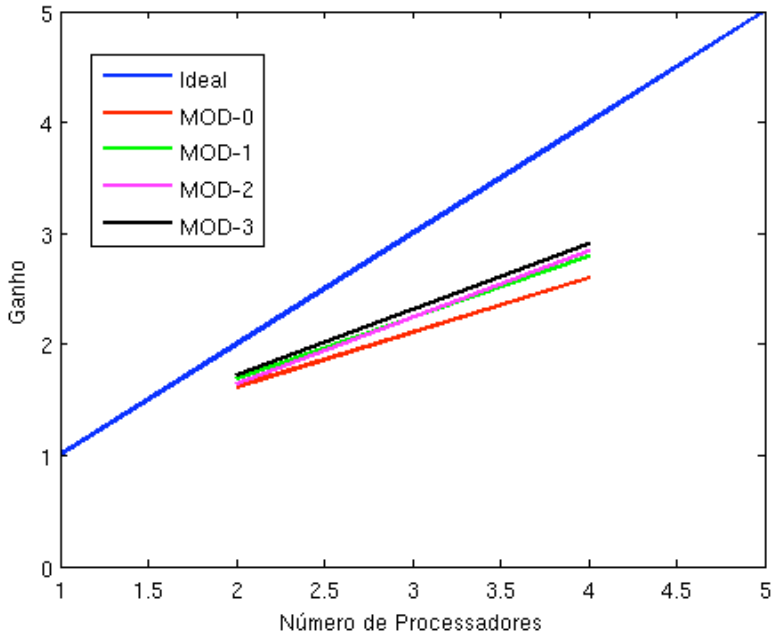

Figura 5.19 – Ganho das redes modulares no *Cyclone*® *III,* comparados com o ideal.

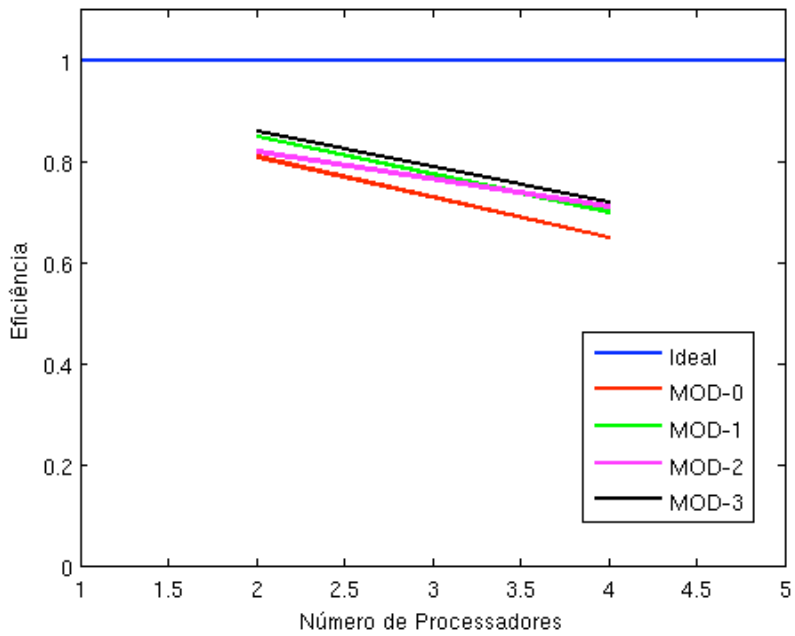

Figura 5.20 – Eficiência das redes modulares no *Cyclone® III*, comparados com o ideal.

