UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ELETRÔNICA

Sistema de Classificação Baseado em uma Máquina com processamento distribuído

---

Autor: André Rabello dos Anjos

Orientador: Prof. José Manoel de Seixas

Examinadora: Prof$^{\underline{a}}$. Marta Mattoso

Examinador: Prof. Luiz Wagner Pereira Biscainho

DEL
Setembro de 1997

Resumo

Na busca de novos canais físicos em experimentos com partículas colididas, sistemas de validação têm se mostrado de grande valia. Normalmente os subprodutos de colisões interparticulares representam física ordinária e conhecida enquanto que nova física aparece camuflada neste meio. A impossibilidade de gravação e análise do imenso volume de dados produzido nestes ambientes exige o uso de sistemas de validação para que se maximize o espaço de gravação e se minimize o espaço de procura de novos fenômenos.

Em particular, no CERN, o par acelerador/colisionador do LHC, que estará operacional no ano de 2005, utilizará um destes sistemas de validação baseado em 3 níveis em cascata de complexidade crescente e velocidade decrescente. Este sistema tem por objetivo a análise e filtragem, em tempo real, de um volume de dados cuja taxa chega à impressionante faixa de 100.000.000 por segundo.

A divisão do sistema em 3 etapas distintas visa produzir um sistema de validação o mais eficiente e dinâmico possível, sem que se sobrecarregue nenhuma das partes. Para o primeiro nível estima-se a utilização de processadores velozes, com nível baixo de programação, capazes de suportar a taxa inicial dos eventos. Para o terceiro nível o uso de pesado ambiente computacional é previsto.

No segundo nível ambientes altamente programáveis serão combinados com técnicas de paralelização de aplicações para que atinjamos a taxa de processamento requerida de 100.000 eventos por segundo.

Vários tipos de tecnologia estão sendo testadas em todo o mundo para que se decida, não somente sobre a arquitetura, mas, também, sobre o tipo de equipamento a ser empregado neste extenso sistema de classificação.

Este trabalho é sobre a implementação em uma máquina com processamento distribuído de uma das arquiteturas previstas para o segundo nível de validação (ou classificação) do experimento ATLAS-/LHC. A máquina em questão é um sistema Telmat TN310 com processamento distribuído por 16 nós padrão HTRAM totalmente conectados através de uma rede de chaves assíncronas. A arquitetura mencionada prevê a utilização de técnicas de paralelismo de dados e fluxo na obtenção de menores tempos de processamento.

O objetivo final é entender se o processamento em sistemas semelhantes a uma TN310 (visamos o tipo de nó-de-processamento e o padrão de conexão entre estes) pode ser viável para o segundo nível de validação. Isto se dará através da análise e capacidade de abstração proporcionadas pelo desenvolvimento da aplicação sugerida no equipamento.

Soma-se ao trabalho o desenvolvimento de uma unidade de decisões globais baseado em redes neurais. A unidade constitui processo central do sistema de validação. Resultados atingidos são expostos e discussões sobre técnicas de implementação são realizadas no decorrer da documentação.

Sistema de Classificação Baseado em uma Máquina com Processamento Distribuído

André Rabello dos Anjos

25 de março de 2007

Sumário

1 Introdução **7**
1.1 Física de Partículas, o que é? . . . 7
1.1.1 O bóson de Higgs, a caminho da massa zero . . . 7
1.1.2 A teoria do Big-Bang . . . 7
1.1.3 Aceleradores . . . 9
1.2 O CERN(Laboratório Europeu para Física de Partículas) . . . 10
1.2.1 O LHC (*Large Hadron Collider*) . . . 12
1.3 Sistemas de Validação . . . 16
1.3.1 Porque utilizar sistemas de validação? . . . 16
1.3.2 O sistema de *Trigger* para o experimento LHC/ATLAS . . . 16
1.3.3 O Primeiro Nível de *Trigger* . . . 17
1.3.4 O Segundo Nível de *Trigger* . . . 18
1.3.5 O Terceiro Nível de *Trigger* . . . 20
1.3.6 Compondo o sistema de validação para o experimento LHC/ATLAS . . . 21
1.4 Próximos capítulos . . . 21

2 Revisão da Literatura **23**
2.1 Redes Neurais Artificiais e o processamento global para o segundo nível de *trigger* . . 23
2.2 A utilização de ambientes de processamento distribuído para o segundo nível de *trigger* 25
2.2.1 DSP-s . . . 27
2.3 Arquiteturas para o segundo nível de *trigger* . . . 27
2.3.1 Modelo A . . . 28
2.3.2 Modelo B . . . 28
2.3.3 Modelo C . . . 28
2.4 Concluindo alguns pontos... . . . 28

3 Fundamentos Teóricos, Métodos e Materiais **33**
3.1 Redes Neurais Artificiais . . . 33
3.1.1 O Neurônio Artificial . . . 34
3.1.2 Formando Redes Neurais . . . 36
3.1.3 Treinamento de redes de múltiplas camadas - retropropagação . . . 38
3.1.4 Redes Neurais e aspectos gerais . . . 40
3.2 Computação distribuída e o sistema TN310 . . . 40
3.2.1 Terminologia . . . 41
3.2.2 Processamento Distribuído - uma solução à demanda de velocidade . . . 41
3.2.3 Arquiteturas Paralelas . . . 42
3.2.4 O Sistema TN310 . . . 45
3.2.5 Desenvolvendo aplicações para a TN310 . . . 49
3.2.6 Resumindo a seção. 54
3.3 Técnicas de Paralelismo . . . 56

3.3.1 Paralelismo de Controle . . . 57
3.3.2 Paralelismo de Dados . . . 57
3.3.3 Paralelismo de Fluxo . . . 59
3.3.4 O que usar na aplicação? . . . 60
3.4 Os dados utilizados neste projeto . . . 61

4 Implementando os algoritmos **62**
4.1 A Unidade de Decisão Global (GDU) - Identificando a RoI . . . 62
4.1.1 A Implementação da GDU em 1 nó . . . 62
4.1.2 A implementação em até 16 nós . . . 65
4.2 Implementação do segundo nível de validação . . . 70
4.2.1 Simulando o segundo nível . . . 73

5 Resultados obtidos e Conclusões **87**
5.1 Teste de comunicações . . . 87
5.1.1 Introdução ao sistema de comunicações . . . 87
5.1.2 Os testes . . . 88
5.2 Resultados para a GDU . . . 89
5.2.1 Replicando o JETNET . . . 89
5.2.2 A GDU concorrente . . . 91
5.2.3 Conclusões sobre a unidade de decisões globais . . . 92
5.3 Resultados para a implementação do segundo nível de *trigger* . . . 92
5.3.1 Resultados para o algoritmo completo rodando em apenas 1 processador . . . 92
5.3.2 Resultados para a versão 1 . . . 93
5.3.3 Resultados para a versão 2 . . . 93
5.3.4 Resultados para a versão 3 . . . 94
5.3.5 Concluindo sobre o sistema de validação . . . 94

6 Discussões **102**
6.1 Usando os DSP-s *on-board* . . . 102
6.2 Fundindo aplicações e aumentando o número de processos por nó . . . 103
6.3 Anexando processos . . . 103

A Exemplo comentado de um arquivo de descrição de redes **107**

B Exemplo comentado de arquivo CFS **114**

C Exemplo de arquivo `Makefile` **117**

D GDU em InMOS C para 1 nó HTRAM **119**

E O supervisor para GDU rodando em 16 processadores **126**

F Versão final do simulador **134**
F.1 Arquivo CFS (`level2.cfs`) . . . 134
F.2 Supervisor (`master.c`) . . . 137
F.3 Extratores de característica . . . 151
F.3.1 Calorímetro (`time_cal.c`) . . . 151
F.3.2 TRT (`time_trt.c`) . . . 153
F.3.3 SCT (`time_sct.c`) . . . 153
F.3.4 Múon (`time_muo.c`) . . . 153
F.4 Rede Local . . . 153
F.4.1 LN0 (`ln0.c`) . . . 153

F.4.2 LN1 (`ln1.c`) 157
F.5 Decisão Global (`global.c`) 161

Lista de Figuras

1.1 O mundo subatômico conhecido. 8
1.2 A formação do universo, depois do Big-Bang 9
1.3 Uma fotografia do túnel subterrâneo de 3,8 metros de largura por onde passam os canhões de alumínio do LEP (à direita e embaixo). 11
1.4 A localização dos anéis (foto aérea) do LEP/LHC 12
1.5 Os diversos equipamentos do CERN, em escala, e suas conexões. 13
1.6 O detetor ATLAS. 14
1.7 Um evento reconstruído 15
1.8 Um esquema simplificado para o primeiro nível de *trigger* do experimento LHC/ATLAS. 17
1.9 Uma ilustração sobre a idéia de mapeamento da parede de um detetor. 19
1.10 A arquitetura para o sistema de validação do experimento ATLAS/LHC. 22

2.1 A distribuição de características para cada subdetetor genérico. Os subsistemas são LI-s. 24
2.2 Exemplo de classificação num espaço bidimensional 24
2.3 Classificação em um espaço de variáveis linearmente dependente. 26
2.4 Um diagrama esquemático para o modelo arquitetural A do segundo nível de validação. 29
2.5 Um diagrama esquemático para o modelo arquitetural B do segundo nível de validação. 30
2.6 Um diagrama esquemático para o modelo arquitetural C do segundo nível de validação. 31

3.1 O desenho mostra as diversas partes de um neurônio 34
3.2 O desenho mostra a transmissão de sinais entre neurônios por intermédio de sinapses. 35
3.3 Um neurônio artificial. 35
3.4 Um neurônio artificial com função de ativação. 36
3.5 Uma rede neural com uma única camada. 37
3.6 Uma ANN com mais de uma camada. 38
3.7 A arquitetura sequencial ou SISD 42
3.8 Esquema representativo de uma arquitetura MIMD com memória global. 43
3.9 Esquema representativo de uma arquitetura MIMD com memória distribuída. 44
3.10 A passagem de dados em um sistema com memória global. 44
3.11 A passagem de dados(por chaves) em um sistema com memória distribuída. 45
3.12 Um diagrama de uma aplicação *master-slave* 46
3.13 Algumas configurações topológicas de redes com memória distribuída que não utilizam chaves. 46
3.14 A arquitetura básica do *Transputer* T9000. 48
3.15 TN310, conexão dos diversos nós HTRAM (dados recolhidos pela ferramenta T9SPY). 50
3.16 As diversas camadas de programação disponíveis no *T9000 Toolset*. 51
3.17 Esquema de uma comunicação entre tarefas utilizando canais. 55
3.18 O paralelismo de controle num diagrama de blocos. 58
3.19 Um diagrama mostrando a paralelização de uma aplicação utilizando a técnica de paralelismo de dados. 59

3.20 Um *pipe* de dados. O paralelismo de fluxo é aconselhável aqui. 60

4.1 O neurônio utilizado na rede da figura 4.2. 64
4.2 A rede para a GDU implementada na TN310. 64
4.3 Exemplo de *Look-up*. 66
4.4 Um esquema da GDU rodando em paralelo, com 16 nós de processamento. 67
4.5 O fluxo de atividades do processo supervisor. 68
4.6 Um supervisor trabalhando sob-demanda em uma configuração *master-slave*. 68
4.7 A comunicação entre o supervisor e uma GDU, com (a) e sem (b) *hand-shake*. 69
4.8 As partes a serem implementadas na TN310 do segundo nível de *trigger* do experimento ATLAS/LHC. 71
4.9 Diagrama da implementação genérica do sistema de validação da figura 4.8 no sistema TN310. 72
4.10 O diagrama da aplicação a ser simulada na TN310. 74
4.11 Fluxograma do processo supervisor para a simulação do segundo nível de *trigger*. . . . 75
4.12 As instruções-chave que implementam a inicialização do sistema pelo supervisor. . . . 76
4.13 As rotinas de envio de dados para os extratores. 79
4.14 O diagrama mostra a alocação de processos e a "distância" entre estes na primeira versão do programa simulador. 84
4.15 O diagrama mostra a alocação de processos e a "distância" entre estes na segunda versão do programa simulador. 85

5.1 O processo de deleção de cabeçalho em uma rede com chaveamento assíncrono de pacotes. 88
5.2 O diagrama da aplicação implementada no sistema TN310. 95
5.3 As tarefas da aplicação. 96

B.1 A topologia de conexões entre as tarefas da aplicação *PIPELINE*. 114

Lista de Tabelas

2.1 Tabela de eficiências para uma classificação utilizando um “E” lógico e Redes Neurais Artificiais simples, sem camadas escondidas. 24

3.1 As bibliotecas não-ANSI no InMOS C ToolSet (Retirado de [18]). 53
3.2 Algumas das funções e tipos em `channel.h` (Retirado de [18]). 56

4.1 As eficiências (em %) obtidas durante a fase de teste de treinamento da unidade de decisões globais (JETNET). 63
4.2 Os dados enviados a cada vez para os extratores de característica. Valores em bytes. . 77
4.3 O cabeçalho enviado junto com cada RoI. Valores em bytes. 77
4.4 Tempos de processamento simulados para cada tipo de extrator. 80
4.5 Os dados externalizados pelos extratores de característica para LN0 (em bytes). 80
4.6 A alocação de processos na primeira versão do programa simulador. 83
4.7 Uma proposição mais eficiente para para a alocação das tarefas na TN310. 86

5.1 Os tempos gastos para os diversos tamanhos de dados comunicados a diferentes nós de processamento. 90
5.2 As eficiências para a rede 12-6-4 da GDU usando InMOS C. 91
5.3 A GDU concorrente, versão 1 - Mapa de alocação de processos. 91
5.4 A GDU concorrente, versão 2 - Mapa de alocação de processos. 91
5.5 Os resultados obtidos com a implementação da primeira versão do simulador para o sistema de validação. Totais para 500 RoI-s. 93
5.6 Os resultados obtidos com a implementação da segunda versão do simulador para o sistema de validação. Totais para 500 RoI-s. 94
5.7 Os resultados obtidos com a implementação da terceira versão do simulador para o sistema de validação. Totais para 500 RoI-s. 94
5.8 Os resultados para a GDU concorrente quando reduzimos o número de processos escravos. 99
5.9 Tempos gastos na passagem de dados da aplicação supervisora para os extratores. . . 99
5.10 O tempo gasto em processamento dos dados em cada tipo de extrator de característica. 99

Capítulo 1

Introdução

1.1 Física de Partículas, o que é?

A física de partículas é o estudo dos elementos básicos da matéria e das forças que atuam sobre estes. Ela planeja determinar as leis fundamentais que controlam a criação de matéria e do universo físico[1].

A estrutura do mundo subatômico, como conhecemos hoje, pode ser melhor entendido se voltarmos a atenção à figura 1.1.

Na figura identificamos que os átomos consistem de um núcleo pequeno, positivamente carregado, muito denso, e cercado por elétrons negativamente carregados presos a uma órbita por forças predominantemente coulombianas (eletromagnéticas). O núcleo é feito de Neutrons que não possuem carga (daí o nome) e prótons carregados positivamente. prótons e Neutrons são, por sua vez, formados por Quarks, que possuem carga de $-1/3$ ou $+2/3$. Existem seis diferentes tipos de Quarks na natureza ainda que prótons e Neutrons sejam formados pela combinação de apenas dois tipos *Up* e *Down*. Juntamente com elétrons, Quarks são os elementos construtores de todo tipo de matéria com a qual estamos familiarizados.[2]

1.1.1 O bóson de Higgs, a caminho da massa zero

Um bóson é, por definição, uma partícula carregadora de força. Como exemplo vemos os Fótons, partículas praticamente sem massa. A luz visível ou radiação eletromagnética qualquer pode ser pensada como sendo composta por Fótons.

Uma corrente do pensamento moderno na física de partículas busca para provar que a temperaturas altas o suficiente todas as forças fundamentais que conhecemos hoje tornam-se instâncias da manifestação de uma única força. Hoje, no universo frio em que vivemos, tanto a força eletromagnética(representadas por Fótons) como as Interações Fracas (representadas pelos Bósons W e Z[1]) possuem massa; isto restringe a força a um campo muito pequeno de atuação.

Rumo à confirmação das teorias sobre a formação do universo e as diversas facetas que compõem a formação da matéria, o Professor Peter Higgs modelou a existência de novas partículas, os Higgs, que seriam os equivalentes na formação da matéria ao papel desempenhado pelos fótons ao magnetismo. Sua massa[2] provável estaria em torno de 100 a 1000 GeV.

1.1.2 A teoria do Big-Bang

É pensado que o universo teve início por volta de 15 bilhões de anos atrás, numa grande explosão conhecida como Big-Bang e vem resfriando e expandindo desde então. Para físicos, o mais interessante

[1] Os Bósons W e Z possuem massa máxima de 90 GeV (aqui energia e massa se confundem já que Einstein previu que $E = mc^2$).

[2] Quando falamos em massa, não queremos aqui referenciar matéria, mas energia contida na partícula.

Figura 1.1: O mundo subatômico conhecido.

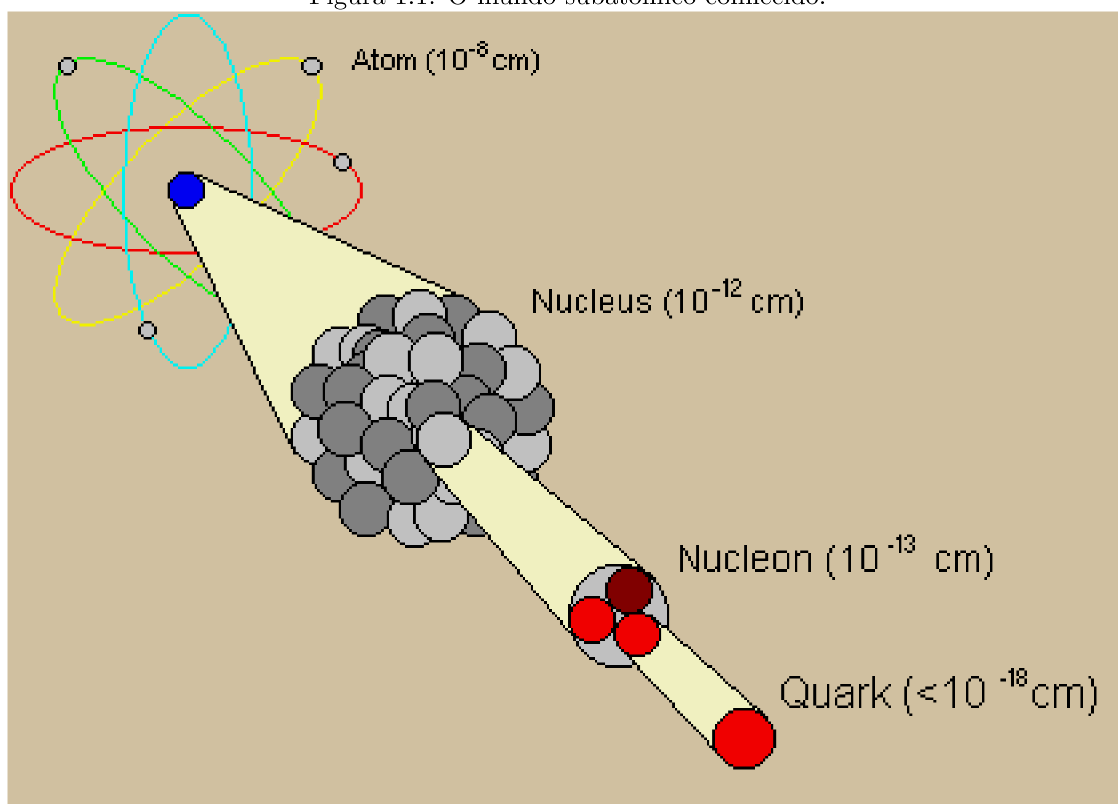
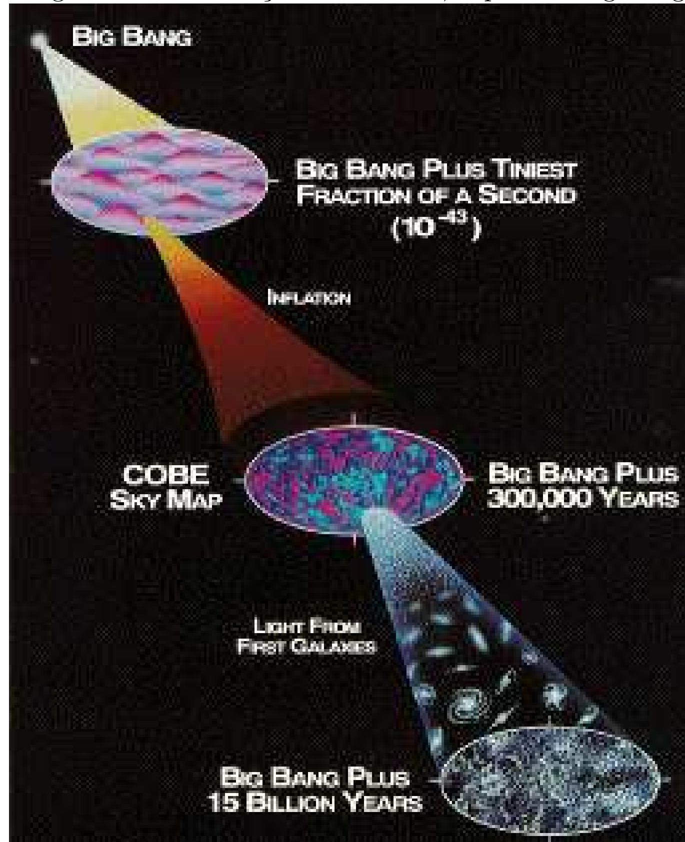


Figura 1.2: A formação do universo, depois do Big-Bang



momento foi durante os primeiros 10^{-34} segundos quando as condições eram tão extremas que as leis da física como as conhecemos hoje não poderiam ser aplicadas. Depois de aproximadamente 0,01 segundos o universo já estaria frio o suficiente para Quarks se juntarem formando prótons e Neutrons. Estes formariam o primeiro núcleo de Hélio depois de 100 segundos ainda que o primeiro átomo somente apareceria depois de 100.000 anos. Passados alguns bilhões de anos estrelas começariam a se formar utilizando o Hidrogênio e o Hélio para produzir elementos mais pesados que constituem o mundo a nossa volta, isto é, elementos mais pesados que o Hélio devem sua existência às estrelas. A figura 1.2 pode ser ilustrativa quanto à explanação anterior.

Pelo que se entende, chegar ao que foi o início dos tempos significa entender as diversas partes da matéria, como se correlacionam e quais as leis e forças que atuam sobre elas. Este estudo pode ser feito de diversas formas ainda que a mais utilizada seja a desintegração de núcleons ou outras partículas.

1.1.3 Aceleradores

A desintegração de núcleons pode ser atingida com o uso de equipamentos conhecidos como aceleradores/colisionadores de partículas. O par funciona de seguinte forma: grupos de partículas de número variante são acelerados a velocidades próximas a da luz quando são então colocados em rota de colisão com outro conjunto de partículas também aceleradas de forma que a energia contida no centro de massa da colisão seja máxima. As partícuas colidem. Durante a colisão nem todas as partículas destes "pacotes" são destruídas, mas as poucas que colidem, devido a grande energia que possuem(uma vez que foram super-aceleradas), quebram-se em partículas menores que por um breve instante de tempo

excitam o equipamento de deteção. Equipamentos de deteção devem ser utilizados em experimentos deste tipo, vista a impossibilidade de visualização a "olho nu" das colisões e decaimentos resultantes.

Os pacotes de partículas utilizados num experimento não têm necessariamente partículas iguais, podendo em alguns casos utilizar equivalentes anti-materiais.

Anti-matéria

No caso de aceleradores como o LEP (*Large Electron-Positron collider*) no CERN (*European Centre for Particle Physics*), a colisão é realizada utilizando-se diferentes tipos de "pacotes" de partículas, um de elétrons, e o outro de pósitrons. Pósitrons são o "anti-equivalente" de elétrons, i.e., formam o equivalente de elétrons no espaço da anti-matéria.