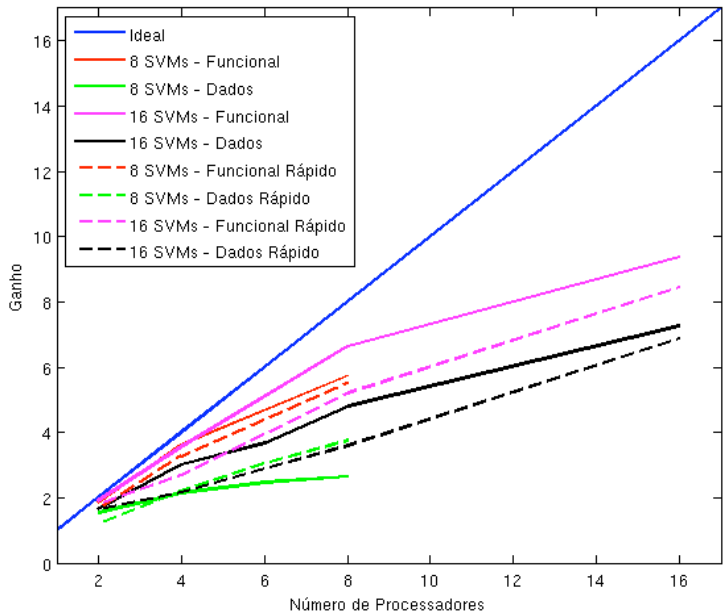

Figura 5.21 – Ganho dos *ensembles de SVM* no *Cyclone® III,* comparados com o ideal.

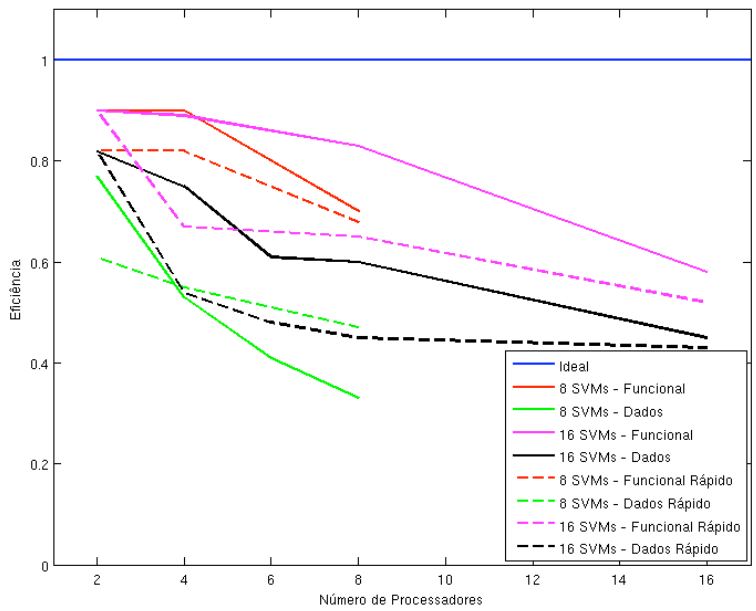

Figura 5.22 – Eficiência dos *ensembles de SVM* no *Cyclone® III,* comparados com o ideal.

Na literatura sobre processamento paralelo é mostrado que um ganho de desempenho e uma eficiência ideal são difíceis de serem alcançados, devido a custos de comunicação, sincronismo e ociosidade do processador. Entretanto, mesmo com os valores diferentes dos ideais, os resultados obtidos foram satisfatórios.

Utilizando o FPGA foi possível obter a mesma taxa de classificação alcançada nos com sistemas com processadores de uso geral. Além disso o FPGA apresenta uma potência reduzida quando comparada a outras unidades de processamento, característica importante em sistemas.

Capítulo 6

Considerações finais e trabalhos futuros

Apresentou-se neste trabalho uma nova abordagem para implementação de arquiteturas complexas de redes neurais em FPGA, utilizando vários processadores em um único *chip*. Foram analisados trabalhos relacionados, e foi visto que a maior dificuldade na utilização de redes neurais em FPGA era a dificuldade de implementação, modificação dos algoritmos implementados e a restrição de recursos disponíveis nessa plataforma. Como alternativa a essa restrição pesquisadores têm feito uso de processadores descritos em *software*, como o Nios® II, para implementar suas redes neurais em FPGA. Entretanto, os trabalhos observados eram limitados ao desenvolvimento de redes neurais menos complexas, com poucos neurônios e poucas camadas, devido o desempenho modesto apresentado por esses processadores. Para contornar esse problema, foram utilizadas técnicas de processamento paralelo desenvolvendo algoritmos paralelos para plataformas com múltiplos processadores, aumentando o desempenho do sistema.

Os trabalhos existentes que utilizavam múltiplos Nios® II, apenas citavam que era realizado a comunicação sem maiores detalhes. Nesse trabalho foi investigado as duas formas possíveis de comunicação, detalhando e realizando experimentos para definir qual dessas era mais vantajosa em ser utilizada. Para isso foi implementado uma multiplicação de matrizes, *benchmark* conhecido na computação paralela para analisar o desempenho da comunicação em arquiteturas paralelas.

Após a definição de qual forma de comunicação seria utilizada, foram desenvolvidos algoritmos de redes neurais em paralelo, para utilização na arquitetura paralela desenvolvida. Com a implementação desses algoritmos foi possível observar ganhos alcançados através da utilização do paralelismo. Foram implementados redes neurais mais complexas, que as encontradas em trabalhos relacionados. Enquanto, nos trabalhos observados foram encontrados neurônios implementados fisicamente e MLP implementadas em Nios® II, neste trabalho foram implementadas máquinas de comitê com MLPs e SVMs, que são arquiteturas de redes neurais com estruturas mais

complexas que as citadas. Nos experimentos, o ganho obtido em alguns casos foi maior que cem por cento, viabilizando a implementação dessas redes nestes tipos de sistemas.

A quantidade de blocos lógicos existentes no FPGA é quem limita a quantidade de processadores no sistema. Pode-se pensar em trabalhos futuros a utilização de *clusters* de FPGA, para agregar ainda mais desempenho ao sistema, já que é possível, por exemplo, realizar uma comunicação *ethernet* utilizando esses dispositivos.

Uma outra sugestão de trabalho futuro é desenvolver um *hardware*, ou dedicar um processador para receber informações de sensores em tempo real para realizar o processamento dos dados, ao invés de utilizar dados armazenados ou obtidos de simulações.

Para comparar o desempenho da arquitetura com uma implementação física, pode-se realizar a implementação em linguagem de descrição de *hardware* das redes neurais implementadas. Pode-se pensar no desenvolvimento de um *IP Core* com a função de um neurônio e que o mesmo pode-se ser reutilizado para tentar diminuir a quantidade de elementos lógicos necessários para a implementação física da rede neural. Partindo dessa idéia, como seqüência desse trabalho além do desenvolvimento dos *IPs*, pode-se pensar em formas de como a rede neural que utilize esses *IPs* possa ter seus parâmetros configurados dinamicamente.

Referências Bibliográficas

Altera Apex (2009) *Comparing Altera® APEX 20KE & Xilinx Virtex-E Logic Densities*. Disponível em: http://www.altera.com/products/devices/apex/features/apx-compdensity.html

Altera Avalon (2009) *Avalon Interface Specifications*. Disponível em: http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

Altera Mailbox (2009) *Mailbox Core*. Disponível em: http://www.altera.com/literature/hb/nios2/n2cpu_nii53001.pdf

Altera Mutex (2009) *Mutex Core*. Disponível em: http://www.altera.com/literature/hb/nios2/n2cpu_nii51020.pdf

Altera Simultaneous (2009) *Multi-Mastering with the Avalon Bus*. Disponível em : http://www.altera.com/literature/an/an184.pdf

Altera Nios II (2009) *Literature: Nios II Processor*. Disponível em: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

Alves, R. L. de S. (2002) *Uma Contribuição ao treinamento de Perceptrons de Múltiplas Camadas usando Retropropagação Competitiva e Processamento Paralelo*. Dissertação de Mestrado, UFRN, Natal – RN.

Amd (2009). Disponível em http://www.amd.com

Amdahl, G. (1967) *Validity of the single processor approach to achieving large-scale computing capabilities*, em Processinngs of AFIPS Conference, Pp.: 483 – 485.

Anderson, C. W; Hong, Z. (1994) *Reinforcement Learning with Modular Neural Network Control*, em IEEE International Workshop on Neural Networks Applied to Control and Image Processing NNACIP'94.