Anti-matéria é muito parecida com a matéria ordinária, mas carrega carga oposta. No momento do Big-Bang, matéria e anti-matéria, acredita-se, foram criadas em igual quantidade. O que parece que aconteceu foi que de alguma forma, mais tarde, colisões entre estes tipos tenham destruído quase toda a anti-matéria existente mas deixado um pouco de matéria para trás, da qual nosso universo é feito. Um dos motivos pode ser uma pequena assimetria entre as formas que matéria e anti-matéria decaem criando assim, um excesso de matéria.

Luminosidade em aceleradores

Um conceito muito utilizado em experimentos com aceleradores/colisionadores é o de *luminosidade*; ele é utilizado para medir a probabilidade de colisão entre partículas de um dado par acelerador/colisionador, isto é, quanto maior a colimação dos feixes no momento da colisão, maior a luminosidade. Mais informalmente, o conceito de luminosidade também está conectado à questão da probabilidade de encontrarmos partículas de dada dimensão num conjunto de partículas geradas após uma colisão. Luminosidade não será um conceito abordado por este trabalho, ainda que informação relevante possa ser encontrada em [3].

1.2 O CERN(Laboratório Europeu para Física de Partículas)

Os laboratórios do CERN hoje são dos mais bem equipados no mundo para experimentos com Física de Partículas. Dispondo de um dos maiores aceleradores do mundo[3], o LEP (figuras 1.3 e 1.4) encontra-se sobre a fronteira franco-suíça a cerca de 100 metros de profundidade, sendo que seu perímetro chega à impressionante marca de 27 Km. Várias das descobertas nessa área da física se devem à existência do CERN.

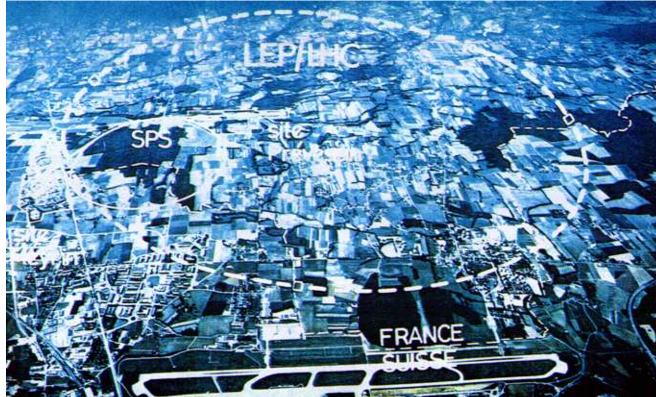
O complexo do CERN inclui também vários outros aparatos. Seja com a função de pré-aceleração ou produção do material a ser usado para colisão, estes equipamentos também possuem grandes dimensões, comparáveis à do próprio LEP. A figura 1.5 mostra o maquinário disponível no CERN, em tamanho reduzido, com suas interconexões. Na figura, vemos que os pacotes de pósitrons e elétrons seguem sob o mesmo tunel ainda que em direções opostas. Também identificamos os quatro pontos de colisão do acelerador onde se encontram os detetores OPAL, ALEPH, L3 e DELPHI.

Como explanado o LEP se tornou (e ainda é) útil na pesquisa de muitos dos pontos ainda obscuros na física de partículas, ainda que sua energia máxima de colisão no centro de massa não ultrapasse o pico de 100 GeV. O estudo de novos canais físicos como o Higgs exigirá a construção de um novo aparato (a massa de tal partícula poderá se encontrar na distante faixa 1 TeV). Rumo ao estudo de momentos anteriores aos 10^{-34} segundos do ínicio de nosso universo e de outros canais físicos ainda não desvendados, foi iniciada a construção do que será o maior instrumento científico do mundo, o LHC (Large Hadron Collider). O LHC tem início de operação previsto para o ano de 2005 e emergentes tecnologias têm sido utilizadas para suprir sua demanda de velocidade, precisão e custo.

[3]Sendo também o maior instrumento científico já construído

Figura 1.3: Uma fotografia do túnel subterrâneo de 3,8 metros de largura por onde passam os canhões de alumínio do LEP (à direita e embaixo).

Figura 1.4: A localização dos anéis (foto aérea) do LEP/LHC



1.2.1 O LHC (*Large Hadron Collider*)

O LHC será um versátil acelerador e poderá colidir prótons com energia em torno de 7 TeV perfazendo um total de impressionantes 14 TeV de energia no centro de massa. Isto permitirá que físicos e cientistas penetrem ainda mais na estrutura da matéria e recriem as condições prevalescentes somente 10^{-12} segundos depois do Big-Bang quando a temperatura era de $10^{16}\ ^oC$. Neste momento a quantidade de bósons de Higgs era abudante, e, sendo assim, estima-se que se possa estudá-los utilizando este novo aparato.

Uma vez que as interações entre partículas atômicas e sub-atômicas são invisíveis existe a necessidade da utilização de detetores de partículas. Detetores são instrumentos especiais que se excitam produzindo algum tipo de sinal ou dado quando partículas passam por dentro de seus corpos. Estes dados ou sinais têm, em geral, conotação direta com as partículas, podendo representá-las por meio de parâmetros, como energia, direção de movimento segundo um campo magnético, momento etc. É prática comum, no entanto, em experimento mais complexos, a utilização de superdetetores, que são conjuntos de detetores operando com funções dedicadas (como medição de energia, deteção de rota, deteção de momento, deteção de partículas específicas etc), aumentando a precisão dos dados recolhidos durante os experimentos.

Dois detetores, ATLAS e CMS, têm a responsabilidade de detetar as interações geradas na colisão dos pacotes de prótons. Em especial, os estudos com o superdetetor ATLAS não ficarão restritos à colisão entre prótons, estendendo-se à colisão iônica (com íons de Chumbo) que produzirão uma energia total próxima a 1150 TeV. A construção deste e do equipamento de suporte às colisões como o sistema de validação e filtragem tem se tornado um desafio para os cientistas e para indústrias de todo o mundo. Estes desafios vêm estimulando desenvolvimento de áreas tecnológicas de ponta como a de super-condutores, ultra-baixas temperaturas e de sistemas que operem em tempo real.

O ATLAS (*A ToroidaL ApparatuS*)

O detetor ATLAS será, na verdade, uma composição de vários subdetetores com funções e características próprias, são eles:

1. **Detetor Interno (*Inner Detector*)**- Este detetor é composto de vários sub-detetores, são eles:

    (a) SCT Frontal (*Foward SCT*)

    (b) SCT de Barreira (*Barrel SCT*)

    (c) TRT (*Transition Radiation Tracker*)

Figura 1.5: Os diversos equipamentos do CERN, em escala, e suas conexões.

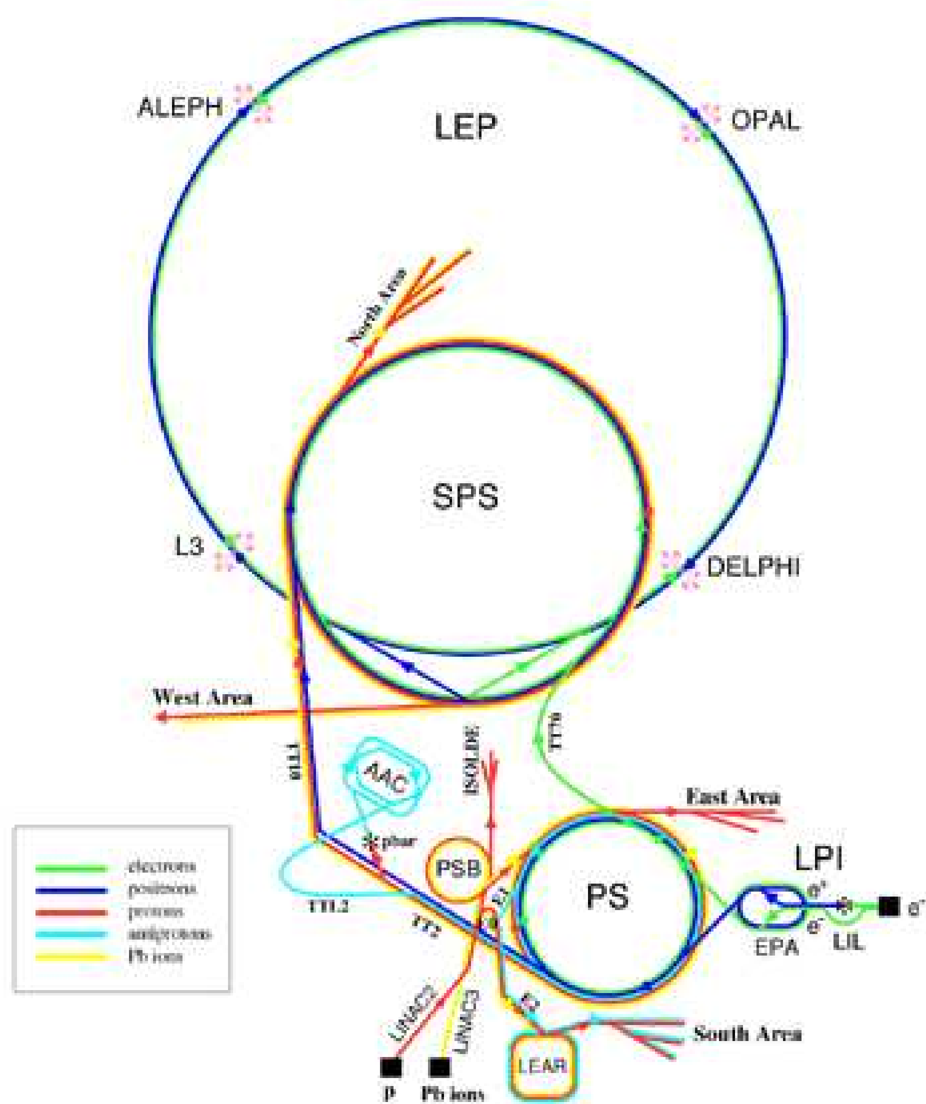
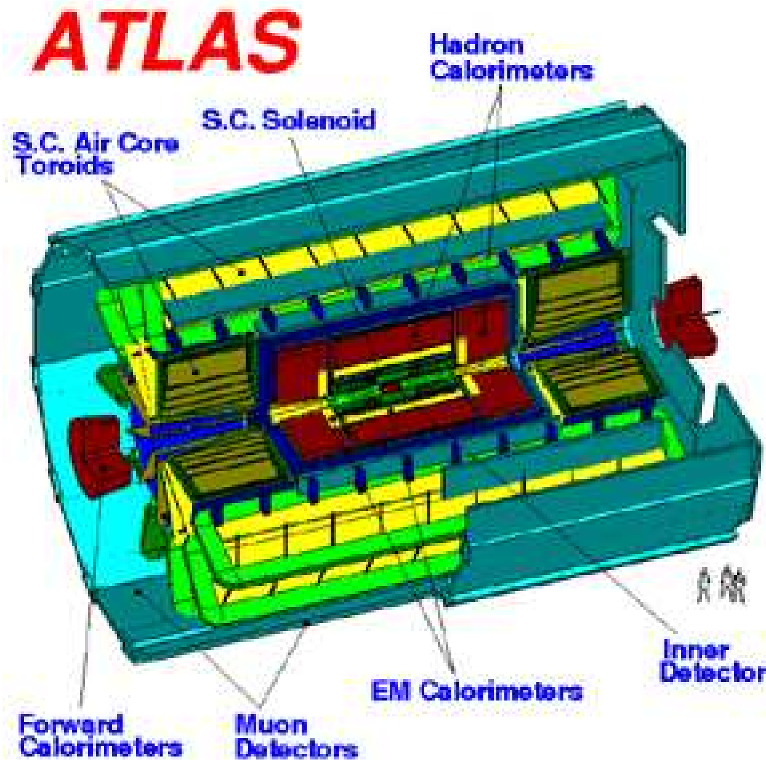


Figura 1.6: O detetor ATLAS. Repare os desenhos de um ser humano em escala, à direita e em baixo.



   (d) Detetores de "pixels" (*Pixel Detector*)

2. **Calorímetro de Argônio Líquido (*Liquid Argon Calorimeter*)**
3. **Calorímetro de telhas (*Tile Calorimeter*)**
4. **Esprectrômetro (ou Detetor) de Múons (*Muon Sprectrometer*)**

A figura 1.6 mostra um desenho de todo o detetor.

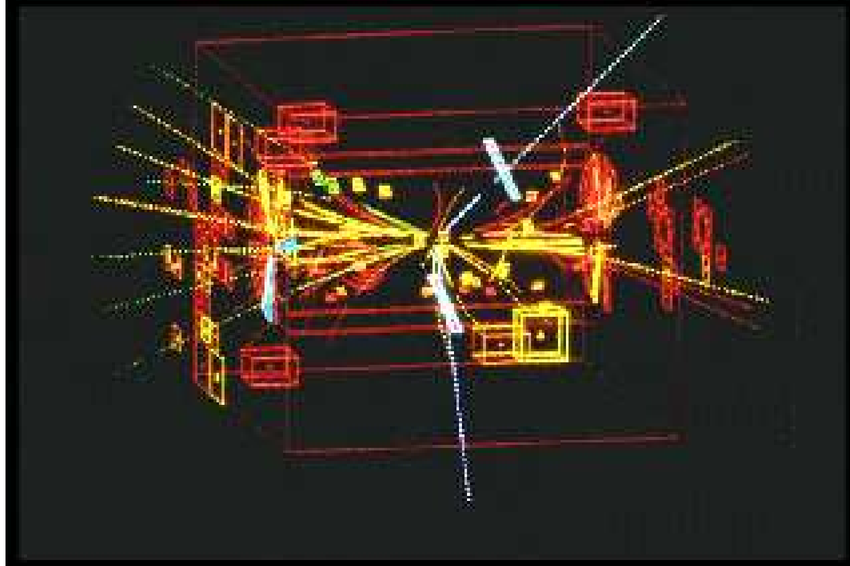
A forma como detetores são utilizados é, na realidade, bastante simples, diferentemente da complexidade da função que executam. Eles, os detetores, são colocados ao redor do ponto de colisão, de forma a cobrir todas as rotas possíveis de partículas expelidas da rota original pelo impacto[4]. Uma vez que haja colisão, o detetor é excitado produzindo um imenso volume da dados sobre as partículas geradas.

O que é um evento ? Definiremos evento como o possível acontecimento de um Higgs. Uma vez que Higgs são muito instáveis (lembre-se que em 10^{-34} segundo já existiam Quarks) eles rapidamente decaem em partículas menos energéticas e de maior massa (existe uma troca energia/massa bem visível aqui) que são as que realmente são detetáveis pelos aparelhos.

Higgs, então, não serão alvo de procura, mas sim sua forma "disfarçada", partículas provenientes de seu decaimento.

[4]É claro que fica difícil detetar partículas que colidiram elasticamente uma vez que sua rota é em direção ao túnel e não às paredes do detetor.

Figura 1.7: Um evento reconstruído



Eficiência e precisão têm custo: um grande volume de dados. Os experimentos com o duo LHC/ATLAS produzirão a incrível taxa de 10^8eventos por segundo. Isto se traduz num estrondoso volume de dados a serem armazenados. Isto é:

1. **Inviável,** já que não há necessidade de analisar tal volume de dados, visto que alguns destes têm alta probabilidade de não serem o alvo da pesquisa (o bóson de Higgs).

2. **Impossível,** já que não há mecanismo rápido o suficiente para tal, um evento pode produzir centenas de kilobytes de dados, que demandariam a utilização de mídia com taxas de gravação altíssimas.

Para resolver este problema, prevê-se a construção de um sistema de validação que deve operar em tempo real, separando eventos com maior probabilidade de apontarem um Higgs de eventos com menor probabilidade (a grande maioria), que devem ser descartados. Quando é dito apontar, quer se dizer indicar indiretamente a presença. Na verdade, uma vez que bósons de Higgs só existem em condições muito extremas (calor de $10^{16o}Celsius$): assim que há a desintegração particular liberando um Higgs, quase instantaneamente ocorre o decaimento deste em partículas maiores, i.e., com maior massa, mas com menor energia, formando uma cascata eletromagnética. Sistemas de validação são melhor elucidados em 1.3.

Reconstituindo os eventos...

Um dos objetivos de recolher tantos dados, filtrá-los etc é o de chegar a uma fotografia final de um evento. Esta fotografia, incluindo os dados de cada partícula envolvida no processo, representará o produto final de toda a análise.

Um evento reconstruído pelos computadores do CERN, em uma simulação, pode ser visto na figura 1.7

1.3 Sistemas de Validação

1.3.1 Porque utilizar sistemas de validação?

Como já mencionado na seção 1.2.1, as colisões próton-próton que ocorrerão no experimento envolvendo o complexo LHC/ATLAS produzirão um volume estrondoso de dados **que não podem ser gravados em sua totalidade**. Isto torna o trabalho a ser executado bem complexo, visto que a filtragem destes dados deve ser feita em tempo real. Neste experimento específico a taxa de eventos chegará à 10^8/segundo, significando que cada evento deve ser totalmente "digerido" num espaço de tempo não maior que 10 nanossegundos.

Um evento, i.e., um provável decaimento de um Higgs, pode se dar de várias maneiras diferentes e excitar diferentes regiões dos detetores, acarretando um diferente número de regiões excitadas por evento. É comum chamarmos as regiões excitadas nas paredes dos detetores de *Regiões de Interesse*. O número de Regiões de Interesse (RoI) por evento não é fixa, podendo variar de 1 a 25, embora a esperada seja de 5 RoI's por evento.

Fica evidente que uma eletrônica simples não poderá resolver o problema; algoritmos computacionais devem ser misturados a tecnologia *hard-wired* para produzir um mecanismo eficiente de separação. A validação ou *triggering* será mais eficiente, neste caso, se a dividirmos em níveis de filtragem progressiva com eficiência ascendente e velocidade descendente, ou seja, o sistema deverá ser dividido em sub-níveis em cascata com complexidade e rapidez compatíveis com os dados de entrada.

1.3.2 O sistema de *Trigger* para o experimento LHC/ATLAS

A proposta do CERN para tal sistema seria utilizar 3 níveis de processamento[5] operando concorrentemente, mas em cascata, da seguinte forma [6]:

- Nível 1 (*Level 1*)- o primeiro nível ficaria responsabilizado por receber os dados diretamente dos detetores e, baseado na "assinatura"[6] de partículas deixadas em alguns destes detetores, fazer uma identificação de eventos potencialmente interessantes. O requerimento básico deve ser velocidade, que, segundo indicações, não poderá ser atingida com a utilização de ambientes computacionais programáveis, restando somente a opção de *hard wired electronics* para esta implementação, i.e., sistemas de processamento programáveis em baixo nível.

- Nível 2 (*Level 2*)- é onde a rejeição obtida com a análise local pode ser implementada. "Local" quer dizer que somente o acesso a areas específicas dos detetores é requerida. O casamento de assinaturas de partículas usando informações de **vários** detetores e a análise topológica do evento como um todo são metas deste nível. As possibilidades de utilização de algoritmos programáveis, e de supressão de informação no processamento existem.

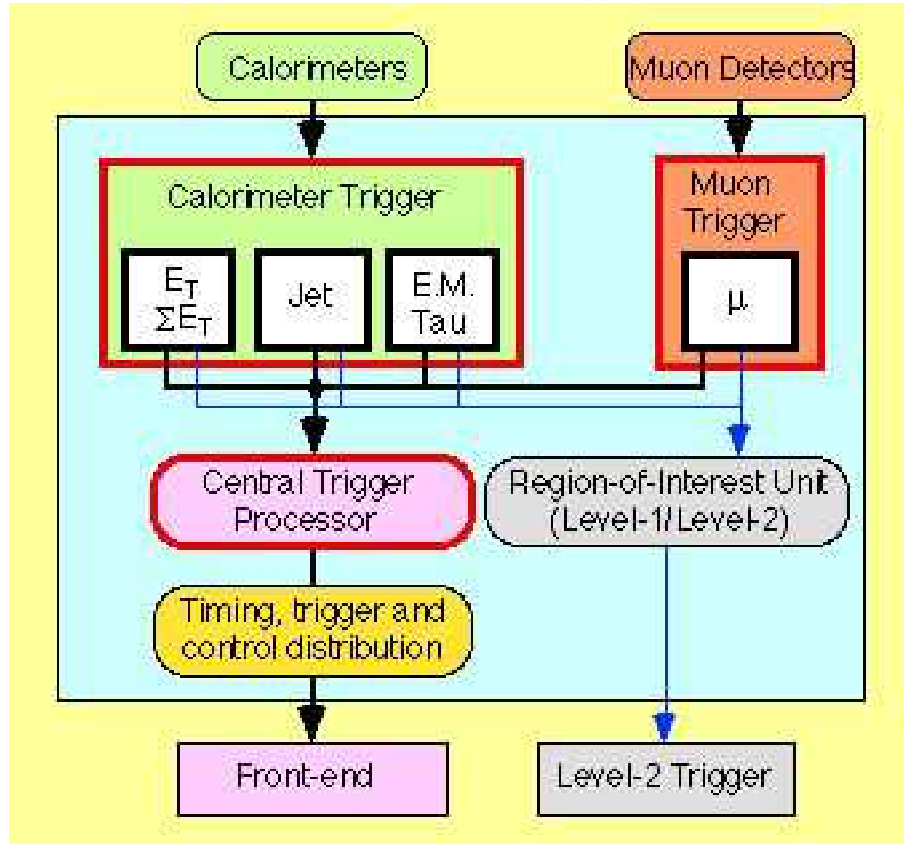
- Nível 3 (*Level 3*)- A rejeição, aqui, é feita analisando-se os todos os dados de um evento, evento a evento. Exige pesado ambiente computacional e deve ser feito com os dados pré-gravados (em sistemas comuns de armazenamento) sem que a demanda de eficiência nos algoritmos seja essencial.

O primeiro nível deve suportar uma taxa de entrada máxima de eventos de 40×10^6 por segundo(ou 40 MHz) e deve entregar ao segundo nível um máximo (em momentos de pico) de 100.000 eventos por segundo. Já o segundo nível deverá reduzir a taxa de 100 KHz máxima cedida pelo 1ºnível para não mais que 1 KHz, estima-se que o máximo de redução utilizando a análise local no primeiro nível não

[5]Observe que complexidades não pertinentes ao experimento com pacotes de prótons não serão mencionadas aqui, ainda que possam ser consultadas em [5]

[6]Diz-se da forma específica com que uma partícula qualquer excita um determinado detetor, marcando-o. Por exemplo: elétrons possuem cascata eletromagnética notavelmente mais contida que a cascata desenvolvida por hadrons, como o píon; assim, vemos que a "assinatura" de píons em calorímetros é diferente da de elétrons quando estamos medindo a energia depositada pela cascata eletromagnética formada pelo seu decaimento ao longo de uma trajetória.

Figura 1.8: Um esquema simplificado para o primeiro nível de *trigger* do experimento LHC/ATLAS, repare que o sistema não utiliza os dados de todos os detetores existentes (quatro ao total) mas somente dos calorímetros e do Sprectrômetro de Múons (Retirado de [7]).



ultrapasse o nível de redução de 1000:1. No 3ºnível uma taxa de redução próxima a 10:1 é factível e esperada. Isto dará uma redução máxima de aproximadamente 10^7:1, restando cerca de 10 a 100 eventos para serem totalmente gravados a cada segundo(nos casos de performance máxima do sistema, é claro)[6].

Segue-se uma descrição mais detalhada de cada nível e, no fim desta seção (1.3), uma visão global do sistema de validação completo retomada a partir das descrições. O segundo nível será melhor detalhado, visto ser o alvo deste projeto.

1.3.3 O Primeiro Nível de *Trigger*

O primeiro nível de *trigger* para o experimento LHC/ATLAS consiste de disparos[7] gerados por processadores que operem sobre dados produzidos pelos calorímetros (telhas e argônio líquido) e processadores operando sobre os dados de detetores rápidos de Múons. Um processamento central feito por um Processador de Disparo Central (*Central Trigger Processor - CTP*) resultará no produto final da validação a ser repassado ao segundo nível. A figura 1.8 mostra um sequema simplificado do funcionamento do primeiro nível de validação.

[7]Diz-se que um *trigger* dispara quando os processadores deste nível encontram um evento potencialmente interessante para análise posterior pelos demais níveis.