Arribas, P.C; Macia, F. M. H. (2002) *FPGA board for real time vision development systems*, em 4th IEEE International Caracas Conference on Devices, Circuits and Systems (ICCDCS'02) – Aruba Pp**:** T021-1 - T021-6.

Benini, L; Micheli, G. (2002) *Networks on Chips: A New SoC Paradigm*, em Computer, Volume: 35, Issue: 1, Janeiro. Pp: 70 – 78.

Benchmark Repository, Disponível em: http://ida.first.fraunhofer.de/projects/bench/benchmarks.htm

Byte Craft Limited. (2002) *First Steps with Embedded Systems*. Editora: Byter Craft, Ontário – Canadá, 228p. Disponível em http://www.bytecraft.com.

Cândido, C. K. S. S. (2008) *Classificação de distúrbios na rede elétrica usando redes neurais e wavelets*, Tese de Doutorado, UFRN, Natal – RN.

Castro, E. O. (2007) *Multiprocessador em Eletrônica Reconfigurável para Aplicações Robóticas*. Dissertação de Mestrado, UNICAMP, Campinas – SP.

Cao, B; Chang, L; Li, H. *Implementation of the RBF Neural Network On a SOPC for Maximum Power Point Tracking*, em 21th IEEE Canadian Conference on Electrical and Computer Engineering. Niagra Falls, Canadá. PP.: 981 – 986.

Cofer, R. C; Harding, B. F. *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Embedded Technology Series. Editora: *Newnes*, 320p.

Dimitrakakis, C; Bengio, S. (2005) *Online adaptive policies for ensemble classifiers*. em Neurocomputing, (64):211–221.

Flynn, M. J. (1966) *Very High-Speed Computing Systems*, em Proceedings of the IEEE, volume 54. Pp: 1901 – 1909.

Foster, I. (1995) *Designing and Building Parallel Programs – Concepts and tools for parallel software enginnering*. 1ª Edição, Editora: Addison-Wesley, 430p.

Frey, G; Litz, L. (2000) *Formal methods in PLC programming*, em IEEE International conference on Systems, Man, and Cybernetics – Nashville, TN, USA. Volume 4. Pp: 2431 – 2436.

Furht, B. (1997) *Processor Architectures for Multimedia: A Survey*. invited paper, Proceedings of Multimedia Modeling Conference, World Scientific, Singapore, November 1997, pp. 89-109.

Ganeshamoorthy, K; Ranasinghe, D. N. (2008) *Implementations on Distributed Memory Architectures*, em IEEE International symposium on Cluster Computing and the Grid. Lyon – France, Pp. 90 – 97.

Geer, D. (2005) *Chip Makers Turn to Multicore Processors*, em Proceedings of the IEEE Computer Society, volume 38, issue 5. Pp: 11 – 13.

Gil, D; Soriano, A; Ruiz, D; Montejo, C. A. (2007) *Embedded System for Diagnosing Dysfunctions in the Lower Urinary Tract*, em ACM symposium on Applied computing (SAC'07), Seoul – Korea, Pp. 1695 – 1699.

Gorder, P. F. (2007) *Multicore Processors for Science & Engineering*, em Proceedings of the IEEE Computer Science and Engineering, volume 9, issue 2. Pp: 3 – 7.

Gustafson, J. L. (1988) *Reevaluating Amdahl's law*, em Communications of the ACM, New York, NY, USA. Volume 31, number 5. Pp. 532 – 533.

Haykin, S. (2001) *Redes Neurais. Princípios e prática*. 2ª edição, Porto Alegre. Editora Bookman, 900p.

Huerta, P; Castillo, J; Martínez, J. I; Lopez, V. (2005) *A MicroBlaze based MultiProcessor SoC*, em World Scientific and Engineering Academy and Society (WSEAS'05) Transactions on Circuits and Systems. Pp: 423 – 430.

Huerta, P; Castillo, J; Martínez, J. I; Pedraza, C. (2007) *Exploring FPGA Capabilities for building Symmetric Multiprocessor Systems*, em 3th Southern Conference on Programmable Logic (SPL'07) – Mar Del Plata, Argentina. PP. 113 – 118.

Hwang, K; Briggs F.A. (1985) *Computer Architecture and Parallel Processing*. Editora McGraw-Hill, 1985, 846p. Série: McGraw-Hill series in computer organization and architecture.

Intel (2009). Disponível em http://www.intel.com

Jerraya, A. A.; Wolf, W. (2005) *Multiprocessor Systems-on-Chip*. Editora Morgan Kaufmann, 2005, 581p.

Kwon, S; Davis, J; Lynch, M; Prokop, M; Ruggles, S; Torrez P. (2007) *Gain scheduled Neural Network Tuned PI Feedback Control System for the Lansce Accelerator*, em IEEE Particle Accelerator Conference (PAC'07) Alburquerque, USA. Pp.: 2379 – 2381.

Lim, H; You, K; S, W. (2006) *Design and Implementation of Speech Recognition on a Softcore Based Fpga*, em IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP'06) – Toulouse, France. Pp. 1044 – 1047.

Lima, N. H. C; Neto, A. D. D; Melo, J. D. (2009) *Creating an Ensemble of Diverse Support Vector Machines using Adaboost,* em International Joint Conference on Neural Network (IJCNN'09). Atlanta, USA. PP.:

Lee, G. H; Kim, S. S; Jung S (2008). *Hardware Implementation of a RBF Neural Network Controller with a DSP 2812 and an FPGA for Controlling Nonlinear Systems*, em International Conference on Smart Manufacturing Application (ICSMA'08), Pp: 167 – 171.

Magalhães, R. M. (2007) *Uma investigação sobre a utilização de processamento paralelo no projeto de máquinas de comitê*. Dissertação de Mestrado, UFRN, Natal – RN.

Magalhães, R. M; Santos C. K. S; Melo J. D; Medeiros M. F; Neto A. D. D. (2008) *Application of a hybrid Algorithm in the Modular Neural Nets Trainning with Multilayers Specialists in Electric Disturbance Classification*, em Proceedings of International Conference on Intelligent Engineering Systems, pp. 121-125.

Medeiros Jr, M. F; Santos, C. K. S; Oliveira, J. T; Pires, S. M; Melo, J. D; Neto, A. D. D; Leitão, J. J. A. L. (2007) *Influence of Signal Pre-Processing in the Efficiency of Algorithms Based on Neural Networks Disturbance Classification*, em Computacional Intelligence in Image and Signal Processing. Pp.: 95 – 100.

Mendel, J. M; McLaren R. W. (1970) *Reinforcement Learning control and pattern recognition systems*, em Adaptative Learning and Pattern Recognition Systems: Theory and applications, volume 66. Pp.: 287 – 318.

Muthuramalingam, A; Himavathi, S; Srinivasan, E. (2008) *Neural Network Implementation Using FPGA: Issues and Application*, em International Journal of Information Technology. Volume 4, number 2. Pp. 86 – 92.

Niu, J; He R; Hu, J. (2005) *MPEG-4 Video Encoder Based on DSP-FPGA Techniques*, em International Conference on Communications, Circuits and Systems (ICCCAS'05) – Hong Kong, China. Pp: 518 – 522 Vol. 1.

Omondi, A. R; Rajapakse, J. C. (2006). *FPGA Implementations of Neural Networks*. Editora: Springer, 360p.

OpenSPARC (2008) *Internals.* Livro disponível em: http://www.opensparc.net/publications/books/opensparc-internals.html

OpenRISC (2009) disponível em http://sourceware.org/cgen/gen-doc/openrisc.html

Ou, J; Prasanna, V. K. (2006) *Design Space Exploration Using Arithmetic-Level Hardware Software Cosimulation for Configurable Multiprocessor Platforms*. ACM Transactions on Embedded Computing Systems V.5, Nº2. Pp: 355 – 382.

Oleskovicz, M. (2004) *Apostila de Qualidade de Energia*. Escola de Engenharia de São Carlos, Sp.