Nesta figura mostra-se a conexão entre os diversos processadores de validação, enfatizando a interdependência entre estes. Dois outros processadores são ativados por estes anteriores, o CTP e o *RoI builder*, que serão responsáveis, respectivamente, pelos mecanismos de sincronismo/nomeação dos eventos válidos e respectivas RoI-s e pela construção/alocação das RoI-s em *buffers* rápidos que serão consultados pelo seguinte nível de processamento (segundo), comandados pelo CTP. Uma RoI é sinalizada pelo primeiro nível por um grupamento de células excitadas nos calorímetros ou na câmara de Múons, segundo determinações de limitação espacial e variantes de tamanho para cada um dos detetores. Em geral é esperado que uma RoI isolada pelo primeiro nível tenha um formato pseudocônico com raio crescente do centro de colisão para a extremidade da câmara de Múons. Os *buffers* rápidos mencionados são conhecidos por *Read-Out Buffers - ROB-s*.

A taxa de rejeição prevista para este nível será de aproximadamente 1000 para 1, reduzindo a ordem de grandeza do espaço de dados de entrada de 10^8 para 10^5, i.e., a freqüência dos dados de entrada deverá baixar de cerca de 40 MHz para um valor entre 40 e 100 KHz, sendo 100 KHz o máximo previsto para a taxa de entrada do segundo nível de *trigger*.

1.3.4 O Segundo Nível de *Trigger*

O segundo nível de *trigger* deve reduzir a taxa de dados para um valor que possa ser sustentado pelo sistema de construção de eventos no 3ºnível(este valor deve estar entre 1000 e 100 KHz)[5].

O processamento para este nível consiste na execução de algoritmos cujos dados de entrada são subconjuntos de dados ainda não processados nas ROB-s, cuja procedência são RoI-s selecionadas por um disparo do 1ºnível de validação e algumas informações cedidas pelo 1ºnível. A seleção dos subconjuntos de dados a serem processados é função do 2ºnível de validação. Os algoritmos aqui utilizados produzirão variáveis que permitirão uma melhora na relação sinal/ruído, e assim melhor selecionando eventos definidos como descartáveis (*background*) e retendo a maior fração possível de canais físicos interessantes. Limitações de implementação podem levar a diferentes modelos de arquitetura para este nível, ainda que todas devam manter os conceitos de processamento baseado em RoI-s.

Discutiremos distintos passos no algorítmo que devem se manter em qualquer implementação para o segundo nível de validação e aquisição de dados do experimento:

1. Pré-processamento;
2. Coleção dos dados de uma mesma RoI nos diferentes ROB-s;
3. Extração de Características ou *Feature Extraction*;
4. Decisão Global ou *Global Decision*.

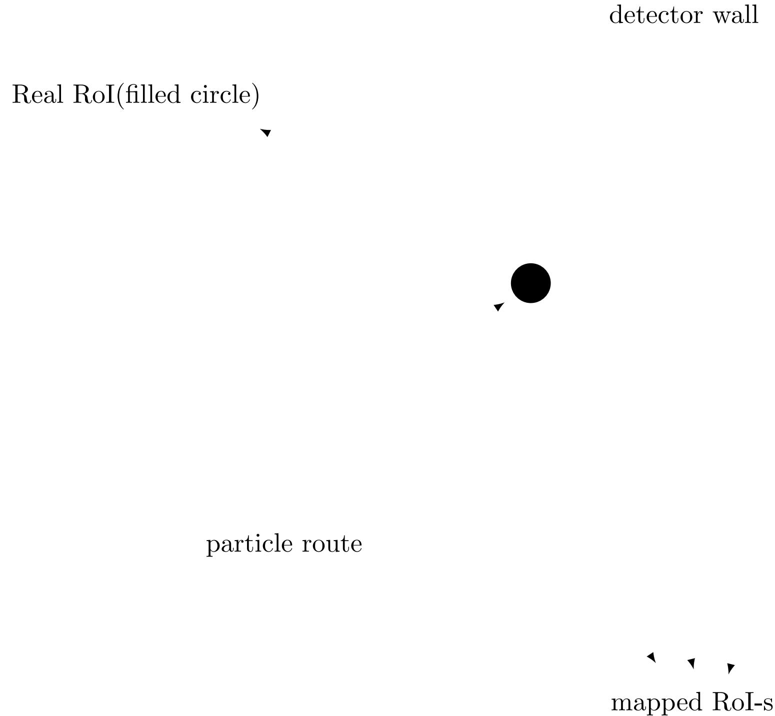
A parte do processamento relativa à decisão global ainda pode ser dividida em 2, a reunião de diferentes *features* geradas a partir dos dados de vários detetores sobre uma mesma RoI para a identificação da partícula que a gerou e a reunião dos dados de todas as RoI-s de um dado evento para a decisão final. Esta decisão é feita a partir da comparação das variáveis lidas pelo 2ºnível com múltiplas interpretações físicas possíveis (hipóteses de decaimento). Ainda, um supervisor comandará todo o processamento deste nível, podendo também ser qualificado como processo a ser implementado.

Pré-Processamento

Esta parte do processamento do 2ºnível está diretamente associada aos dados contidos nos ROB-s, gravados por um disparo do 1ºnível de processamento. Os dados gravados pelo primeiro nível nas ROB-s seguem a determinação de RoI-s encontradas nas paredes dos detetores.

Antes de qualquer colisão, regiões são delimitadas nos detetores por um processo de marcação de zonas, ou seja, é feito um mapeamento das paredes dos detetores formando pequenas regiões de interesse como mostrado na figura 1.9. É claro que as partículas provenientes de eventos não irão necessariamente ocupar somente uma das diversas possíveis regiões de um detetor, podendo ocupar

Figura 1.9: Uma ilustração sobre a idéia de mapeamento da parede de um detetor.



2, 3 ou como mostra a figura, 4 regiões distintas. É possível entender o que se passa se admitirmos a existência de uma RoI que represente um espaço físico, previamente marcada no detetor, a **RoI mapeada**, e uma **RoI real** (representada pelo círculo preenchido na figura). Partículas geralmente excitam mais de uma Roi mapeada.

O pré-processamento consiste em converter os dados retirados de mais de uma RoI mapeada em uma única RoI real, ou seja, supondo que estejamos medindo energia depositada pela partícula ao longo da trajetória como na figura, vários pontos não excitados pela partícula serão gravados nos ROB-s, pois o processo de disparo somente identificará que 4 RoI's mapeadas foram excitadas e portanto devem ser completamente salvas nos ROB-s. Isto não é uma boa coisa quanto ao ponto de vista do 2º nível de *trigger*, pois só consistirá em mais dados a serem analisados. Para reduzir a carga em cima dos algoritmos de análise deste nível, suprimem-se as partes irrelevantes dos dados de todas estas RoI-s mapeadas sobrando apenas os dados equivalentes a um disparo sobre uma RoI real. É disto que consiste o pré-processamento.

Seleção e Agrupamento de Regiões de Interesse(*RoI Collection*)

O processamento relativo a esta fase consiste em receber todos os dados de uma ou mais unidades pré-processadoras[8] e deles selecionar um subconjunto de dados relacionados a cada RoI (vista por cada sub-detetor) e rearrumá-las de forma a facilitar a execução dos algoritmos de Extração de Característica.

[8] Enfatiza-se que os dados de uma RoI real podem estar espalhados por mais de um ROB e que o pré-processamento não é responsável por juntar estes dados, somente otimizá-los como explica a seção 1.3.4.

Extração de Características (*Feature Extraction*)

Nesta fase, os processadores de *Feature Extraction* recebem os dados já organizados caracterizando as RoI vistas por cada subsistema do superdetetor. O processamento consiste em traduzir estes dados em variáveis inteligentes que podem ou não ter uma clara conotação física, mas que com certeza aumentam a relação sinal/ruído do ambiente. Estas variáveis serão utilizadas para a tomada de decisões sobre a partícula excitadora da RoI.

Empregam-se diferentes técnicas para os diferentes tipos de dados de cada subsistema de deteção. Para calorímetros, por exemplo, redes neurais têm se mostrado em vantagem contra algoritmos clássicos [8], [9]. Esta fase de extração de características é altamente paralelizável visto que dados de diferentes RoI-s ou sub-sistemas são totalmente independentes podendo ser separadamente processados sem perda de informação.

Decisão Global (*Global Decision*)

Aqui a variáveis de uma mesma RoI, já traduzidas pelo processamento de Extração, são reunidas[9] e sobre estas é feita uma decisão sobre a classe da partícula que excitou a região de interesse. Uma vez de posse deste e dos dados de todas as RoI-s que compõem um mesmo evento, esta fase do processamento será responsabilizada por decidir sobre o disparo sobre este evento. Uma vez tendo sinalizado ao supervisor que o evento deve ser retido para análise pelo nível seguinte este realizará o disparo, fazendo com que o 3º nível de validação carregue o evento. O evento será descartado caso a unidade de Decisão Global não o valide, liberando, assim, o espaço nos ROB-s.

1.3.5 O Terceiro Nível de *Trigger*

Para aqueles eventos que satisfazem ao menos uma das condições de *trigger* para o nível 2, os dados em sua totalidade devem ser transferidos dos ROB-s para a memória de algum processador do pesado ambiente computacional do terceiro nível de *trigger*. Este requerimento baseia-se na assunção de que o segundo nível proverá uma satisfatória rejeição a partir da na análise local de RoI-s de tal forma que uma maior rejeição somente poderá ser obtida com uma análise global dos eventos reconstruídos. O terceiro nível de *trigger* é composto de 5 elementos: O construtor de eventos (*Event Builder*), uma interface chaveável totalmente conectada às ROB-s e aos processadores do terceiro nível, processadores dedicados à análise de eventos reconstruídos, um sistema de armazemento e um supervisor, como descritos abaixo:

- **Event Builder (EB)-** sua responsabilidade é de identificar todos os dados relativos a um evento validado pelo segundo nível e repassá-los à interface de chaveamento, devidamente identificados.
- **Interface de chaveamento-** é responsável por repassar os dados identificados como comuns a um mesmo evento a um processador de terceiro nível livre.
- **Processadores Dedicados de terceiro nível-** têm a função de minimizar a quantidade de dados que serão permanentemente gravados em 2 formas:
  - rejeitando eventos não interessantes com base na análise global do evento; e
  - formatando e reduzindo o volume total de dados por evento.

  É viável a utilização de sistemas de processamento mais complexos ao invés de baseados em apenas um nó para os processadores deste nível.

  Os processadores devem possuir capacidade suficiente para rejeitar **quaisquer** eventos que estejam fora das características requeridas pela análise estatística dos canais físicos a serem observados. Ainda, o processamento deve ser feito distribuído com relação ao nível de eventos, ou

[9] Na realidade um processamento local é responsável pela reunião das diversas características de uma mesma RoI.

seja, o número de eventos a serem analisados por base de tempo demandará o número de nós de processamento necessário que operarão concorrentemente na análise dos eventos.

- **Sistema de Armazenamento-** constituirá da tecnologia que melhor se adaptar ao modelo de dados para o ATLAS para gravação e análise.
- **Supervisor-** Controla todas as operações descritas anteriormente.

1.3.6 Compondo o sistema de validação para o experimento LHC/ATLAS

Podemos resumir o sistema de validação para cada subnível em:

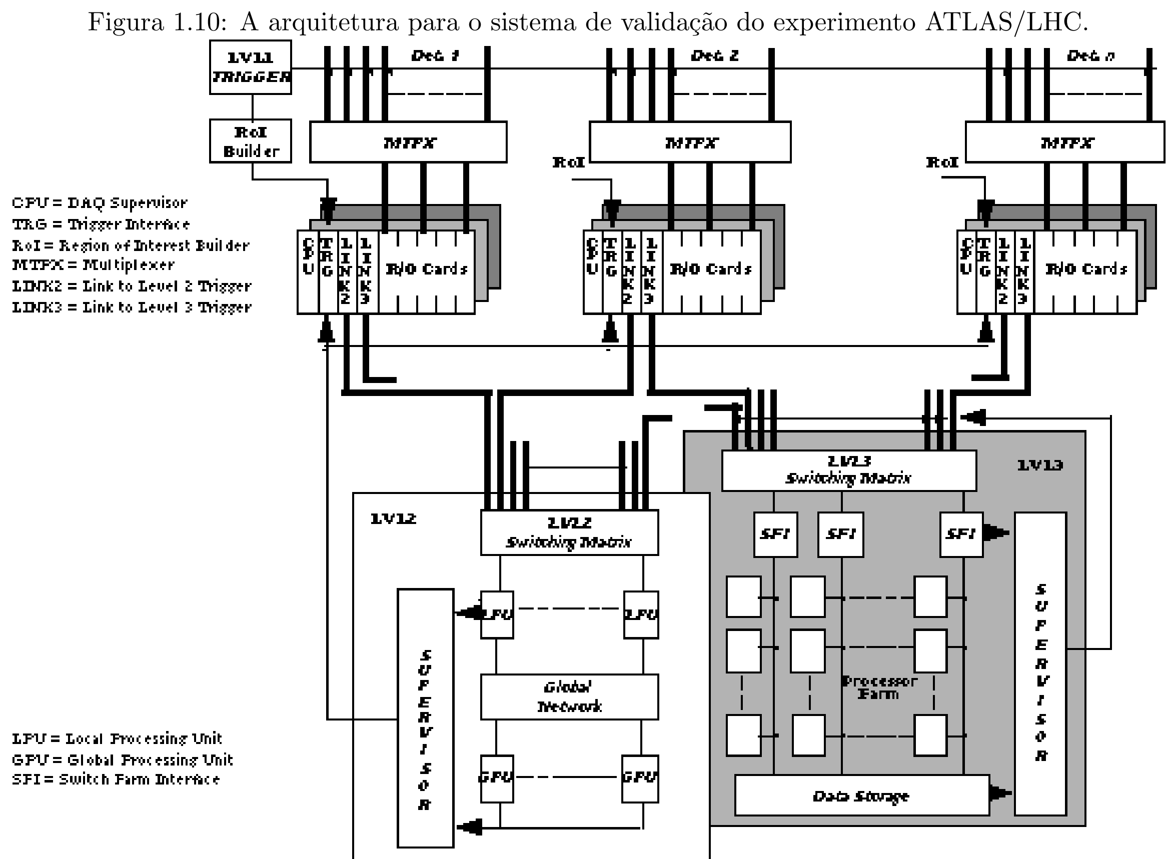
- **Nível 1**
  1. Taxa de redução esperada na ordem de 10^3, para uma taxa de entrada de 40 MHz;
  2. Utiliza ambientes não-programáveis (*hardwired resources*);
  3. É responsável por ativar o CTP e o RoI Builder, ativando assim o 2ºnível.
- **Nível 2**
  1. Taxa de redução na ordem de 10^3;
  2. Utiliza lógica programável e altamente paralelizável para vários níveis, inclusive os de Extração de Características e Decisão Global;
  3. É composto de níveis de processamento local (Pré-processamento,*RoI Collection* e Extração de Características) e níveis de processamento global, como o de Decisão Global;
  4. Todo processamento é controlado por um processo supervisor.
- **Nível 3**
  1. Taxa de redução de 10 a 100 para 1;
  2. Utiliza pesado ambiente computacional, altamente programável;
  3. A análise é global, simultânea, de todos os dados do evento, uma vez que a separaçao utilizando análise local foi superutilizada no segundo nível;
  4. Todo o processamento é controlado por um processo supervisor.

A figura 1.10 mostra um esquema simplificado de arquitetura para o sistema de validação do experimento.

1.4 Próximos capítulos

Nos próximos capítulos deste documento estaremos rumando a uma solução de simulação do 2ºnível de validação do experimento LHC/ATLAS e conclusões deste trabalho. No capítulo 2 revisaremos alguns documentos publicados sobre o assunto e implementações correlacionadas as que utilizaremos neste projeto. No capítulo 3, uma revisão de alguns dos fundamentos teóricos empregados, como Redes Neurais Artificiais e Paralelismo em aplicações gerais, assim como uma descrição da máquina que disporemos para implementação serão feitos. No capítulo 4, uma descrição de fusão das técnicas mencionadas versus sua implementação real será discutida. O capítulo 5 estará reservado para os resultados e para as conclusões sobre o trabalho. O leitor ainda achará no capítulo 6 uma discussão sobre possíveis continuações deste projeto.

Figura 1.10: A arquitetura para o sistema de validação do experimento ATLAS/LHC.



Capítulo 2

Revisão da Literatura

Neste capítulo faremos a revisão de algumas implementações e conceitos bem-sucedidos na descrição e operação do 2ºnível de validação para o experimento ATLAS/LHC; destacam-se a utilização de Redes Neurais como algorítmo para alguns Extratores de Característica e unidades de Decisão Global e, ainda, a utilização de processamento paralelo em algumas implementações.

O motivo de tal direcionamento deste estudo deve-se à disponibilidade de equipamento com processamento distribuído e bem-sucedidas implementações através de nosso grupo (Colaboração Internacional CERN/COPPE/UFRJ) utilizando redes neurais artificiais na execução de extratores de características e unidades de decisão global para a identificação de partículas.

2.1 Redes Neurais Artificiais e o processamento global para o segundo nível de *trigger*

Como elucidado na seção 1.3.4, o segundo nível de *trigger* deve combinar as características vindas de diferentes subdetetores, de forma a aumentar a qualidade de classificação das partículas. Neste estágio é melhor não realizar uma identificação baseada em lógica exclusiva, pois poderemos diminuir a qualidade de decisão, mas gerar um outro conjunto de variáveis, com probabilidades sobre a natureza da partícula [10].

Considere o exemplo simples onde dois subdetetores não são correlacionados, ou seja, suas características extraídas são linearmente independentes (LI-s). A forma mais ineficiente de fazer uma decisão é a de classificar a partícula segundo cada subdetetor e depois perfazer um "E" lógico (AND) entre as classificações. Considere a ilustração da figura 2.1 com as diferentes características extraídas da cada subdetetor (genérico), de tal forma que o nível de identificação para 2 tipos de excitadores de RoI, elétrons e jatos, se dá com uma sobreposição de 10%, o quer dizer que se traçarmos um corte para identificação em cada distribuição teremos um erro mínimo de 10%. Neste exemplo a classificação exclusiva indicará somente 81% de correção na identificação das partículas (o produto das probabilidades).

A decisão poderá ser melhor tomada se as duas características forem analisadas juntas, em um espaço bidimensional (figura 2.2). Cortando com um plano como é mostrado, ambos os tipos de partículas são 96% corretamente classificadas. Estes tipos de cortes são realizados também por estruturas de Redes Neurais Artificiais utilizando a função identidade como a função de transferência para os neurônios. Se o número de subdetetores crescer, a diferença entre estas duas maneiras de classificação ficará mais explícita, como é possível observar na tabela 2.1. Neste teste foi considerado que todas as características foram extraídas de subsistemas LI-s, assim sendo, não houve necessidade da utilização de camadas escondidas.

Embora pareça simples, a tarefa para a identificação de partículas é um processo que envolve carac-

Figura 2.1: A distribuição de características para cada subdetetor genérico. Os subsistemas são LI-s.

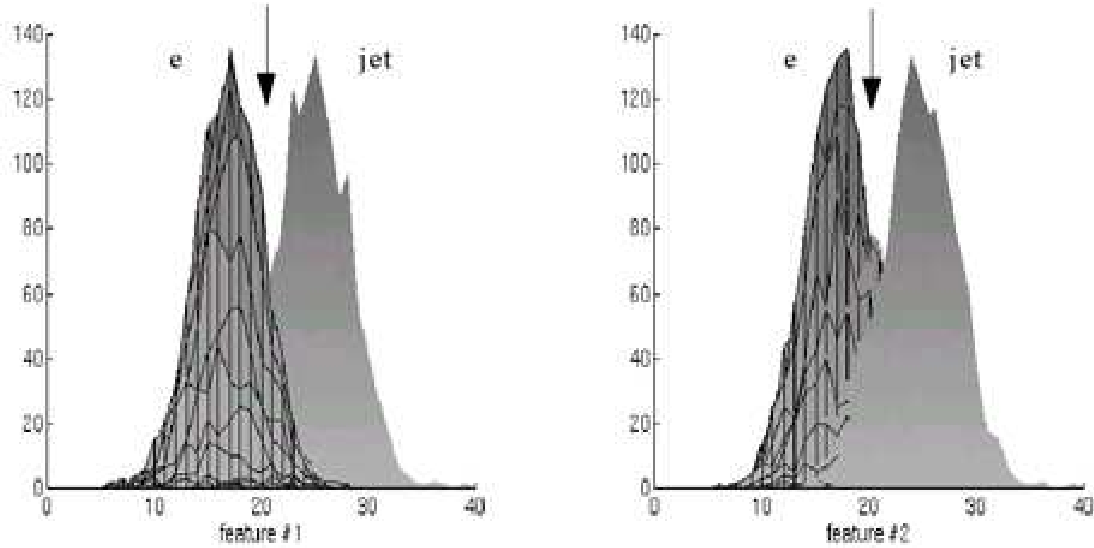


Figura 2.2: Exemplo de classificação num espaço bidimensional

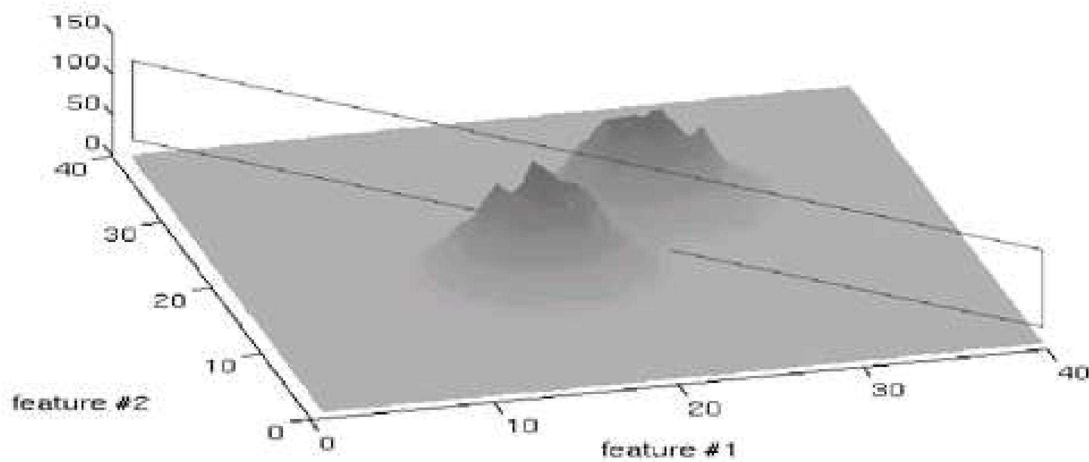


Tabela 2.1: Tabela de eficiências para uma classificação utilizando um "E" lógico e Redes Neurais Artificiais simples, sem camadas escondidas.

Número de Detetores	Classificação com "E" lógico	Classificação com ANN
1	90%	90%
2	81%	96.5%
3	73%	98.8%
4	65%	99.4%
5	59%	99.8%

terísticas extraídas dos dados de um número de subdetetores menor que o número de características[1], portanto, algumas destas são linearmente dependentes (LD-s). Para máxima eficiência na separação com variáveis LD-s a utilização de camadas escondidas é necessária. A figura 2.3 mostra o resultado de separação obtida sobre dados LD-s utilizando 3, 2 ou nenhum neurônio na camada escondida.

Conclusão: O processamento de Decisão Global terá ótima eficiência se utilizarmos Redes Neurais Artificiais (ANN-s) ao invés de algoritmos basedos em classificação exclusiva, devido ao número de subsistemas e variáveis envolvidas. Em razão da dependência entre as variáveis dos diversos subsistemas parece inevitável a utilização de ao menos uma camada escondida em uma rede neural simples, diretamente conectada (*feed forward*).

Em particular, resultados com redes na configuração 12-6-4, isto é, 12 entradas, 6 neurônios na camada escondida e 4 na de saída têm se mostrado satisfatórios onde tempo de processamento e eficiência são fatores dominantes como é possível ver em [8]. O número de saídas é fortemente dependente da quantidade de partículas diferentes que se deseja identificar no processamento da RoI. No caso específico supracitado a identificação se fez por uma separação entre 4 diferentes tipos de partículas (elétrons, jatos, píons e múons).

Devemos destacar ainda que redes neurais são capazes de encontrar correlações em espaços multidimensionais ainda que na presença de grande quantidade de ruído. Redes Neurais artificiais também podem ser muito competitivas onde preço e robustez são fatores delimitantes.

2.2 A utilização de ambientes de processamento distribuído para o segundo nível de *trigger*

Devido às altas taxas de eventos a que será submetido o segundo nível de *trigger* (muitos Gigabytes por segundo e uma latência máxima presumida de 1ms), a decomposição do problema através de paralelização das atividades parece apontar um caminho para a solução. Esta paralelização pode ser implementada em 2 níveis:

1. Na paralelização de uma atividade simples como a extração de uma característica ou a identificação de uma partícula, de modo a reduzir a latência do processamento localizado; e

2. Na paralelização de uma classe de atividades ou de todo o processamento para este nível, por exemplo, a extração de características utilizando uma *farm* de processadores operando em paralelo sobre dados de diferentes RoI-s, reduzindo a latência do processamento como um todo e não localmente como no item anterior.

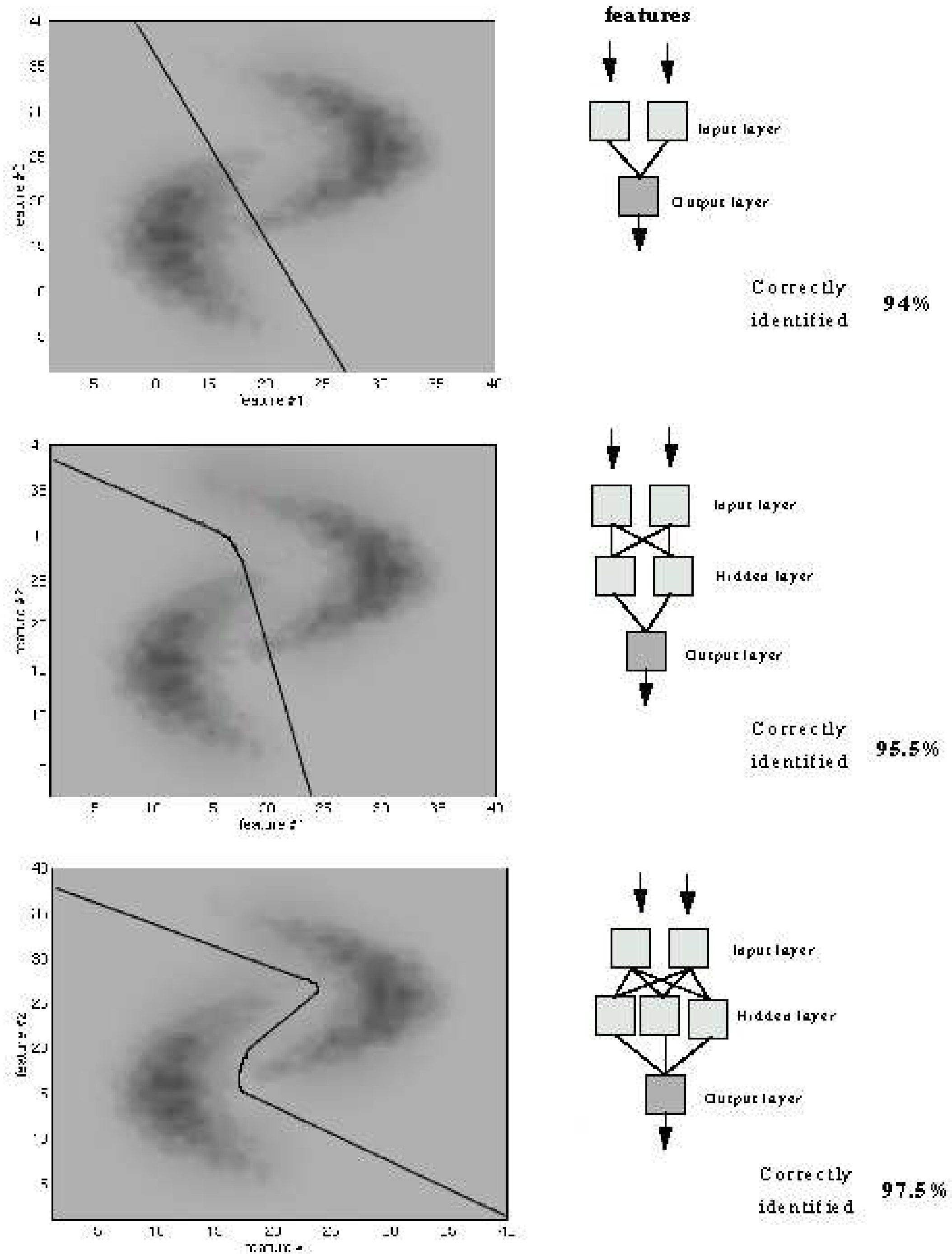
Os dois tipos de paralelização levam a diferentes enfoques de implementação: no primeiro teremos um algoritmo baseado na taxa de dados (*Data Driven*) de entradab visto que a paralelização é feita de modo a reduzir as latências individuais, reduzindo a latência global do processo para cada evento. Nesta implementação deve-se utilizar uma arquitetura capaz de sustentar a taxa de dados, como em [11].

No segundo enfoque, a latência dos processos individuais não é demasiado importante, visto que poderemos compensar uma maior latência com a utilização de mais processadores naquele nível de processamento. Este tipo de implementação é conhecido como *Asynchronous Processor Farms Implementation* ou Implementação em *farms* de processadores assíncronos. O assincronismo vem do fato das atividades paralelas serem independentes quanto ao tempo de execução entre si, podendo ser executadas totalmente livres do final (ou começo) de processamento de qualquer evento ou dado em qualquer outra unidade de processamento.

No caso do segundo nível em específico, podemos enxergar paralelismo em algumas das atividades realizadas e no processamento como um todo também. Para algoritmos de extração de características

[1] O número de detetores está entre 4 e 6 e o de características, entre 12 e 15.

Figura 2.3: Classificação em um espaço de variáveis linearmente dependente.



em calorímetros, técnicas de processamento paralelo assíncrono têm se mostrado eficientes no atendimento das taxas de entrada do experimento, como é possível ler em [9].

Para as unidades de Decisão Global, a utilização de redes neurais artificiais têm mostrado vantagem, como explicita a seção 2.1. Redes Neurais são algoritmos altamente paralelizáveis por serem baseados em multiplicações e somas vetoriais totalmente independentes ao nível de cada camada[2]. Ainda, otimizações podem ser realizadas a nível assíncrono utilizando-se mais de uma rede para o processamento global.

Por fim, as atividades do segundo nível como um todo são paralelizáveis visto a independência entre diferentes eventos. Estes podem ser processados independentemente (e, desta forma, assíncronamente) uns dos outros, sem perda de informações, de forma que a latência para cada evento possa ser aumentada proporcionalmente do valor alvo de 1ms para outro valor maior, dependendo de quantos forem os nós de processamento adicionados. Este enfoque, é claro, reduz a necessidade de otimização dos subníveis de processamento internos ao segundo nível, como é o caso do enfoque baseado na taxa de dados (*Data Driven Approach*).

2.2.1 DSP-s

Digital Signal Processors (processadores digitais de sinais) não são mais do que rápidos processadores matemáticos que executam funções aritméticas básicas e outras funções complexas com poucos pulsos de *clock*. Alguns DSP-s como o ADSP-21020 da Analog Devices podem processar em apenas 1 ciclo de *clock* uma multiplicação entre números reais(ou pontos flutuantes).

Vantagens na utilização de DSP-s são explícitas quando o processamento depende do cálculo de muitas variáveis. No caso do processamento para o segundo nível, existem muitos subníveis em que é possível atingir uma redução da latência de processamento utilizando-se DSP-s como processadores centrais do subnível [11]. Este é o caso da maioria dos extratores de característica e das unidades de decisão global, se desenhados como Redes Neurais Artificiais.

2.3 Arquiteturas para o segundo nível de *trigger*

Limitações de tamanho e custo podem nos levar a 3 possíveis arquiteturas mutuamente exclusivas para o segundo nível de processamento[12]:

- **Modelo A:** Utiliza uma partição local/global do segundo nível de processamento, com processamento até a extração de características (inclusive) sendo realizado com arquitetura baseda no enfoque *data-driven*, enquanto que o processamento global utiliza uma *farm* de processadores utilizando o enfoque de processamento assíncrono;

- **Modelo B:** Utiliza uma partição local/global de todo o processamento do segundo nível, fazendo uso de várias *farms* feitas de processadores idênticos ou similares com extração de características paralela para cada subdetetor e para cada RoI;

- **Modelo C:** Uma única *farm* de processadores realizando, cada um, o processamento relativo a um evento inteiro.

Estas arquiteturas seguem alguns critérios de funcionamento tais como não possuir latência de operação[3] superior a 2 ms (modelo A) e 10 ms (modelos B e C); caso esta ultrapasse, o supervisor deve se responsabilizar por abortar a operação de validação para este evento. Estes critérios podem ser melhor estudados em [12] e [5].

[2]É claro que os resultados da camada de ordem n dependem dos resultados da camada de ordem $n-1$ e, sendo assim, não podemos tornar isto concorrente, ou seja, o processamento ainda deve ser feito camada a camada.

[3]Tempo de demora entre a decisão do 1º nível e a decisão do segundo.

2.3.1 Modelo A

Neste modelo A, a informação de cada RoI é comunicada pelo *RoI Builder* às ROB-s (isto ainda é no 1ºnível). Os dados são então "empurrados" através do pré-processamento e coleção de RoI-s para a extração de características. Nesta parte de operação as unidades de processamento (acopladas, formando um único subsistema) deverão suportar a mais alta taxa de eventos possível. As operações são totalmente independentes para os dados de diferentes subdetetores. Para cada RoI, um vetor de dados é preparado contendo as características extraídas e um cabeçalho de identificação. Depois da extração de características, estes vetores são encaminhados a uma chave que é responsável por distribuí-los pelos processadores globais. A figura 2.4 pode ser elucidativa quanto ao mencionado nesta seção.

2.3.2 Modelo B

Neste modelo B, a informação de cada RoI é passada ao supervisor que controla **todo** o fluxo de dados. Os dados de cada RoI são passados dos ROB-s através do pré-processamento e coleção de RoI-s a uma chave que se encarrega de passar os dados relativos a cada RoI para extratores de características. Nesta fase, os subsistemas relativos a cada subdetetor são totalmente independentes, podendo ter seus próprios supervisores locais. As características extraídas são, então, passadas através de outra chave (que reúne as características extraídas de cada subdetetor sobre uma mesma RoI), que repassa estes dados às unidades de decisão global. A figura 2.5 mostra um diagrama ilustrativo.

2.3.3 Modelo C

Neste modelo C, a informação é passada a um supervisor central que dedica um dos processadores de uma *farm* homogênea a todo o processamento do evento. Este processador controla todo o fluxo de dados da RoI pelos ROB-s e pelas unidades de pré-processamento e coleção de RoI-s. Os dados, depois de coletados, seguem a uma chave que os repassa ao devido processador de evento, para que este realize a extração de características e a decisão global. A figura 2.6 mostra um diagrama ilustrativo.

2.4 Concluindo alguns pontos...

É possível utilizar ANN-s para desenhar alguns subsistemas do segundo nível de validação, em especial Unidades de Decisão Global e Extratores de Características para Calorímetros. Estas redes especializadas no reconhecimento de padrões podem ter sua latência reduzida se utilizarmos computação distribuída uma vez que seu algoritmo é altamente paralelizável. Ainda, se utilizarmos DSP-s como processadores centrais de alguns destes subsistemas poderemos nos beneficiar de sua alta performance computacional em algoritmos envolvendo repetidos cálculos.

A paralelização da aplicação (ainda que complexa) poderá acontecer em 2 instâncias: na primeira, local, há uma paralelização buscando a otimização de um algoritmo utilizado em um subsistema, de forma a atender às taxas de entrada e latência máximas deste subsistema; chamamos este enfoque de *data-driven* uma vez que os dados fluirão por estes subsistemas sem restrições ou controle, já que estes suportarão as taxas de entrada máximas. Na segunda instância, global, acontece uma paralelização de subsistemas idênticos (exemplo: Extratores de Característica) de forma que a latência máxima para cada evento seja aumentada proporcionalmente ao número de processadores anexados. Em particular a paralelização em primeira instância não será aqui utilizada visto que equipamento específico para cada subsistema paralelizável é requerido. Utilizaremos, no entanto, o conceito de paralelização global, ou em segunda instância como veremos mais a frente.

Existem vários modelos para a implementação do segundo nível, cada um com suas vantagens e desvantagens, embora de uma forma geral bem plausíveis. Para fins e objetivos deste projeto, no entanto, nos concentraremos no modelo B. O modelo A requer utilização de processamento diversificado para a camada de pré-processamento, coleção de RoI-s e Extração de Características, o que não será

Figura 2.4: Um diagrama esquemático para o modelo arquitetural A do segundo nível de validação.

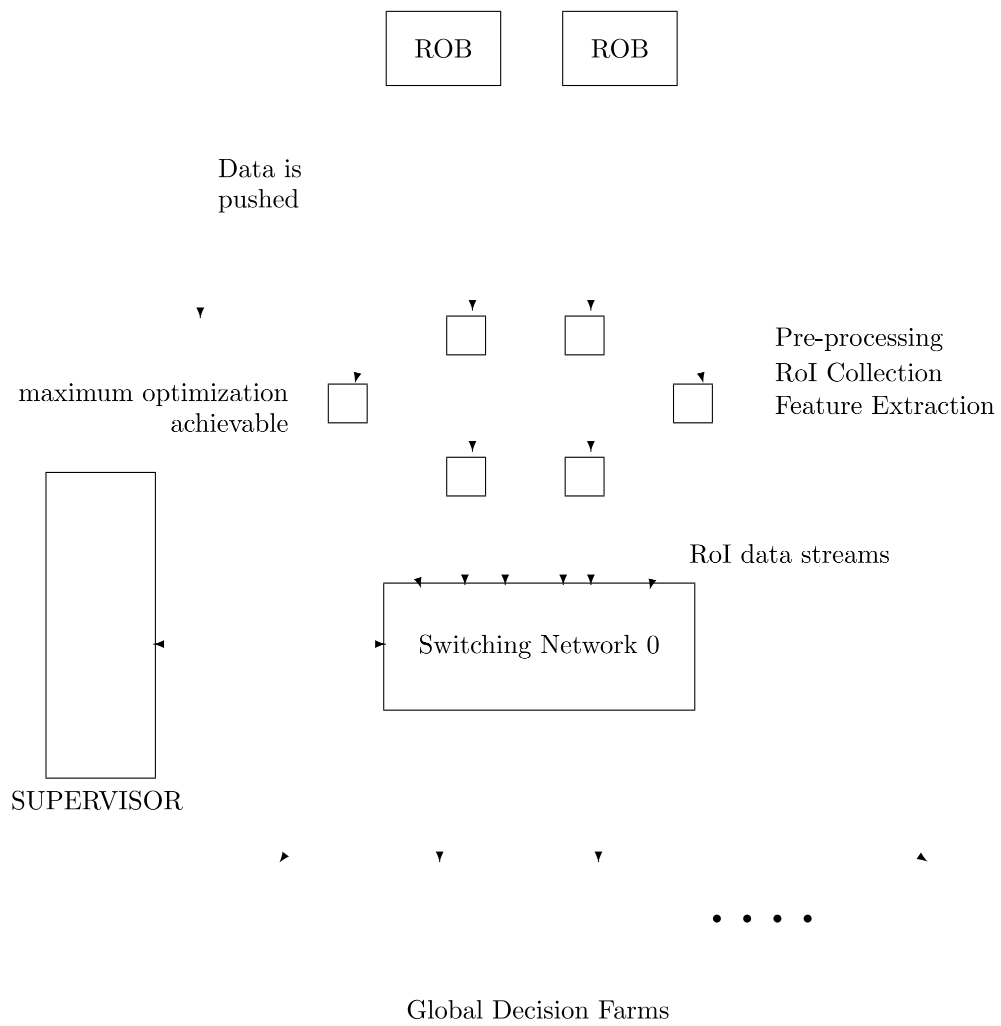


Figura 2.5: Um diagrama esquemático para o modelo arquitetural B do segundo nível de validação.

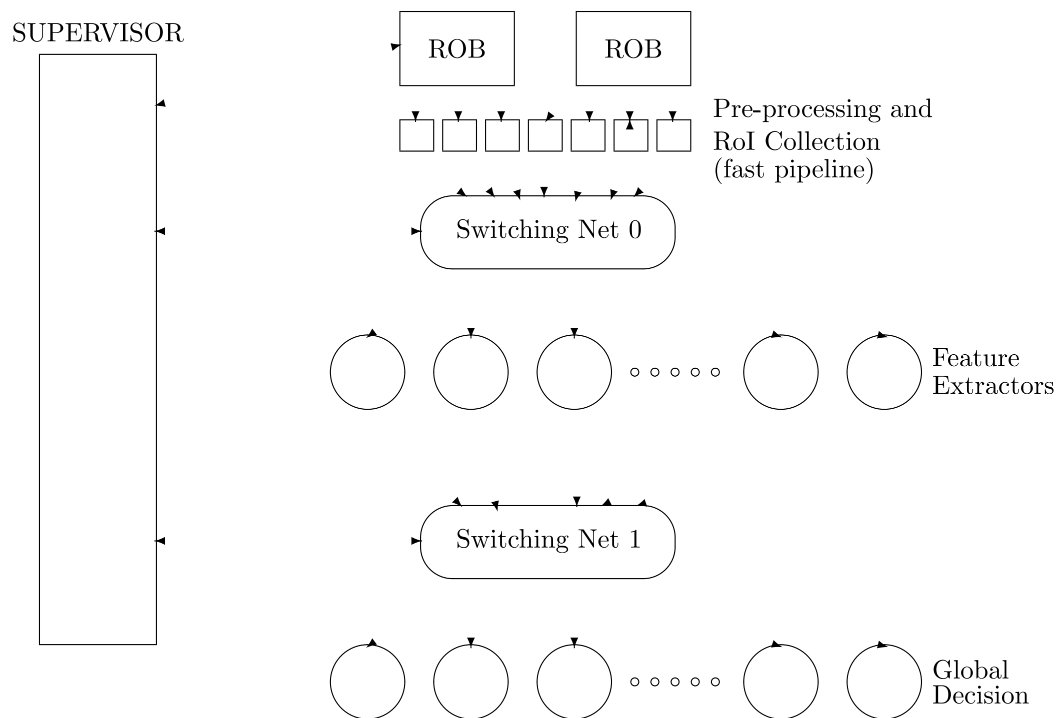
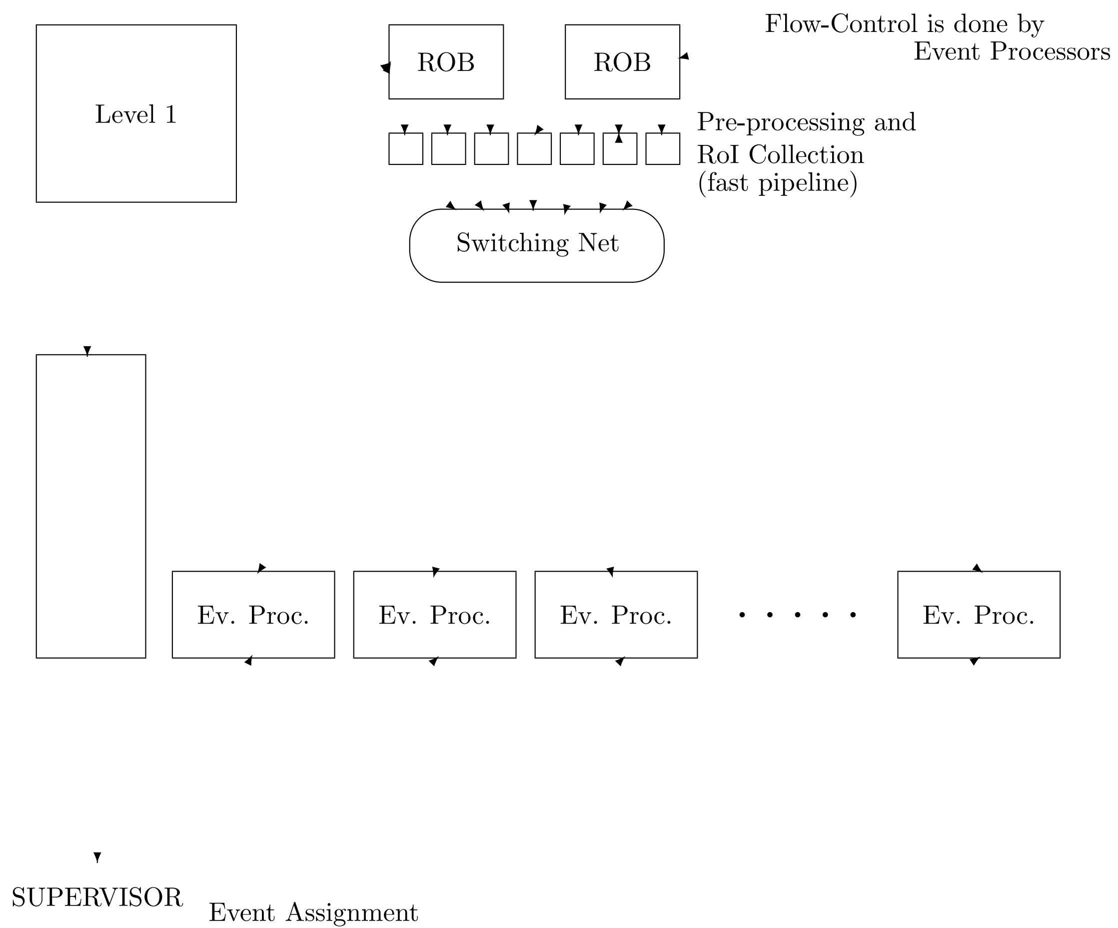


Figura 2.6: Um diagrama esquemático para o modelo arquitetural C do segundo nível de validação.



possível, pois somente dispomos de 16 processadores em nossa máquina (vê-la-emos em detalhes mais adiante) que terão seus potenciais subaproveitados se os utilizarmos em uma paralelização local ou em primeira instância.

O modelo C exige um chaveamento a partir dos ROB-s utilizando uma chave muito rápida (tecnologia ATM), que também não se encontra em nosso poder. A arquitetura B, no entanto, nos parece mais adptável ao equipamento disponível (TELMAT TN310) e que queremos avaliar; sendo assim, tentaremos abordar o problema e construir o simulador baseado neste desenho. Lembramos que queremos atingir 2 metas aqui: a primeira é construir um sistema de validação, a segunda, testar a tecnologia contida em uma TN310 de forma a avaliar globalmente arquitetura e equipamento para a realização do segundo nível de validação para o experimento ATLAS/LHC.

No que se segue exporemos em detalhes alguns dos conceitos, materiais e métodos utilizados neste projeto, isto é, Redes Neurais Artificiais, a máquina mencionada - Telmat TN310, os dados utilizados e alguns conceitos de paralelismo.

Capítulo 3

Fundamentos Teóricos, Métodos e Materiais

Neste capítulo abordaremos alguns dos conceitos, materiais e métodos a serem empregados no projeto. Isto inclui Redes Neurais Artificiais, algoritmos inspirados em neurônios animais que exibem altíssima eficiência no reconhecimento de padrões em ambientes ruidosos, a Telmat TN310, uma máquina com processamento distribuído contendo 16 nós de processamento onde implementaremos o projeto, Técnicas de paralelismo, métodos empregados para a paralelização de atividades seqüenciais e os dados utilizados para a simulação, de onde foram extraídos e sua validade.

3.1 Redes Neurais Artificiais

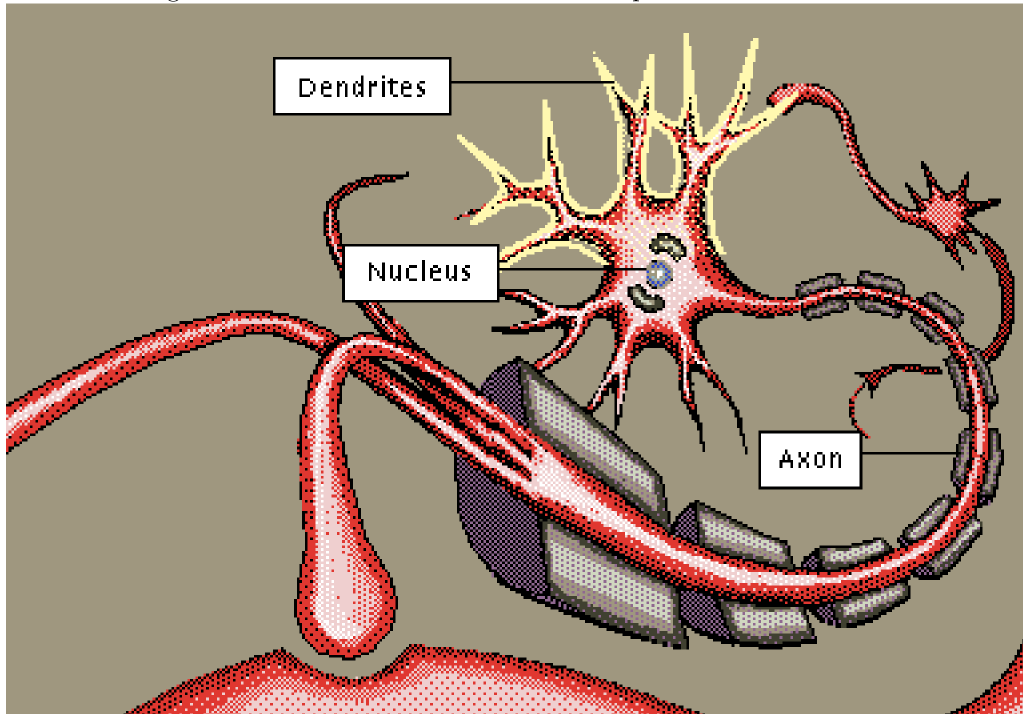
Redes Neurais Artificiais são biologicamente inspiradas[13], isto é, são compostas de elementos que funcionam de uma maneira análoga às funções mais elementares de um neurônio biológico. Estes neurônios artificiais são agrupados de modo a formar uma rede complexa (ou não) que pode ou não se parecer com estruturas cerebrais ou nervosas de um ser vivo. Estas redes são as que dão o nome a este algoritmo.

O surgimento de tais neurônios artificiais se deve a estudos na área de neurobiologia e neuroanatomia. Estudiosos do campo conseguiram aos poucos desvendar características do funcionamento de neurônios naturais, como a forma com que se comunicam e algumas configurações de operação. Estes estudos levaram ao desenvolvimento de modelos matemáticos capazes de provar alguns dos fatos desvendados neste complexo ambiente que possui centenas de bilhões de células interconectadas a outras centenas de bilhões, formando uma complexa rede de processamento.

Os modelos de neurônios artificiais são na verdade muito simples, constando somente de multiplicações e somatórios aplicados a uma função de ativação. Em outros termos, estes modelos não são mais do que extensões da lógica computacional hoje existente. A construção de redes contendo diversos neurônios interconectados de diversas formas diferentes levou os cientistas a comprovarem, em modelos artificiais, várias das capacidades de uma rede neuronal real, como o reconhecimento de padrões baseado em aprendizagem (ou seja, estas redes artificiais são capazes de aprender a reconhecer um padrão), a abstração a partir de um experiência anterior (por exemplo, reconhecer que uma cadeira é uma cadeira, mesmo sem nunca tê-la visto antes) e por fim a capacidade de generalização ou entender um padrão por trás de um ambiente muito ruidoso.

Com períodos de maior ou menor fervor, o estudo de redes neuronais, técnicas de aprendizado para os neurônios e aplicações para estas estruturas começaram a surgir e despontar em várias partes do mundo, indicando um novo conceito em programação no reconhecimento de padrões. Nas próximas subseções veremos alguns dos conceitos aqui introduzidos, como o modelo neuronal e técnicas de aprendizado.

Figura 3.1: O desenho mostra as diversas partes de um neurônio



3.1.1 O Neurônio Artificial

Para que cheguemos a um modelo artificial, partiremos do entendimento de um neurônio real de forma controlada, isto é, analisaremos alguns dos pontos interessantes nestes dispositivos que nos levem a um modelo satisfatório para o neurônio artificial. Pesquisadores nesta área estão sempre pensando sobre a organização do cérebro quando consideram configurações de redes e algoritmos. Neste ponto a correspondência entre mundo real e modelagem matemática deve acabar; a complexidade cerebral é tão grande que poderemos nos atrapalhar ao invés de rumarmos a uma modelagem cabível.

As figuras 3.1 e 3.2 mostram um desenho de partes de um neurônio e do mecanismo de trasmissão de sinais através de neurônios usando uma sinapse.

O funcionamento de tal estrutura é, segundo nosso modelo orientado, simples: o impulso elétrico é transmitido por intermédio de um líquido sináptico das sinapses do neurônio transmissor ao dendrito do neurônio receptor, que recebe, concomitantemente, de acordo com o tipo de operação, sinais de outros neurônios, com menor ou maior intensidade. Estes sinais são somados, ponderadamente, de acordo com o local e intensidade de recebimento, e um novo sinal é gerado, sendo, então, repassado ao próximo neurônio, que repetirá o processo até que este se conclua de alguma forma.

O equivalente artificial deve aproveitar algumas características deste modelo e outras, não. Para esta implementação prática modelou-se o neurônio artificial como um processo matemático contendo diversas entradas que serão ponderadamente somadas e fomarão uma única saída. Os elementos de entrada e saída estarão no conjunto dos números reais, ainda que isto não seja uma regra. A figura 3.3 pode ser elucidativa quanto a este modelo.

Observamos que a célula possui vários dendritos por onde entra a informação a ser tratada; estes dendritos possuem um peso atribuído que é, então, multiplicado a cada entrada. As entradas ponderadas são somadas e aí tem-se a saída, usalmente chamada de saída da rede ou NET.

A saída da rede ou NET ainda é processada, em muitos casos, por uma função de ativação F de

Figura 3.2: O desenho mostra a transmissão de sinais entre neurônios por intermédio de sinapses.

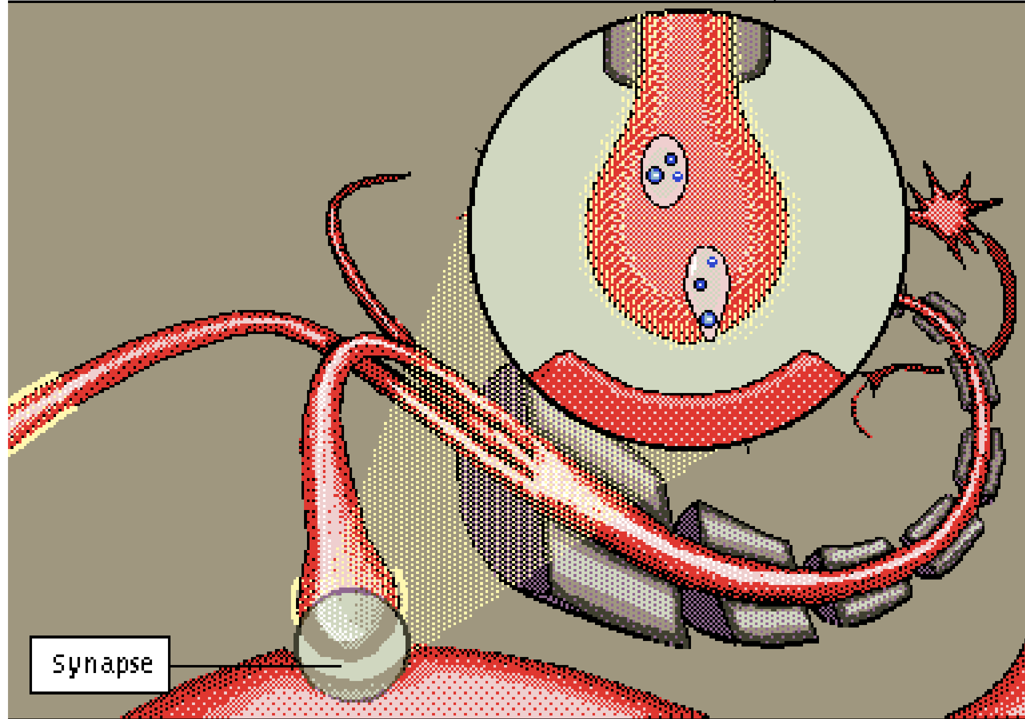


Figura 3.3: Um neurônio artificial.

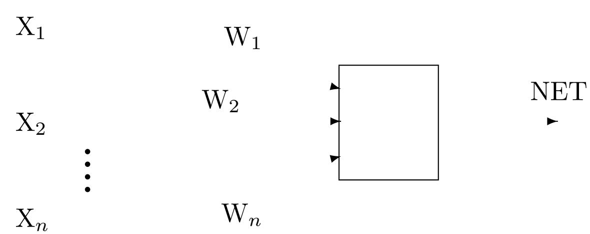
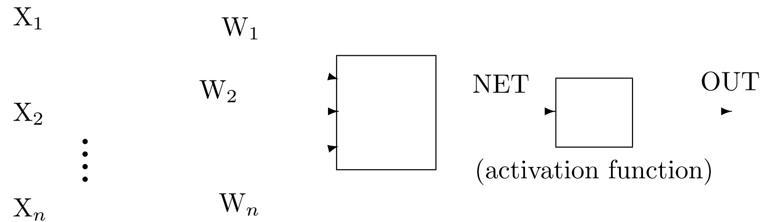


Figura 3.4: Um neurônio artificial com função de ativação.



forma a produzir a saída do neurônio, OUT. Esta função pode ser uma simples função linear ou uma função não-linear que se aproxime mais às características não-lineares de transferência de um conjunto de neurônios reais. Existem fundamentos que comprovam que a utilização de funções não-lineares como 3.1 ou 3.2 nos levam, além da aproximação ao modelo real, à solução do dilema da saturação por ruído [13]:"Como poderá a mesma rede de neurônios lidar com sinais grandes e pequenos?". Isto nos leva à questão de que redes adaptadas a um sinal de entrada tipicamente baixo saturarão se a entrada for alta, e redes adaptadas a um sinal alto não terão sensibilidade o suficiente para distinguir sinais de baixo nível.

$$OUT = \frac{1}{1 + e^{-NET}} \tag{3.1}$$

$$OUT = \tanh(NET) \tag{3.2}$$

Funções não-lineares como 3.1 e 3.2 e funções de ativação em geral podem ser pensadas como definindo um ganho não-linear para o neurônio artificial. Este ganho é calculado achando-se a taxa de mundaça de OUT para uma pequena mudança em NET. Então, o ganho é a inclinação da curva num ponto específico de excitação. Ele varia de um valor baixo para NET-s mais elevados para valores mais altos com NET-s próximos à origem. Isto resolve o dilema de Grossberg! A figura 3.4 mostra um neurônio com função de ativação. Considerá-la-emos como a célula básica daqui por diante.

3.1.2 Formando Redes Neurais

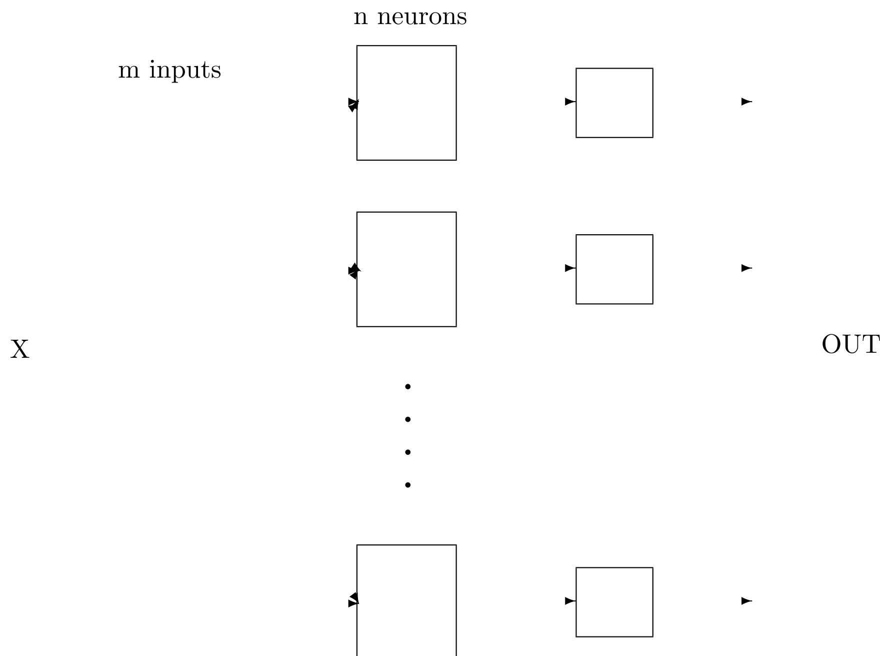
Embora um neurônio sozinho seja capaz de executar algumas funções de reconhecimento de padrões, o verdadeiro potencial desta técnica encontra-se em montar redes com neurônios artificiais formando camadas de neurônios. Configurações mais simples possuem apenas uma camada de neurônios. Aquelas mais complexas podem conter inúmeras camadas e, ainda, neurônios de camadas de maior nível com saída conectada a entradas de neurônios em camadas de menor nível, ainda que este estudo não seja revisado aqui. Neste documento nos limitaremos à revisão de redes na configuração *Fully Connected/-Feed Forward*, que nomeia redes cujos neurônios têm suas saídas totalmente conectadas aos dendritos da camada seguinte, e não há retorno de informação de camadas superiores.

Redes com uma camada

A mais simples rede neuronal é formada por um grupo de neurônios arrumados numa única camada, como mostra a figura 3.5. O conjunto de entradas X tem cada um dos seus elementos conectados a cada neurônio artificial (NA) através de um peso (vetor W). Cada neurônio somente computa a soma ponderada das entradas da rede e a passa pela função de ativação, obtendo a saída final. Alguns preferem se referenciar a neurônios nesta configuração como percéptrons.

A prova do teorema de aprendizado de percéptrons foi dada em 1962 e demonstrou que um percéptron pode aprender qualquer coisa que pode representar. Ou seja, este pode aprender a funcionar como qualquer função que possa representar, dando correta saída para entradas conhecidas, desde que, claro, seja treinado para isto.

Figura 3.5: Uma rede neural com uma única camada.



Devido à limitações no número de funções que percéptrons podem representar (eles não conseguem representar um simples ou-exclusivo), esta ferramenta tornou-se pouco expressiva e a tecnolgia ficou adormecida por alguns anos, até que a criação de métodos de treinamento eficientes propôs configurações bem-sucedidas de redes neuronais utilizando mais de uma camada. Estas novas configurações resolveram muitos dos problemas com a representação de funções linearmente separáveis, como é o caso do ou-exclusivo ou o caso representado na seção 2.1. Neste último, reparamos que, se não houver neurônios na camada escondida, a separação nunca atingirá seu máximo, já que a função em questão não é linearmente separável.

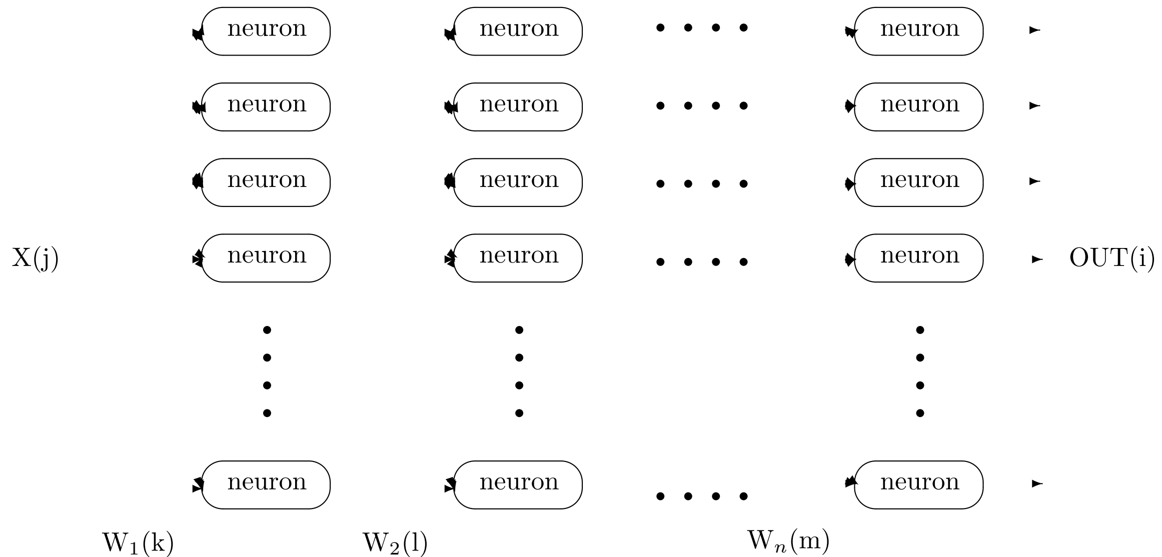
Redes com múltiplas camadas

Redes com múltiplas camadas são formadas colocando-se redes com camadas simples em sucessão. Um exemplo de rede neural com múltiplas camadas pode ser vista na figura 3.6. Nesta figura consideramos os neurônios com as respectivas funções de ativação como um só bloco (oval) e tentamos nos generalizar ao máximo na confecção do desenho. O vetor de entrada X é entregue à primeira camada, que, por sua vez, calcula os valores para este nível e passa sua saída para a camada seguinte, até chegar ao vetor final de saída OUT.

A identificação das linhas que separam diversos padrões como vimos na seção 2.1 é realizada pelos neurônios das camadas intermediárias, também conhecidas como camadas escondidas. Os neurônios da camada de saída servem somente para que haja uma interpretação da saída produzida pelos neurônios da camada escondida em valores conhecidos e pré-determinados durante a fase de treinamento.

Existem muitos métodos para o treinamento deste tipo de rede, supervisionados e não-supervisionados. Revisaremos aqui somente o método de retropropagação, visto que foi o que utilizamos em nosso projeto durante a fase de treinamento das redes.

Figura 3.6: Uma ANN com mais de uma camada.



3.1.3 Treinamento de redes de múltiplas camadas - retropropagação

Métodos eficientes de treinamento para redes neuronais de múltiplas camadas despontaram ao longo dos últimos anos. O método de retropropagação foi o primeiro deles, e teve seu nascimento vinculado ao nome de grandes pesquisadores que de forma quase concomitante o elaboraram.

O processo de cálculo das saídas de uma rede de múltiplas camadas é simples: coloca-se a entrada (vetor X) nos neurônios da primeira camada, eles calculam e geram uma saída para esta camada, que serve como entrada para a próxima, até que esta operação atinja a última camada e produza uma saída final para a rede. Chamar-se-á este passo de processamento de "passo propagado" denotando que os dados fluem na rede da entrada para a saída. Isto é independente do treinamento da rede e é implícito à sua natureza lógica e operacional.

O problema chega agora: Como elaborar um método eficiente que possa, de forma supervisionada, ajustar os pesos de uma rede de múltiplas camadas para que consigamos representar uma função qualquer (dado que esta possa ser representável por uma rede neuronal)? Esta questão vem do fato de que é possível pensar em um método eficiente para ajustar os pesos de uma rede de apenas uma camada. Por exemplo, é possível para cada vetor de entrada achar uma saída através da rede, compará-la a um valor alvo e ajustar os pesos de forma a minimizar este erro. Já para redes de múltiplas camadas fica difícil a utilização deste método visto que não possuímos os valores alvos para os neurônios situados em camadas escondidas. A resposta é um tanto simples: da mesma forma que podemos dar um passo propagado na rede achando a saída é possível pensar em um algoritmo que dê um passo retropropagado. De acordo com a saída que rede obteve para um vetor de entrada comparada com um valor-alvo para aquele vetor, re-calculamos os pesos de forma a minimizar a diferença obtida na comparação da saída da rede com o valor alvo de saída estipulado para aquele vetor de entrada.

Utilizam-se neste tipo de treinamento 2 vetores, chamados comumente de "par de treinamento". O primeiro vetor é o vetor de entrada; o segundo, a saída-alvo da rede ou saída que a rede, deve gerar para aquele vetor de entrada. O processo de treinamento pode ser assim entendido :

1. Calcula-se o passo propagado para um dado vetor de entrada;
2. Calcula-se a diferença entre o valor de saída da rede e o valor-alvo;
3. Utiliza-se esta diferença no ajuste dos pesos da rede de forma a minimizar o erro;

4. Retorna-se ao primeiro passo.

Não há garantias de que a rede se ajustará, repetindo a função que representará, de forma rápida, ou de que o número de passos de treinamento será pequeno. Apenas há garantias de que o número de passos requeridos é finito e que, quando este número for atingido, a rede estará apta a representar tal função a que se propõe. Métodos e adaptações para mais rápida convergência de redes neuronais existem, ainda que constituam parte dos pontos não abordados aqui. Maiores referências podem ser vistas em [13].

O passo retropropagado pode ser dividido em 2 subpassos como se segue:

Ajustando os pesos da camada de saída. Por possuirmos os valores-alvo de saída para cada neurônio da camada final poderemos ajustar facilmente os pesos para cada neurônio p na camada escondida j ligado ao neurônio q da camada de saída k utilizando o seguinte cálculo:

$$\delta = \frac{d(OUT)}{d(NET)}(Alvo - OUT) \tag{3.3}$$

$$\Delta w_{pq,k} = \eta \delta_{q,k} OUT_{pj} \tag{3.4}$$

$$w_{pq,k}(n+1) = w_{pq,k}(n) + \delta w_{pq,k} \tag{3.5}$$

Em 3.3 calculamos o que comumente é chamado de regra do delta, esta expressão reflete quanto estamos longe do alvo, isto é, usamos a derivada da função de ativação para chegarmos a um valor para δ que dependa linearmente da diferença de $(Alvo - OUT)$. Na equação 3.4, chegamos ao valor de mudança do peso, multiplicando por δ um valor η que expressa o coeficiente de aprendizagem ou a variante da velocidade que impomos à rede em seu treinamento[1]. Assim chegamos ao valor de mudança nos pesos da camada de saída, sendo concretizado em 3.5.

Ajustando os pesos das camadas escondidas. As camadas escondidas não possuem vetor-alvo; então, o processo de treinamento descrito anteriormente não pode ser utilizado. Esta falta de algoritmo de treinamento cabível foi uma das grandes responsáveis pela estagnação desta tecnologia nos anos 70. O Algorítmo de retroPropagação treina as camadas escondidas propagando o erro da saída através das camadas internas, ajustando os pesos, camada a camada.

As equações 3.4 e 3.5 são utilizadas para todas as camadas, de saída e escondidas; entretanto, para camadas escondidas δ deve ser gerado sem o benefício de um valor alvo. Inicialmente, δ é calculado para cada neurônio na camada de saída, como na equação 3.3. Ele é utilizado para ajustar os pesos que alimentam a camada de saída e depois retropropagados através dos mesmos pesos, gerando um valor de δ para cada neurônio da primeira camada escondida a contar da camada de saída. Estes valores de δ são, então, utilizados para ajustar os pesos desta camada e depois retropropagados para que ajustem os pesos das camadas anteriores, seguindo os mesmos princípios.

Considere um neurônio na camada escondida imediatamente anterior à camada de saída. No passo propagado, este neurônio, através do peso que conecta esta camada à de saída, cede o valor por si calculado à camada final. Durante o treinamento acontece o inverso: o valor calculado para δ é retropropagado através dos pesos que conectam este neurônio à camada de saída de tal forma que gerem um δ para este neurônio. Este delta é então utilizado para calcular os pesos deste neurônio. O valor para o δ da camada escondida pode, então, ser calculado como se segue:

$$\delta_{pj} = \frac{d(OUT_{pj})}{d(NET_{pj})}(\sum_q \delta_{q,k} w_{pq,k}) \tag{3.6}$$

Deste ponto é possível utilizar as equações 3.4 e 3.5 para calcular e ajustar os pesos do neuronio escondido.

[1] Deixa-se claro aqui que velocidade é diferente de qualidade no treinamento, um maior η pode vir a acarretar um treinamento mais demorado, tanto quanto um menor, dependendo da funçao objetivo da rede.

3.1.4 Redes Neurais e aspectos gerais

É possível mencionar alguns dos pontos que se destacam na utilização de redes neuronais em aplicações que exigem velocidade e segurança, são eles:

Robustez Redes Neuronais são capazes de operar ainda que alguns de seus neurônios parem de funcionar, gerando saídas coerentes e possuindo baixa degradação de “performance” em situações deste tipo. Este item parece ser bem interessante em ambientes onde possuímos condições extremas de operação como altas temperaturas e ambientes com alta taxa de radioatividade

Velocidade Uma vez que utilizam operações simples como multiplicações e somas, redes neuronais podem ser muito rápidas. Uma redução da velocidade de operação causada pela função de ativação pode ser sobreposta se utilizarmos tabelas de conversão ao invés de calcularmos o valor de função (OUT) no ponto (NET) por aproximação em séries[2].

Saída probabilística Esta característica é intrínseca à operação com redes neurais. Redes Neuronais, se não são treinadas à exaustão (o que quase sempre é inviável), ou treinadas de tal forma à reconhecerem todos os padrões possíveis para uma dada entrada, são incapazes de afirmar com total precisão sobre os dados de entrada, sugerindo em sua saída um vetor de probabilidades ao invés de uma afirmação tácita sobre a natureza de um evento (no caso da função de saída ser contínua). A utilização de um vetor de dados probabilísticos na identificação de partículas pode ser mais vantajosa que uma resposta binária sobre uma dada RoI ou um evento como indica [10]. Isto se deve basicamente ao fato de que estaremos tentando reconhecer nova física, que não se enquadra em nenhum padrão conhecido; logo, a identificação se fará baseada em dados probalísticos, não justificando a necessidade de um algoritmo que rotule eventos ainda não estudados.

Preço Redes Neuronais podem ser competitivas quanto ao custo por sua simplicidade lógica.

Aprendizado Redes de Neurônios Artificiais podem aprender tanto quanto forem treinadas. Isto é pontecialmente favorável em um ambiente onde desconhecemos as facetas de operação e no qual pode ser necessária a atualização quanto a novos padrões de entrada e saída.

Abstração Uma das virtudes de redes deste gênero é capacidade de abstrair qualidades de fontes extremamente ruidosas, exibindo alto poder de reconhecimento de entradas muito distorcidas pelos estágios anteriores.

Assim sendo, Redes Neuronais Artificiais constituem base segura e sólida onde é possível desenvolver aplicações para ambientes como este ao que propomos, por suas características únicas e grande robustez.

3.2 Computação distribuída e o sistema TN310

O sistema TN310 é um computador de processamento distribuído. Isto significa que sua Unidade de Processamento Central é dividida em várias partes, cada uma contendo um processador reponsável por gerenciar seus próprios recursos, que podem executar todas as mesmas ou diferentes funções ao mesmo tempo.

No caso da TN310, o sistema é dividido de tal forma que cada processador possua (e gerencie) um banco de memória particular e um coprocessador, que neste caso vem a ser um ADSP21020 da Analog Devices. Exporemos características e vantagens do processamento distribuído juntamente com uma explanação do equipamento no que segue.

[2] A maior parte dos algoritmos computacionais e bibliotecas é baseado em expansão em séries para o cálculo de funções complexas como $sin(x)$ ou $cos(x)$.

3.2.1 Terminologia

Durante as demais seções e capítulos os termos *processamento distribuído, processamento concorrente e processamento paralelo* significarão o mesmo, i.e., identificarão o tipo de processamento em que atividades são realizadas de forma distribuída entre diversos processadores, paralelizando o algoritmo para realizar uma dada tarefa.

O termo *speed-up* será utilizado para se referir ao fator de redução no tempo de processamento comparando-se o tempo gasto numa máquina com processamento distribuído com o gasto numa máquina com processamento sequencial.

3.2.2 Processamento Distribuído - uma solução à demanda de velocidade

Utiliza-se processamento seqüencial atualmente, em vários dos laboratórios espalhados em todo mundo por causa de vários fatores limitantes, entre eles:

- Preço;
- Carência de material e pessoal treinado para operar e computar de forma distribuída;
- Ausência de demanda.

O processamento seqüencial tem fundamentação no fato de que toda atividade computacional pode ser realizada seqüencialmente, de forma lógica e ordenada. Alguns problemas podem surgir deste tipo de implementação, entre eles a ausência de equipamento rápido o suficiente para executar uma tarefa ou falta de tecnologia que suporte tal volume de dados.

A criação da computação distribuída veio fechar este buraco tecnológico provendo uma solução inteligente para situações onde há demanda de maior velocidade e um maior fluxo de dados sem que haja a necessidade de criarmos nova tecnologia. É possível dividir a tarefa em partes que possam ser executadas quase que independentes, com maior ou menor grau de dependência, de tal forma que possamos juntar 2 ou mais nós de processamento e utilizá-los para um único objetivo (a realização da tarefa). Desta formak elimina-se a necessidade de utilizarmos unidades de processamento super-rápidas (que, na maioria das vezes, são caras ou inexistentes) e alivia o volume de dados suportados por uma única CPU.

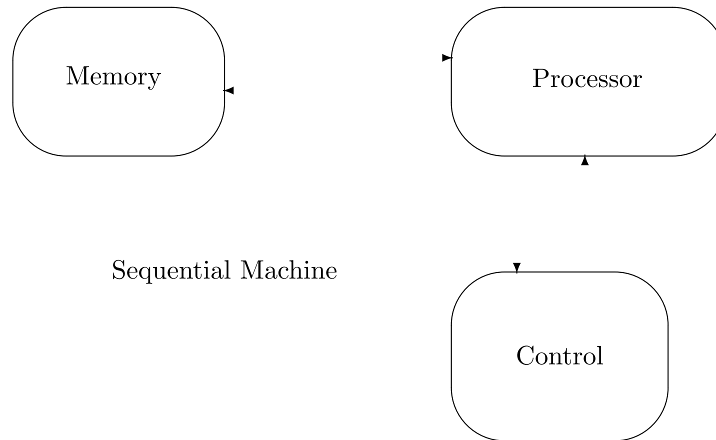
O processamento distribuído é uma forma alternativa de processamento e representa um mercado tecnológico em ascensão. Para que se processe distribuidamente não é necessário a utilização de equipamento especial, basta que seja possível a utilização de várias CPU-s concomitantemente. Como exemplo podemos ver que é possível distribuir a execução de uma tarefa por uma rede como a Internet bastando que haja um sistema operacional responsável por administrar o fluxo de dados e tarefas pela rede.

Aplicações em computação distribuída não são especiais, havendo a necessidade de entender uma dada tarefa e abstrair desta a máxima paralelização atingível. Em outras palavras, deve-se dividir a aplicação de forma a maximizar o poder de processamento disponível e minimizar a carga em cada processador (ou nó de processamento[3]). Esta divisão pode ocorrer de várias formas, dependendo da aplicação, como será elucidado mais à frente.

Algumas aplicações são predominantemente seqüenciais, de forma que não podem ser executadas paralelamente; como exemplo vemos a contagem de segundos em um intervalo de tempo. Analogamente, há atividades que devem ser executadas paralelamente para que haja máximo desempenho; como exemplo vemos a multiplicação de um vetor por uma constante. Neste caso, os diversos valores do vetor podem ser multiplicados ao mesmo tempo sem perda de informação. Assim, torna-se possível agilizar a execução desta e outras tarefas apenas aumentando o número de nós operando concorrentemente(ao mesmo tempo).

[3]O termo nó de processamento é mais elucidativo quanto ao fato de que um processador pode administrar muitos recursos ao mesmo tempo, logo tornando-se um ponto de processamento independente. O termo processador muitas vezes esconde este fato.

Figura 3.7: A arquitetura sequencial ou SISD



3.2.3 Arquiteturas Paralelas

Existem 4 tipos de arquiteturas computacionais segundo a classificação de Flynn, que é baseada na multiplicidade das instruções e dos dados em um nó de processamento [14]. Os 4 tipos são:

S.I.S.D. *Single Instruction Single Data* ou Instrução Única Dados Únicos;

S.I.M.D. *Single Instruction Multiple Data* ou Instrução Única Dados Múltiplos;

M.I.S.D. *Multiple Instruction Single Data* ou Instrução Múltipla Dados Únicos;

M.I.M.D. *Multiple Instruction Multiple Data* ou Instrução Múltipla Dados Múltiplos.

O primeiro item abrevia a classe de processamento conhecida comumente como sequencial. Neste tipo de processamento, um único processador é responsável por tratar os dados residentes em um banco de memória, de tal forma que cada dado está direta e exclusivamente associado a uma única instrução. Um diagrama esquemático pode ser visto na figura 3.7.

Na arquitetura de processamento SIMD, todos os processadores recebem a mesma instrução para processamento, mas executam suas funções sobre dados diferentes. Neste tipo de configuração, uma rede de interconexão entre o banco de memória e os processadores é usada. Os diversos subbancos de memória podem ser acessados concorrentemente e o acesso do mesmo endereço por mais de um processador é possível, podendo causar um "gargalo" no processamento. Outro nome para arquiteturas deste tipo é máquina vetorial.

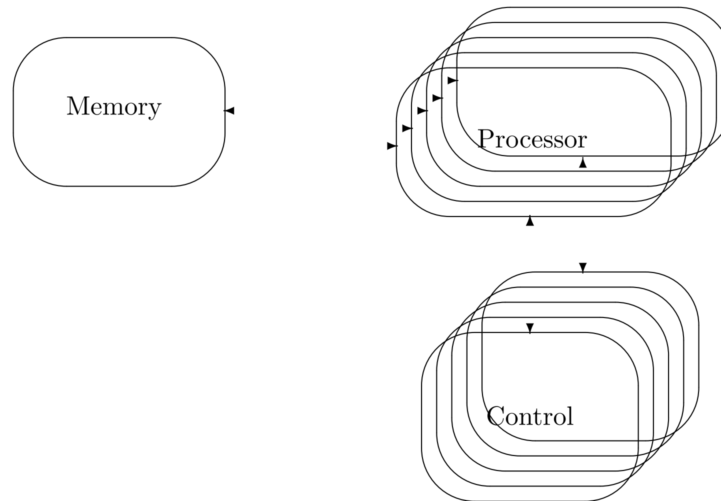
Em arquiteturas do tipo MISD existe um processamento em cima de um mesmo dado segundo várias instruções, resultando em um *pipeline* de dados, isto é, um mesmo dado passa por diversos tipos de processamento. Equipamentos com esta arquitetura são em número reduzido.

A classe de arquiteturas MIMD representa aquelas máquinas que podem operar sobre dados distintos, concorrentemente, com diferentes instruções ou ainda sobre o mesmo dado com diferentes instruções formando um *pipe*. Este tipo de arquitetura é flexível e pode vir configurada de 2 maneiras:

1. **Memória Global (figura 3.8)** todas as unidades de processamento acessam uma única memória compartilhada. O acesso a bancos diferentes pode ser concorrente, ainda que o acesso ao mesmo banco possa causar um aumento no tempo de processamento final. A comunicação neste tipo de configuração é rápida, visto que a memória é local a todos os processadores.

2. **Memória Distribuída (figura 3.9)** os processadores têm cada um seu banco de memória privado. Isto pode facilitar a inclusão de um sem-número de processadores em uma única

Figura 3.8: Esquema representativo de uma arquitetura MIMD com memória global.



máquina sem perda de "performance", ao contrário da configuração com memória global. Uma desvantagem pode ser observada quanto ao acesso de dados em memória não-local, podendo gerar um aumento considerável no tempo de execução de uma tarefa.

Redes de Conexões

Computadores com arquiteturas paralelas são caracterizados pela utilização de redes de comunicação de alta "performance". Redes de comunicação são requeridas em aplicações distribuídas pois processos alocados em diferentes nós podem, em muitos dos casos, ter que se comunicar. A comunicação entre os nós distintos de uma mesma arquitetura pode ser feita de várias maneiras, revisaremos algumas aqui.

A organização do sistema de *hardware* em ambientes distribuídos também pode caracterizar uma arquitetura. Em outras palavras, o esquema de conexões entre os processadores e a memória é fator marcante em arquiteturas paraleas sendo utilizado para sua classificação e respeitado durante a fase de alocação das tarefas nos nós de processamento.

Em sistemas com memória global, a rede de conexões é caracterizada por uma malha de chaves que conectam os processadores aos bancos, de forma que se processadores quiserem passar informações entre si, devem escrever e ler de um endereço comum. Já em sistemas com memória distribuída, a passagem de dados deve ser feita por passagem de pacotes através de canais de comunicação entre os processadores; isto implica numa rede de conexões voltada ao roteamento de pacotes, ao invés de otimizada para o acesso de bancos de memória. As figuras 3.10 e 3.11 exemplificam a passagem de informação nestes 2 tipos de sistema.

Em especial, redes que utilizam roteamento de pacotes (memória distribuída) normalmente o fazem utilizando chaves muito rápidas ou conexões especiais entre processadores. Se utilizarem chaves, a configuração de chaves (figura 3.11) pode ser complexa ou simples dependendo diretamente do número de processadores do sistema (e do número de saídas e entradas disponíveis para comunicação de cada um) e velocidade de operação requerida para o sistema como um todo. Se não utilizarem passagem de pacotes por chaves, estas redes deverão ser arquitetadas de forma específica para uma dada aplicação. Isto quer dizer que aplicações que possuem uma limitação na forma de comunicação entre os processos devem ser alocadas em arquiteturas cabíveis. A título de exemplo, podemos considerar aplicações do tipo *master-slave*, que compõem-se de uma aplicação-mestre que supervisiona o trabalho e cede

Figura 3.9: Esquema representativo de uma arquitetura MIMD com memória distribuída.

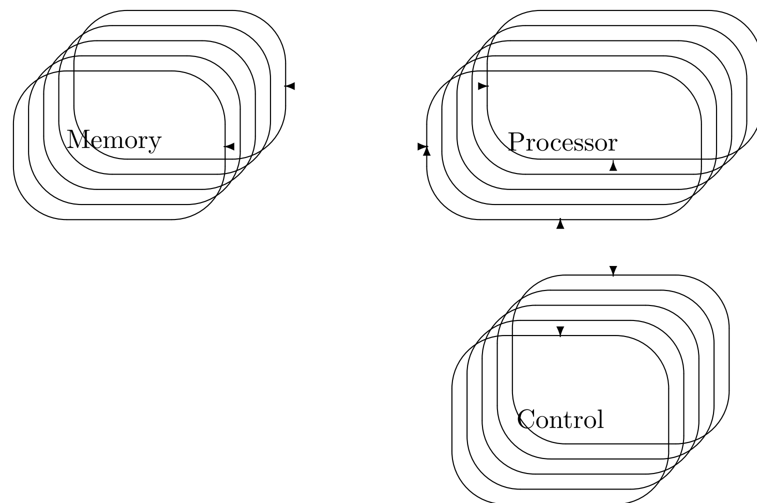


Figura 3.10: A passagem de dados em um sistema com memória global.

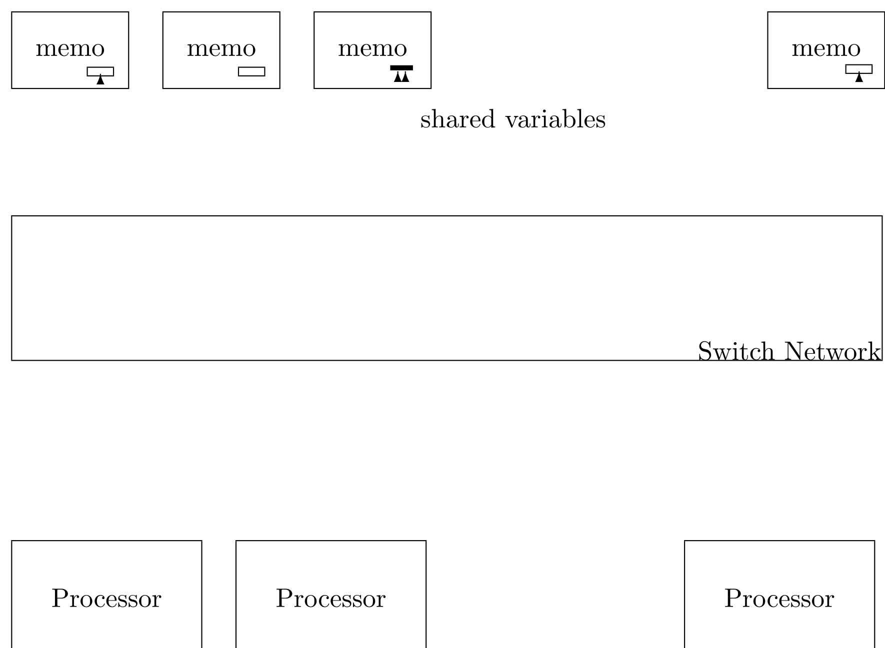
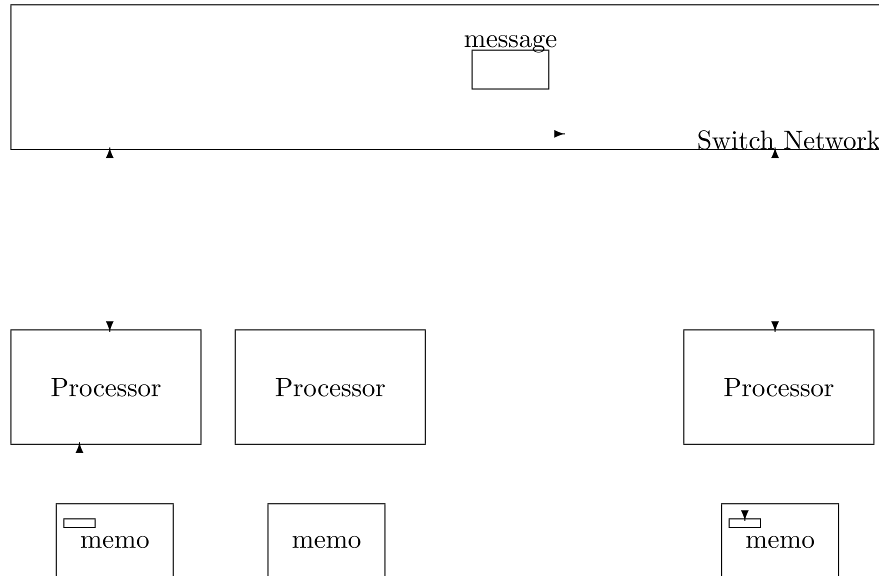


Figura 3.11: A passagem de dados(por chaves) em um sistema com memória distribuída.



dados aos *slaves*. Estas aplicações podem ser diagramadas como na figura 3.12. Seria inviável, por exemplo, a implementação de uma aplicação deste tipo numa rede como na figura 3.13(a). Esta última também traz alguns outros esquemas de conexões entre nós de processamento formando arquiteturas distribuídas. Em 3.13(b) vemos uma conexão tipo anel; em (c), uma topologia totalmente conectada. No esquema (e) é possível observar uma topologia de conexões conhecida como hipercubo e em (d) uma conexão tipo árvore.

Sincronismo. De uma forma geral, a comunicação entre redes que utilizam pacotes para a passagem de dados entre nós distintos de processamento pode ser feita síncrona ou assincronamente. Quando a passagem é feita sincronamente, o processador que passa o dado espera até que o processador recebedor reconheça que recebeu o pacote para prosseguir em sua tarefa, ficando até então parado, esperando tal confirmação. Se a passagem é feita de forma assíncrona, o processador que envia o dado segue em sua operação normal, sem esperar um sinal de confirmação de outros nós de processamento.

Dinamismo. As arquiteturas paralelas podem ainda ter 3 classes distintas:

- estáticas - redes cuja interconexão entre os processadores é imutável;
- pseudodinâmicas - redes cuja topologia é alterada logo antes da execução de uma tarefa;
- dinâmicas - redes que podem ter sua topologia reestruturada durante a execução de uma tarefa ou aplicação.

3.2.4 O Sistema TN310

A TN310 é um computador paralelo tipo MIMD com memória distribuída. O sistema em questão abriga 16 nós baseados no padrão HTRAM (*High performance TRAnsputer Modules*), que se comunicam entre si utilizando-se de chaves rápidas (tipo STC104).

Cada nó de processamento HTRAM (tamanho 4) desse sistema abriga um *transputer* (INMOS T9000), um banco de memória RAM privada de 8 Megabytes, um ADSP-21020 da Analog Devices e

Figura 3.12: Um diagrama de uma aplicação *master-slave*

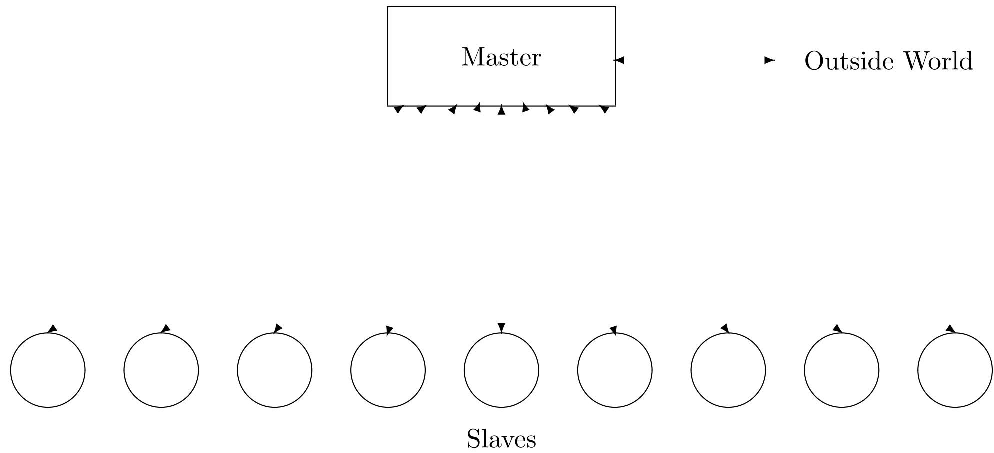
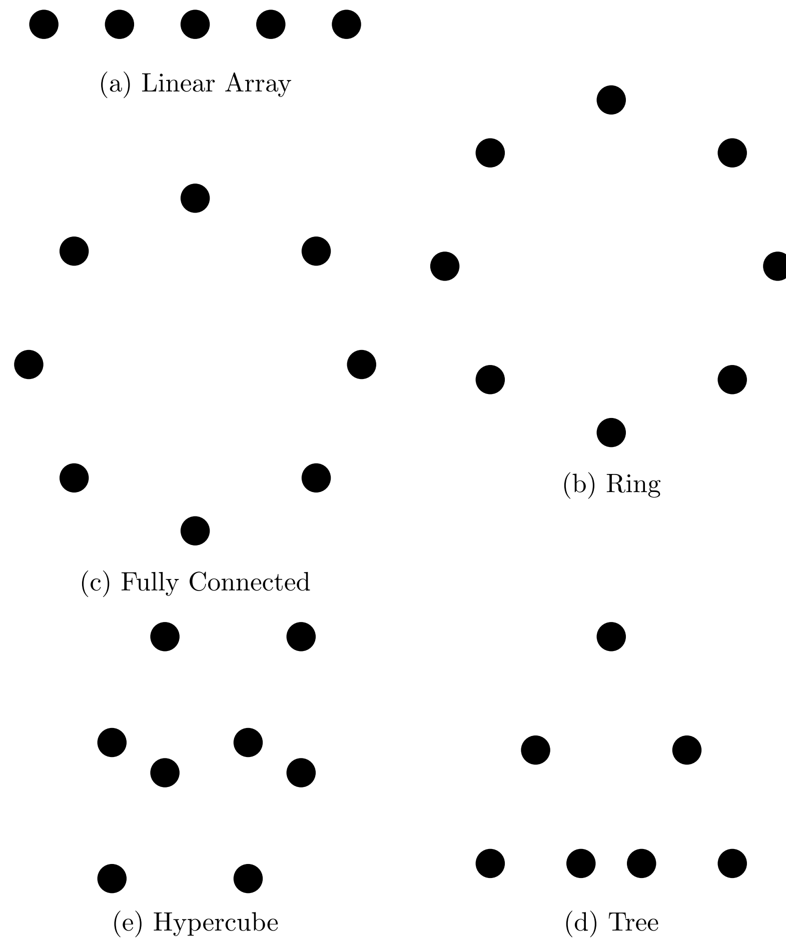


Figura 3.13: Algumas configurações topológicas de redes com memória distribuída que não utilizam chaves. Repare que, para passar dados para processadores mais remotos, estes dados devem passar por outros processadores que apenas os repassarão adiante.



(a) Linear Array

(b) Ring

(c) Fully Connected

(e) Hypercube

(d) Tree

um *buffer* de comunicação entre o transputer e o ADSP de 256 Kilobytes. O ADSP ainda possui uma memória de acesso privado de 196 Kilobytes.

A entrada e saída de dados na máquina é feita através de um sistema hospedeiro, no nosso caso um Personal Computer rodando MS-DOS.

O Transputer T9000

O *Transputer* T9000 é um microprocessador CMOS de 32-bits desenhado para ser utilizado em aplicações que requeiram alta "performance" combinada a alta integração e simplicidade de uso. Algumas de suas características incluem um *clock* de 20 MegaHertz, 16 Kilobytes de *On-Chip RAM* que pode ser utilizada em 3 diferentes configurações: toda em *cache*, metade *cache*/metade memória de acesso rápido ou totalmente configurada como memória de acesso rápido. Possui, ainda, uma unidade aritmética para números em ponto-flutuante (reais) de 64 bits e uma interface de memória altamente programável.

Transputers são processadores versáteis que possuem lógica de baixo nível otimizada para a comunicação interprocessadores. Os *transputers* da linha T9000 possuem 4 canais independentes de comunicação chamados de *DS-Links*, que podem operar concorrentemente recebendo ou passando dados de/para 4 diferentes localidades. Cada *DS-Link* suporta um fluxo máximo de dados de 80 Megabytes por segundo.

A comunicação entre *transputers* se dá através de canais implementados em baixo nível (*hardware*). As instruções de máquina indistinguem a comunicação feita dentro de um mesmo transputer, por diferentes processos, ou feitas entre diferentes nós de processamento. No T9000 a comunicaçao se dá através de canais especiais chamados de **canais virtuais**, que são instâncias de canais reais multiplexados por um *Processador de Canais Virtuais (VCP)* nos 4 *DS-Links*. O VCP é implementado em baixo nível e permite que o programador se despreocupe com relação à administração de canais e protocolos de comunicação entre os diversos *transputers*.

O T9000 ainda exibe capacidade de multiprocessamento através de *time-sharing*. Sua arquitetura suporta a criação e agendamento de **qualquer** número de processos concorrentes, eliminando quase que totalmente a utilização de um sistema operacional. A figura 3.14 mostra um diagrama de blocos do *Transputer* INMOS T9000 retirada de [15].

O DSP *on-board*

As HTRAM-s desse sistema também abrigam um DSP (*Digital Signal Processor*) operando como um coprocessador matemático. A arquitetura básica destes coprocessadores incluem três unidades computacionais independentes: uma Unidade de Lógica Aritmética, um multiplicador/acumulador de ponto fixo e um *shifter*. Esta arquitetura permite rápido processamento, visto que a unidade pode multiplicar em ponto flutuante, armazenar e procurar em um único ciclo que dura 40 ns.

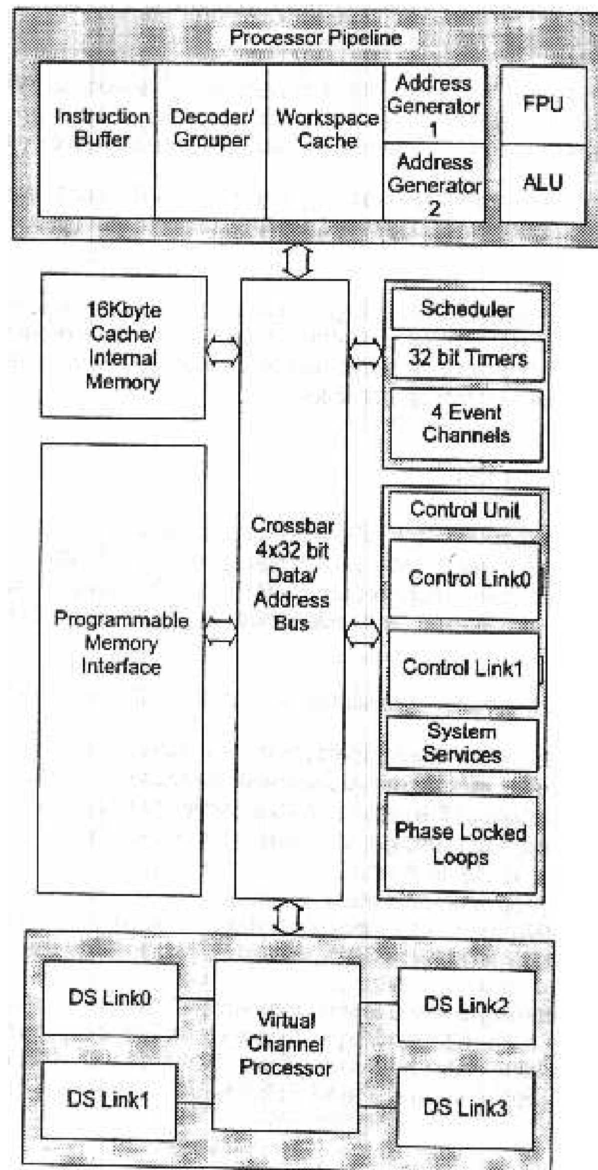
O DSP residente nas HTRAM-s se comunica única e exclusivamente através do transputer na placa, utilizando para tal o *buffer* comum de 256 Kilobytes. A utilização do DSP para processamento matemático pode otimizar o tempo total de processamento gasto em cada nó, por sua versatilidade.

Conectando HTRAM-s - A chave STC104

Os 16 nós de processamento HTRAM do sistema TN310 são conectados através de chaves STC104. As chaves são capazes de conectar 64 canais de comunicações seriais entre si, formando até 32 *links* de comunicação entre os diversos processadores. Os *links* operam concorrentemente a transferência de 32 pacotes de dados sem que haja interferência entre eles. O tempo máximo de chaveamento para um pacote através destas chaves é de 1 μs, significando que esta demorará no máximo 1 μs para direcionar um pacote que chega a um de seus canais. A banda passante das chaves é de 100 Megabytes por segundo, fazendo com que estas não representem um peso à comunicação entre 2 nós distintos.

A figura 3.15 mostra a conexão entre os diversos nós de processamento do sistema TN310. Esta configuração pode ser alterada durante a inicialização do sistema (ele é pseudodinâmico), sendo possível

Figura 3.14: A arquitetura básica do *Transputer* T9000.



o ocultamento de canais e de pontos de processamento. Todos os processadores podem se comunicar entre si através de um número de chaves. Por exemplo, o nó 0 pode se comunicar através da chave 0 e da chave 1 ao nó 7 no mesmo cartão.

A comunicação será tão lenta quanto maior for o número de chaves por que o pacote tenha que passar. Assim o tipo de comunicação mais custosa seria entre os nós 0 e 8 pois os pacotes de dados passariam por 4 chaves (0-1-6-5). O programador deve atentar para este fato se quiser a otimização máxima do processamento no sistema.

A comunicação com o sistema *host* é feita através da chave 0, que está conectada aos 4 nós da HTRAM 0, indicando que esta é a mais próxima do sistema hospedeiro.

Os outros *links* das STC104-s são utilizados para o fluxo de dados de controle de inicialização e operação.

O Sistema Hospedeiro

Um Personal Computer (PC) rodando MS-DOS/Windows 3.1 é o sistema hospedeiro para a TN310, sendo utilizado também como ambiente de desenvolvimento de aplicativos para este sistema. A passagem de dados é feita por uma placa especial que conecta o PC às diversas entradas e saídas da TN310.

Apresentaremos agora o ambiente de desenvolvimento para a TN310 e as ferramentas disponíveis. Quando forem mencionados nomes de arquivos ou diretórios, durante posteriores argumentações na continuação deste documento, subentende-se que o leitor compreenda que estaremos nos referindo à árvore de diretórios do PC hospedeiro acoplado à TN310 no Laboratório de Processamento de Sinais (LPS, Centro de Tecnologia/UFRJ, sala H-220).

3.2.5 Desenvolvendo aplicações para a TN310

O desenvolvimento de aplicações para o sistema TN310 se dá através do "InMOS C ToolSet". Este conjunto de ferramentas abrange configuradores de *hardware* em baixo nível utilizando *Network Description Language (NDL)* e um conjunto de bibliotecas que tornam possível a programação em ANSI C concorrente.

Outras ferramentas do *Toolset* incluem uma interface e ferramentas para operação sobre um "micro-kernel" chamado de RuBIS (*Runtime Basic Interface System*) que atua como um pequeno sistema operacinal rodando no sistema. Este "micro-kernel" é programado em ANSI C, fazendo com que aplicações que rodem sobre esta camada tornem-se mais lentas. O "*toolset*" do sistema TN310 ainda inclui também ferramentas para programação em PVM, uma linguagem mais flexível e portátil, ainda que aplicações que rodem sobre esta camada sejam ainda mais lentas pois constitue uma camada construída acima da interface RuBIS. O diagrama da figura 3.16 pode ser elucidativo quanto ao exposto.

Uma vez que estamos diante de uma aplicação que necessita de máxima otimização e velocidade, ainda que mantendo uma simples interface que possa facilmente ser controlada e mantida, optamos pelo desenvolvimento de nossas aplicações na primeira camada do ToolSet, utilizando C concorrente e as bibliotecas deste conjunto.

Para que se programe o sistema TN310 são necessárias 2 fases distintas de programação. Na primeira fase, um enfoque de baixo nível da aplicação deve ser utilizado; nesta fase, o programador configurará a máquina (usando *Network Description Language*). Na segunda fase um enfoque de alto nível, onde o programador possa adaptar a configuração de sua aplicação ao *hardware* disponível e pré-configurado, deve ser aplicado.

Estas duas fases de programação são totalmente exclusivas e não interferem uma na outra. A primeira fase (deve-se executar a programação da aplicação seguindo esta ordem) consiste em determinar as características do *hardware* disponível, isto é, canais de *hardware*, número de nós de processamento disponíveis, memória disponível em cada nó, sinais de controle etc. Na segunda fase, utiliza-se o sistema configurado na primeira fase para rodar a aplicação em C Concorrente. Veremos um pouco de

Figura 3.15: TN310, conexão dos diversos nós HTRAM (dados recolhidos pela ferramenta T9SPY).

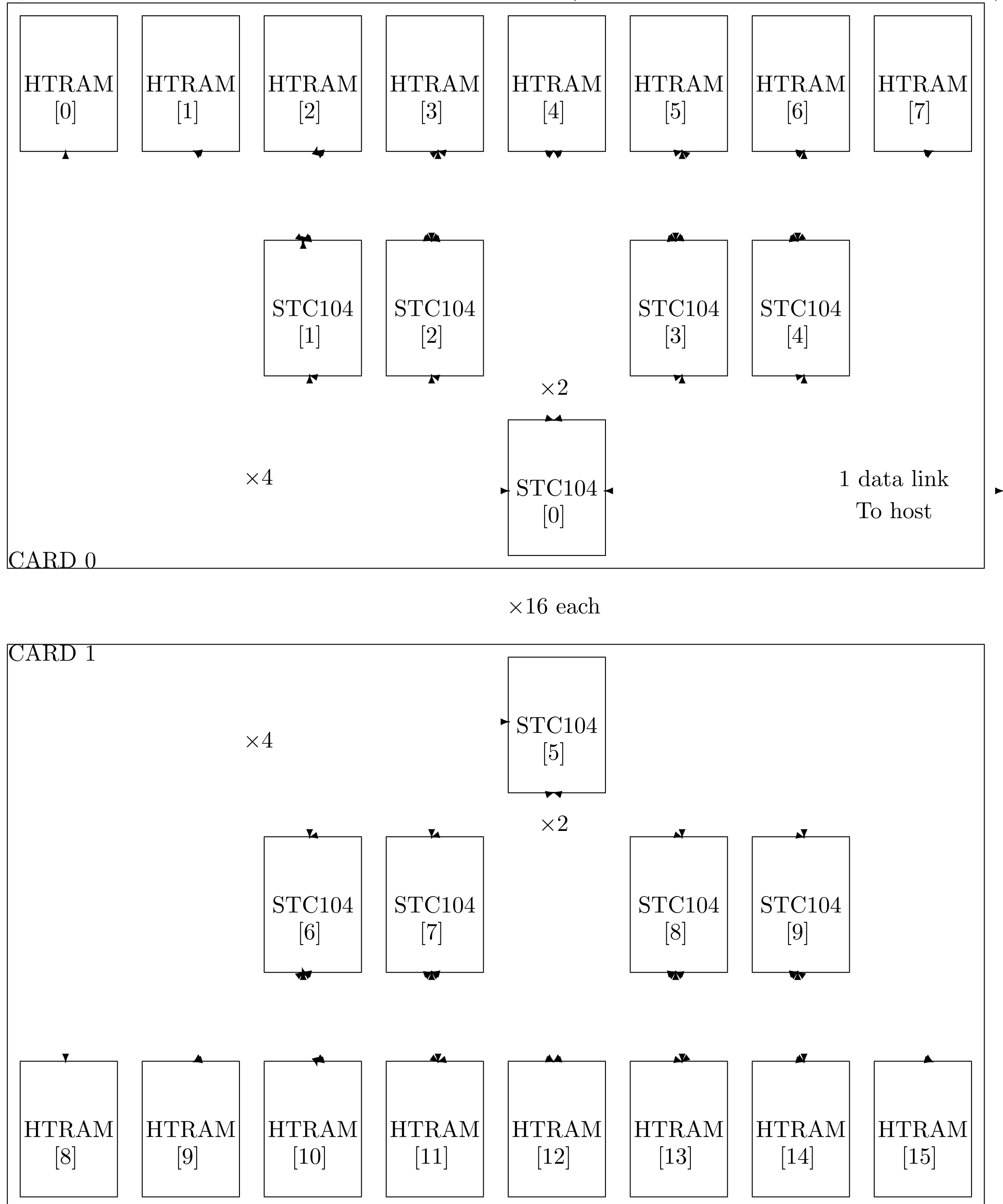
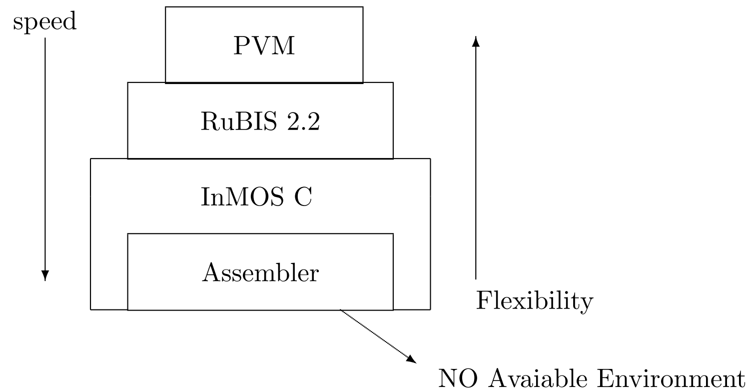


Figura 3.16: As diversas camadas de programação disponíveis no *T9000 Toolset*.



cada uma destas fases no que segue.

Fase 1 - A configuração em baixo nível usando NDL

Esta fase é composta de 2 subfases. Na primeira, existe uma descrição do *hardware* disponível utilizando NDL. Esta fase inclui a configuração de quaisquer parâmetros dos componentes da rede de processadores e chaves. No que se refere aos processadores, este passo inclui a descrição da interface de tempo da memória, a configuração da memória *cache*, as instruções de inicialização e ponteiros para áreas de memória. Para as chaves, a descrição abrange rotulação e detalhes de roteamento. Para ambos os dispositivos (processadores e chaves), o arquivo de configuração de *hardware* deve conter a descrição completa da conexão entre estes elementos e o sistema hospedeiro, incluindo conexões de dados e controle.

A linguagem de descrição de redes (*Network Description Language*) torna possível a utilização de *software* (alto nível de programação) para que se determine o comportamento de um sistema de baixo nível (*hardware*). A descrição é feita utilizando-se de comandos especiais (padrão NDL) que podem ser armazenados em arquivo texto. É possível editar tal arquivo em qualquer tipo de editor padrão ASCII (7-bit). Normalmente estes arquivos são gerados por desenhistas de placas e sistemas e, na maioria das vezes, permanecem inalterados durante toda a vida útil de um sistema. Isto se deve ao fato de que as descrições de redes geralmente tentam aproveitar todos os recursos disponíveis, de tal forma que qualquer outra descrição somente seja um subconjunto da descrição primária.

No caso da TN310 a instalação da máquina (arquivos de sistemas no PC hospedeiro) inclui, além da descrição padrão que inclui todo o sistema, uma descrição para operação com apenas 2 processadores. Estes arquivos encontram-se na árvore de diretórios do drive C sob o nome de `hardfiles\tn310.ndl`[4] e `hardfiles\tn310-2p.ndl`, respectivamente.

Como exemplificação de um arquivo NDL para a TN310, o apêndice A possui o arquivo tn310-2p.ndl transcrito e com alguns comentários.

As diretivas da rede são carregadas durante a inicialização do sistema, no *bootstrap* da máquina. Para que seja possível a utilização do arquivo NDL de descrição do ambiente um arquivo de configuração associando a aplicação construída em C Concorrente e o arquivo NDL deve ser criado, como se segue.

[4]A figura 3.15 traz o esquema de conexões gerado pela inicialização do sistema com esta descrição.

Unindo partes, o arquivo de configuração da aplicação

A utilização de um arquivo de configuração para que seja possível a criação de um arquivo de inicialização global traz inúmeras vantagens, entre elas o benefício de jamais termos que lidar com o arquivo NDL diretamente, somente o referenciando e, assim, eliminando questões de controle de fluxo de dados e alocação de endereços.

O arquivo de configuração da aplicação é um arquivo-texto comum (extensão *cfs*), especificamente produzido para uma única aplicação e compilado junto com esta. Neste arquivo, há uma linha que descreve a configuração da rede a ser utilizada (arquivo NDL) e outras linhas que descrevem a maneira como as subaplicações (ou tarefas de uma aplicação) se comunicarão pela rede, a memória disponível para cada tarefa e onde será alocada cada uma destas.

Durante a inicialização os dados do arquivo NDL são codificados e um "*reset*" é dado na rede, de tal forma que haja uma arrumação desta para a execução da aplicação. Daí, o arquivo CFS é codificado e os endereços onde serão alocadas as tarefas são repassados ao código da aplicação, o que faz com que as aplicações se posicionem corretamente no sistema. Por fim, são repassados às aplicações os endereços relativos aos canais de comunicação entre as diversas tarefas, previamente alocados pelo arquivo de configuração da aplicação. A partir deste momento a aplicação é disparada e pode rodar sem problemas.

De forma simplificada, o arquivo de configuração da aplicação é responsável por traçar um esquema topológico de alocação das tarefas e canais de uma aplicação, juntamente com alguns de seus parâmetros. O apêndice B pode ser elucidativo quanto a tal arquivo, pois mostra um exemplo comentado.

Tarefas, aplicações e o T9000 ToolSet

As diversas tarefas de uma aplicação são programáveis usando o *C ToolSet* que acompanha o sistema TN310. Este conjunto de ferramentas é constituído de bibliotecas especiais que tornam a linguagem C padrão ANSI capaz de executar funções implícitas a sistemas concorrentes. Estas funções incluem alocação de canais, comunicação entre tarefas e criação de processos[5] concorrentes numa mesma tarefa.

O C ToolSet também abriga ferramentas que são utilizadas para rodar e depurar uma aplicação construída em qualquer uma de suas camadas (veja figura 3.16) [16]. As aplicações podem rodar em ambiente Windows. O aplicativo `IRUN` é utilizado para rodar a aplicação no sistema; ele carrega um arquivo "bootável" na TN310, que é gerado a partir do código em C, o arquivo CFS e o arquivo NDL durante o processo de compilação.

Os aplicativos `IPROF`, `IMON` e `INQUEST` são respectivamente um *profiler*, um monitor de carga sobre os nós de processamento e um depurador interativo. Opções especiais de compilação [17] devem ser aplicadas ao código durante esta fase para que haja correta utilização destes aplicativos.

O ToolSet ainda traz um analisador de redes, o `T9SPY`, que traduz as informações de arranjo de uma TN310 inicializada por um arquivo NDL em informações compreensíveis (arquivo texto) que podem ser utilizadas para traçar diagramas como o da figura 3.15.

A compilação do código em C é feita utilizando-se de compiladores e *linker* especiais (`icc` e `ilink`) que fazem parte do ToolSet. A união dos arquivos para formar uma aplicação é feita por aplicativos dedicados, o `inconf` e o `icollect`. Cada um destes aplicativos tem *flags* que devem ser adicionados ou retirados de acordo com o objetivo da compilação. Por exemplo, a utilização dos flags `T9000` e `GAMMAE` no compilador e no *linker* é necessária sempre, pois indica o tipo e versão de *transputer* do sistema. Flags como `G` e `GA` para o compilador e *linker*, respectivamente, somente são necessários quando a compilação for direcionada à depuração e não devem ser utilizados normalmente, pois aumentam o tempo de execução de tarefas e processos.

Normalmente, utiliza-se um `makefile` para a compilação através do aplicativo `make.exe`. Este aplicativo direciona os blocos de compilação e *flags* com regras e diretivas de tal forma que o usuário

[5]Processos são subtarefas que podem ser disparadas por uma tarefa e operar concorrentemente ou não com ela. Processos são subinstâncias de uma tarefa. Não abordaremos este tópico aqui.

Tabela 3.1: As bibliotecas não-ANSI no InMOS C ToolSet (Retirado de [18]).

Header file	Description
boolink.h	Boot link Channel information
channel.h	Channel Handling
dos.h	DOS specific operations
fnload.h	Dynamic Code Load functions
host.h	Host System information
hostlink.h	Host Channel information
iocntrl.h	Low level fiel handling
mathf.h	`float` versions of maths and trigs functions
misc.h	Miscellaneous functions
process.h	Process startup, hadling and control
semaphor.h	Semaphore hadling
stdiored.h	Reduced library string formattion functions

somente precise executá-lo para que consiga o arquivo "bootável" da aplicação. Cada aplicação (incluindo diferentes direcionamentos de uma mesma aplicação como a compilação para depuração) deve possuir seu próprio `makefile`. Um `makefile` exemplificado encontra-se no apêndice C.

O InMOS C

A linguagem de programação que permite máxima velocidade em aplicações para o sistema TN310 é o InMOS C. Esta linguagem é uma versão do ANSI C adaptado a sistemas concorrentes. Diversas bibliotecas são utilizadas, complementando o conjunto de bibliotecas-padrão, de forma que haja máxima portabilidade e flexibilidade das aplicações na adaptação a novos sistemas.

Muitas funções e tipos novos são acrescentados; discutiremos aqui alguns destes, assim como algumas das limitações que impõem. A tabela 3.1 mostra todas as bibliotecas disponíveis no ToolSet. Para uma listagem completa de tais bibliotecas, incluindo a descrição detalhada de novos tipos e funções o leitor deve consultar [18].

Canais. Canais são a forma de comunicação entre as diversas tarefas de uma aplicação. Como elucidado na seção 3.2.5, durante a compilação os endereços dos canais identificados no arquivo CFS são passados para as tarefas através de *software*. Esta passagem de endereços se concretiza durante a execução do código, quando se criam os **canais de *software***. Canais de software são instâncias de canais virtuais que, por sua vez, são administrados por hardware (VCP-seção 3.2.4). Isto retira a carga administrativa do programador sobre os canais utilizados numa aplicação.

A criação dos canais é implementada no arquivo CFS; porém, para que estes canais estejam visíveis à aplicação é necessário "capturar" os endereços de tais canais disponíveis após a inicialização. Isto pode ser implementado com a função `get_param()` de `misc.h`. Esta função recolhe os parâmetros descritos no arquivo CFS dentro da seção `interface` em cada processo. A seguir, um exemplo cruzado (mostrando o arquivo CFS e o arquivo em C) de utilização de tal função:

No arquivo cfs:

```
...
process(stacksize=500k, heapsize=1000k, interface(input HostIn,
output HostOut, input From_pipe, output To_pipe)) receiver;
...
```

Passamos vários parâmetros para a tarefa `receiver`; neste caso todos são canai; devemos capturá-los na aplicação C, veja:

```
#include <misc.h>
#include <channel.h>
...
Channel *input1 = get_param(1);
Channel *output1 = get_param(2);
Channel *input2 = get_param(3);
Channel *output2 = get_param(4);
...
```

Canais são ponteiros para áreas de memória no processador virtual de canais, por isto seus identificadores recebem um "*" (inerente à sintaxe da linguage C).

É possível a criação de vetores e matrizes de canais, visto que são tipos de dados tão robustos quanto quaisquer outros tipos em C. Rotinas especiais foram desenvolvidas pela TELMAT e podem ser encontradas nos exemplos, em `c:\exercice\tp*.*`.

Existem rotinas especiais para a operação de canais ou vetores de canais, entre elas a função `ProcAlt()` de `process.h` que pode detetar em um conjunto de canais de entrada (tipo `input`) algum que queira se comunicar com a aplicação. Uma explicação mais detalhada do funcionamento de tais rotinas pode ser visto em [15] e [18].

Comunicação entre tarefas. É possível utilizar canais para comunicação entre tarefas usando as funções de `channel.h`. Esta biblioteca inclui um conjunto extenso de funções dedicadas à comunicação de inteiros, números reais, caracteres ou mensagens de tamanho variável.

A utilização é simples: depois de declarados, os canais podem ser utilizados para mandar ou receber quaisquer volume de dados de outras aplicações. Lembra-se aqui que canais são unidirecionais e devem ter sua correspondência em duas tarefas distintas. Em outras palavras, para cada canal de entrada em uma aplicação deve haver um canal de saída correspondente na mesma ou em tarefas diferentes.

A comunicação por canais é **totalmente bloqueante**. Isto quer dizer que a tarefa que tenta enviar/receber uma mensagem fica parada neste ponto até que outra tarefa receba/envie a informação esperada[6]. Fica óbvio que canais não correspondidos causam um *dead-lock* na tarefa.

A tabela 3.2 mostra algumas das funções e tipos em `channel.h`. A figura 3.17 exemplifica a comunicação entre duas tarefas de uma mesma aplicação.

Todas as funções descritas na tabela 3.2 possuem um equivalente otimizado para comunicação em canais reais implementados via *hardware* (caso específico de aplicações em InMOS C). Elas são mais rápidas e suas funções são as mesmas; para saber seus nomes apendicione a palavra "`Direct`" em cada função da tabela 3.2. Exemplo: `DirectChanInInt`.

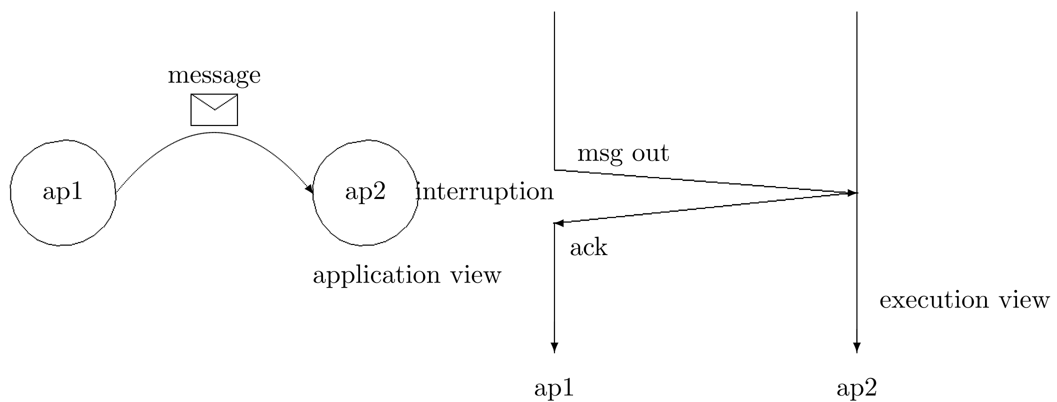
3.2.6 Resumindo a seção...

O sistema TN310 é um computador com sistema distribuído por 16 nós de processamento baseado no padrão HTRAM (tamanho 4). Cada nó possui 1 transputer T9000, 1 ADSP 21020, 8 Megabytes de memória RAM e 256 Kilobytes de *buffer*(única ponte do DSP com outras partes do sistema). Este 16 nós estão totalmente conectados uns aos outros por uma rede de chaves muito rápidas, as STC104.

Segundo a classificação de Flynn, esta é uma máquina MIMD com memória distribuída. Nestes casos, a comunicação entre os diversos processadores se dá através da passagem de mensagens pela rede de chaves. Uma vez que os processadores estão conectados através de diferentes números de chaves, a comunicação entre pares distintos de processadores pode ser mais ou menos custosa em termos temporais.

[6]Se os protocolos de comunicação por canais não forem compatíveis (exemplo: uma tarefa passa um inteiro enquanto a outra está esperando um caractere) pode haver um *dead-lock*.

Figura 3.17: Esquema de uma comunicação entre tarefas utilizando canais.



Os códigos nas aplicações ap1 e ap2 responsáveis por esta troca:

ap1.c

```
#include <misc.h>
#include <channel.h>
...
Channel *OUT = get_param(3);
int k = 4;
...
ChanOut(OUT, &k, sizeof(k));
```

ap2.c

```
#include <misc.h>
#include <channel.h>
...
Channel *IN = get_param(1);
int receive;
...
ChanIn(IN, &receive, sizeof(receive));
```

Tabela 3.2: Algumas das funções e tipos em `channel.h` (Retirado de [18]).

Function	Description
ChanIn	Inputs a message on a channel
ChanInChar	Inputs a character on a channel
ChanInFloat	Inputs a real on a channel
ChanInInt	Inputs an integer on a channel
ChanOut	Outputs a message on a channel
ChanOutChar	Outputs a character on channel
ChanOutFloat	Outputs a real on channel
ChanOutInt	Outputs an integer on channel
ChanVIn	Inputs a variable length message on a channel
ChanVOut	Outputs a variable length message on a channel

Type	Description
Channel	The channel type

A implementação de aplicações se dá em 2 niveis independentes, a configuração do *hardware* e o dimensionamento da aplicação feito em InMOS C, que é uma versão alterada do ANSI C, criada para suportar funções típicas de sistemas concorrentes. Os 2 níveis de implementação da aplicação se juntam através da compilação, utilizando um arquivo de configuração que determina a localização de cada tarefa na rede e aloca os canais nos VCP-s de cada processador. VCP-s são elementos de *hardware*, implementados no próprio *transputer*, que gerenciam a utilização dos canais reais do processador (chamados de DS-Links) multiplexando inúmeros canais concorrentes (canais virtuais) e removendo da responsabilidade do programador tal tarefa.

O InMOS C ToolSet possui funções e ferramentas que podem ser utilizadas para a programação de aplicações no segundo nível. As ferramentas incluem aplicativos Windows (entre eles depuradores e executores) e ferramentas de compilação. Dentre as funções adicionadas à linguagem ANSI C estão aquelas que lidam com canais, existem funções específicas para a troca de variáveis simples e versões otimizadas que podem rodar sobre canais virtuais implementados em *hardware*, como é o caso da programação em InMOS C.

Além da possibilidade da programação em C é possível utilizar PVM ou rotinas embutidas em um μKernel chamado RuBIS. Existe uma troca entre flexibilidade e tempo de execução quando trocamos a camada de implementação.

3.3 Técnicas de Paralelismo

Discutiremos aqui algumas técnicas de paralelização levadas em consideração durante o desenvolvimento de nossa aplicação (simulação de parte do segundo nível de validação para o experimento ATLAS/LHC).

Durante o desenvolvimento de aplicações que operem em um ambiente distribuído, é necessário que o programador observe princípios básicos durante a fase de construção do programa. Estes princípios são técnicas que permitem o desenvolvimento de aplicações paralelas.

O termo “paralelo” implica operações realizadas concorrentemente. Concorrer significa disputar. O termo indica que em ambientes onde há paralelismo as diversas tarefas que compõe uma aplicação são executadas de forma que o máximo de cada tarefa seja executado utilizando um mínimo de tempo, como numa disputa. Operações concorrentes devem ser organizadas da melhor forma possível para reduzir o tempo de processamento global, este é o objetivo final da utilização de ambientes distribuídos.

Paralelismo é a execução simultânea de instruções em múltiplos nós de processamento. A con-

corrência ou paralelismo pode ser atingida durante a construção de uma aplicação para a execução de uma tarefa de 2 maneiras, através de competição ou cooperação. Num ambiente distribuído cooperativo as diversas instruções são executadas como em um *pipeline* de instruções; desta forma, se o sistema deve tratar um dado segundo 3 tipos de filtragem consecutivas é possível utilizar processadores operando cooperativamente para realizar o trabalho. Assim, quando o processador acabar de operar sobre um dado, o segundo começa e depois o terceiro, mas ao mesmo tempo que um filtro libera o dado para o próximo estágio, já pode carregar outro dado diferente, reduzindo o tempo de execução final. No caso de trabalho competitivo, temos vários processadores operando com as mesmas instruções sobre conjuntos diferentes de dados; à medida que os nós de processamento finalizam seu conjunto de instruções, recebem mais dados.

Dois fatores estão envolvidos na paralelização de uma aplicação:

- Fator "Quantitativo": introduz a noção da quantidade de paralelismo inerente a uma aplicação (subjetivo);
- Fator Qualitativo: constitui a qualificação da atividade quanto a 3 diferentes técnicas de paralelismo:
  1. Paralelismo de Controle;
  2. Paralelismo de Dados;
  3. Paralelismo de Fluxo.

As aplicações podem ser paralelizadas por divisão em muitas tarefas ou pela duplicação de instâncias de processamento capazes de processar competitivamente sobre os mesmos tipos de dados. A melhor forma de paralelizar uma aplicação é utilizando um híbrido destes dois métodos, de forma a reduzir ao dependência entre diferentes tarefas. A dependência é um fator que indica uma seqüela na paralelização de uma aplicação pois processos dependentes atrasam-se tanto quanto necessário para manterem esta relação. Exemplo de dependência pode ser visto na construção de uma casa: a construção do telhado depende da construção das paredes, que depende da construção da fundação; logo, estas atividades não são paralelizáveis devendo ser executadas sequencialmente.

3.3.1 Paralelismo de Controle

Esta técnica de paralelização é utilizada em aplicações nas quais as diversas instruções podem ser separadas em blocos diferentes sem que haja necessidade de um bloco auxiliar outro na execução de sua parte da tarefa. Isto quer dizer que os processos são totalmente independentes. Possíveis dependências ocorrem quando o trabalho executado por um bloco de instruções não pode prosseguir até que outro bloco libere um resultado (o exemplo da casa, descrito acima); a dependência de um bloco na execução de uma tarefa de outro bloco introduz uma sequencialização da aplicação. O objetivo neste tipo de paralelização é reduzir ao máximo este tipo de dependência.

O tempo de execução final é avaliado segundo o tempo de execução de cada bloco e as relações de dependência entre eles. Se os blocos forem totalmente independentes, o tempo de execução será igual ao tempo de execução do maior dos blocos.

A figura 3.18 pode ser elucidativa com relação a tal técnica. Na parte (a), vemos uma aplicação na qual podemos dividir sua execução em blocos independentes. Na segunda parte, parte (b), vemos uma aplicação que possui dependência na execução de alguns de seus blocos, levando a trechos de sequencialidade e, portanto, reduzindo a eficácia da paralelização.

3.3.2 Paralelismo de Dados

Existem muitos casos com que nos deparamos durante nossa vida onde a execução de uma tarefa se dá repetidas vezes. A exemplo podemos ver o ato de descascar batatas; cada batata pode ser diferente, ainda que o trabalho seja o mesmo, descascá-las. É possível paralelizar uma tarefa como

Figura 3.18: O paralelismo de controle num diagrama de blocos.

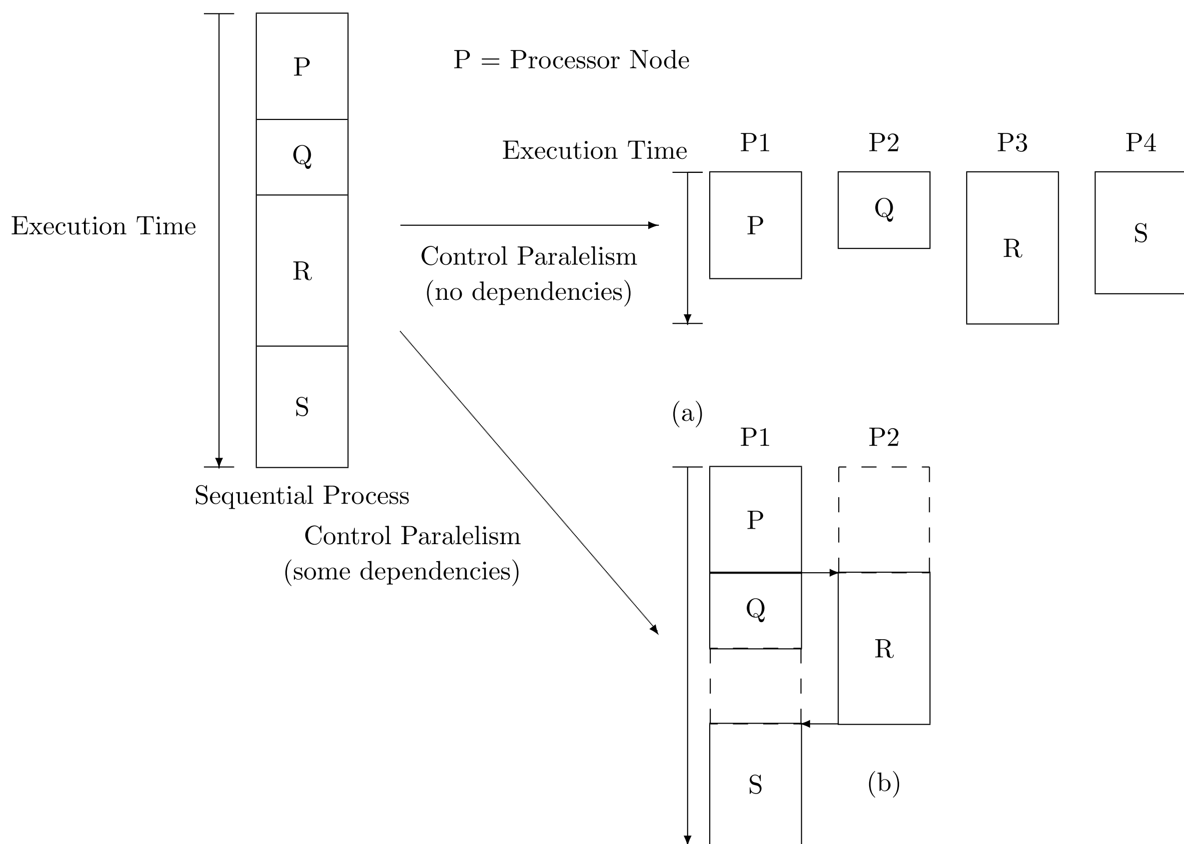
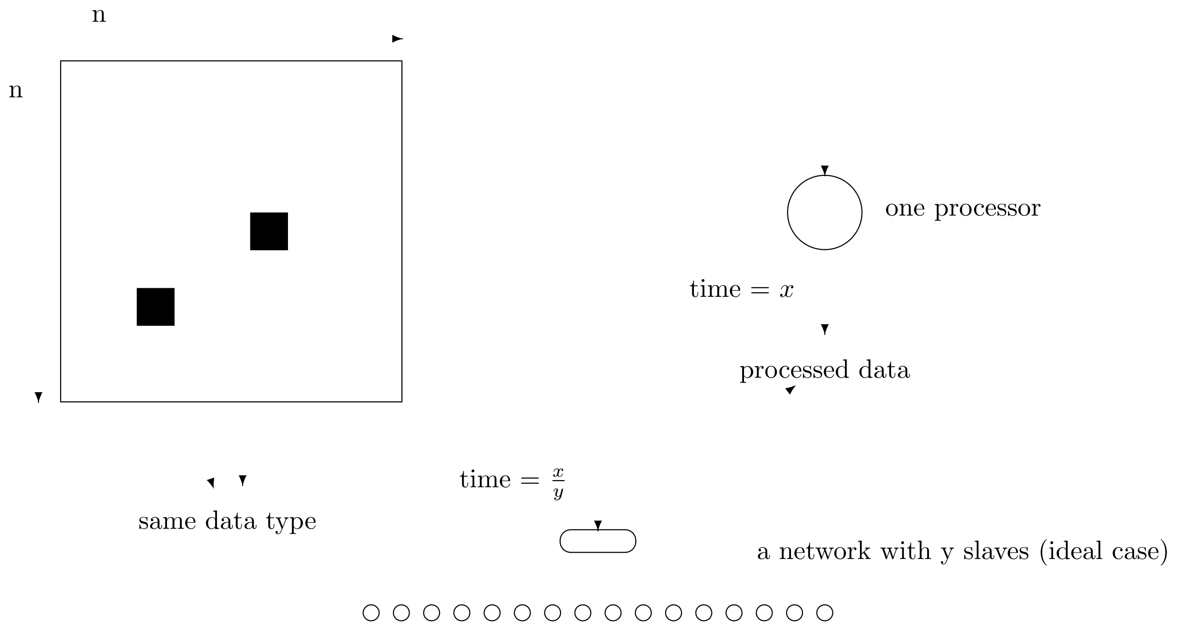


Figura 3.19: Um diagrama mostrando a paralelização de uma aplicação utilizando a técnica de paralelismo de dados.



esta se utilizarmos mais de uma pessoa trabalhando com as batatas, quanto mais pessoas, maior será o *speed-up* da tarefa. Outro exemplo é o de entregar pizzas; quanto mais entregadores, mais rápido o serviço acaba.

A tradução de tais tarefas para programadores deve ser: "Uma aplicação é dado-paralelizável se estamos trabalhando com matrizes ou vetores onde pequenas porções das matrizes ou escalares dos vetores devam receber o mesmo tratamento, seja ele qual for: filtragem, multiplicação ou cálculo de um função não-linear utilizando aquele pedaço da matriz ou o escalar que compõe o vetor". É possível reduzir o tempo de execução, se paralelamente executarmos estes algoritmos de filtragem (ou multiplicação ...) em cada parte do dado-alvo.

No paralelismo de dados existe, via de regra, uma hierarquia entre as aplicações. Isto acontece porque normalmente os dados são passados às aplicações que executam a função-alvo (multiplicação, filtragem ou qualquer que seja), geralmente chamadas de *slaves*, via uma aplicação controladora de todo o processo, a aplicação *master*. A "seqüencialização" da aplicação acontece quando temos que passar os dados da aplicação mestra aos escravos. Esta passagem de dados aumenta o tempo de execução da tarefa, assim reduzindo o *speed-up* final.

Aqui, a concorrência é um fator que pode ser bem visualizado. Escravos que recebem dados mais trabalhosos demoram mais para processar. Isto significa que estas tarefas (escravas) receberão menos dados que outras que consigam processar mais rapidamente[7].

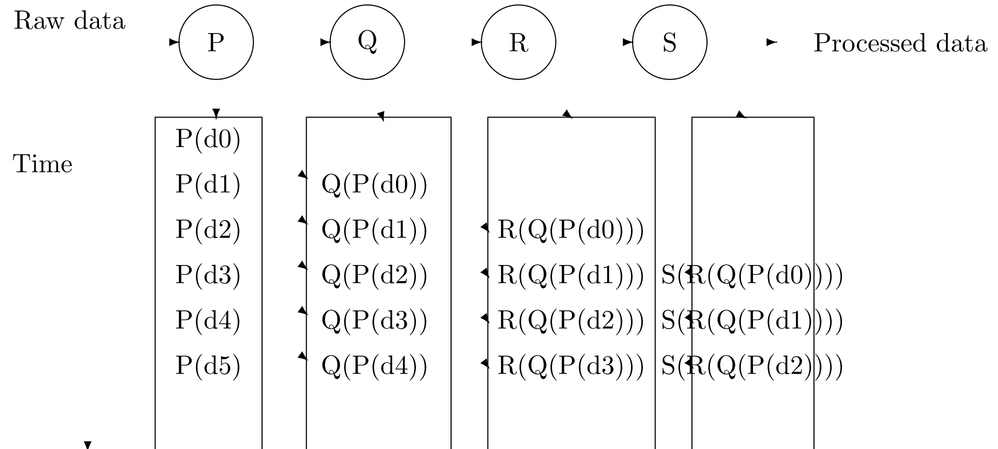
A figura 3.19 mostra um esquema da utilização de paralelismo de dados em uma aplicação genérica.

3.3.3 Paralelismo de Fluxo

Algumas aplicações são baseadas em um modelo tipo *pipeline*. Um exemplo é a construção de uma casa. Inicialmente constrói-se a fundação, depois as paredes, o encanamento e tubulações, e aí faz-se o telhado. Se tivermos que construir 100 casas, o trabalho é paralelizável, ainda que haja dependência

[7] Um ambiente misto, com diferentes tipos de nós de processamento pode, por exemplo, ocasionar neste tipo ocorrência.

Figura 3.20: Um *pipe* de dados. O paralelismo de fluxo é aconselhável aqui.



entre as tarefas. É possível destacar um grupo que somente construa fundações, este grupo constrói a primeira fundação, depois constrói a segunda e, ao mesmo tempo, outro grupo responsável por erguer paredes as ergue sobre a fundação da primeira casa, já construída. Quando o grupo de fundações atinge a vigésima casa, o grupo de construções de paredes estará na décima-nona (se as atividades forem executáveis no mesmo espaço de tempo), o grupo do encanamento estará na décima-oitava e o de construção de telhado, na décima-sétima. Desta forma, reduzimos o tempo de execução da tarefa.

Este tipo de paralelização, formando uma espécie de "linha de montagem", é chamado de paralelismo de fluxo. A seqüencialização é inserida quando temos tarefas que demoram mais para ser realizadas que outras, desta forma atrasando todo o bloco. Neste tipo de paralelismo cada subtarefa executa uma parte do todo a ser executado para a aplicação. A figura 3.20 pode ser elucidativa quanto a este tipo de paralelização.

3.3.4 O que usar na aplicação?

Infelizmente um problema geralmente não contém somente um tipo de paralelismo, e a utilização de todas as técnicas aqui mencionadas torna-se fundamental para que atinjamos o maior *speed-up* possível.

O que utilizar em cada momento deve levar em conta alguns fatores de seqüencialização da aplicação já paralelizada; são eles:

- Controle
  1. A dependência entre as diversas subtarefas;
  2. A sequencialização devido à falta de nós de processamento (recursos)
- Dados
  1. O tamanho do dado (comparativamente à memória disponível)
  2. Dados escalares
- Fluxo
  1. Estado transitório (o tempo que demora até que o *pipe* comece a operar);

3.4 Os dados utilizados neste projeto

Nesta seção estaremos comentando a procedência e validade dos dados utilizados no projeto. Uma vez que estamos interessados em informações de tempos de execução, os dados utilizados foram em número reduzido. Somente houve necessidade de utilizar dados reais durante a fase de testes da unidade de decisão global, identificando a RoI.

Foram simulados dados que partiriam de 4 detetores distintos segundo quatro possíveis decaimentos de Higgs em outras partículas. Os detetores são: um calorímetro altamente granulado, um *Preshower* segmentado (SCT), um Detetor de Transição de Radiação (TRT) e um Detetor de Múons. Os eventos simulados foram gerados usando um algoritmo computacional conhecido como Monte Carlo, assegurando a aleatoriedade e diversidade dos mesmos. Ainda, as partículas analisadas nesta fase (decisão global) englobam 4 tipos de reações, representadas nas equações 3.7, 3.8, 3.9 e 3.10.

$$Higgs \Longrightarrow ZZ \Longrightarrow 4e^{-}. \tag{3.7}$$

$$Higgs \Longrightarrow ZZ \Longrightarrow 2jets + 2e^{-}. \tag{3.8}$$

$$Higgs \Longrightarrow ZZ \Longrightarrow 2e^{-} + 2\mu. \tag{3.9}$$

$$Higgs \Longrightarrow ZZ \Longrightarrow 4\mu. \tag{3.10}$$

Estas reações sobreviveram ao primeiro nível de chaveamento, que também foi simulado. O espaço de variáveis a serem analisadas engloba identificadores de evento, RoI-s e chaveamento, variáveis físicas como momento e ângulo e variáveis de decisão como segundo momento, energia de isolaçao etc.

Capítulo 4

Implementando os