Ouelette, M; Connors, D. (2005) *Analysis of Hardware Acceleration in Reconfigurable Embedded Systems*, em 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) – Denver, Colorado, USA. Pp: 168 – 171.

Plavec, F; Fort, B; Vranesic, Z.G; Brown, S.D. (2005) *Experiences with Soft-Core Processor Design*, em 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) – Denver, Colorado, USA. Pp: 164 – 167.

Pitanga, Marcos. (2008) *Construindo Supercomputadores com Linux*. Terceira Edição, Rio de Janeiro: Editora Brasport, 374p.

Rose, C. A. F; Navaux, P. O. A. (2003) *Arquiteturas Paralelas*. Primeira edição, Porto Alegre: Editora Sagra Luzzato. 152p. Série: Livros Didáticos do Instituto de Informática da UFRGS.

Shiqui, H; Hong, Y; Jian, Z; Daizhi L (2004). *Application of DSP Parallel Processing Technology in SAR Imaging*, em 10th International Conference on Signal Processing (ICSP'04) – Beijing, China. Pp: 2580 – 2583.

Stallings, W. (2002) *Arquitetura e Organização de Computadores*. 5ª edição, Editora: Pearson Prentice-Hall, 786p.

Sutton, R. S; Barto, A. G. (1998) *Reinforcement Learning An Introduction*. *Adaptive Computation and Machine Learning*. Londres. Editora: MIT Press, 322p

Tanembaum, A. S. (2007) *Organização Estrutura de Computadores*. 5ª edição, Editora: Pearson Prentice-Hall, 449p.

Tanembaum, A. S. (1999) *Sistemas Operacionais, Projeto e Implementação*. 2ª edição, Editora: Bookman, 759p.

Temurtas, Feyzullah (2007). *A comparative study on thyroid disease diagnosis using neural networks*, em Journal Expert Systems with Applications, volume 36, issue 1, pp: 944 – 949.

Taright, Y; Hubin, M. (1998) *FPGA Implementation of a Multilayer perceptron Neural Network using VHDL*, em 4th International Conference on Signal Processing (ICSP'98) – Beijing, China. Pp: 1311 – 1314 volume 2.

Texas (2009). Disponível em http://www.ti.com

Tocci, R. J. Widmer, N. S. Moss, G. L. (2008) *Sistemas Digitais, princípios e aplicações*. 10ª edição, Editora Pearson Prentice-Hall, 805p.

Tong, J.G; Khalid, M.A.S. (2007) *A Comparison of Profiling Tools for FPGA-Based Embedded Systems*, em Canadian Conference on Electrical and Computer Engineering (CCECE'07), pp. 1687-1690, 22-26 April 2007.

Vapnik, V. N. (1992) *Principles of risk minimization for learning theory*. Advances in Neural Information Processing Systems, em Volume 04, pp.: 831 – 838.

Yiannacouras, P; Rose, J; Steffan, J. G. (2005) *The Microarchitecture of FPGA-Based Soft Processors*, em International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05) – San Francisco, California, USA. Pp: 202 - 212.

Waldeck, P; Bergmann, N (2004). *Evaluating Software and Hardware Implementations of Signal-Processing Tasks in an FPGA*, em IEEE International Conference on Field-Programmable Technology (ICFPT'04) Queensland, Australia. Pp: 299 – 302.

Wang J; Chen Y; Xie J; Chen B, Zhou Z. (2008) *FPGA based Neural Network PID Controller for Line-scan Camera in Sensorless Environment*, em International Conference on Natural Computation (ICNC'08). Jinan, China. Pp.: 157 – 161.

Wei, Y; Bing, X; Chareonsak, C. (2004). *FPGA implementation of AdaBoost algorithm for detection of face biometrics,* em IEEE International Workshop Biomedical Circuits and Systems (BioCAS'04). Singapore. Pp: 17 – 20.

Wolf, W. (2003) *How Many System Architectures?* em Proceedings of the IEEE Computer Society, volume 36, issue 3. Pp: 93 – 95.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas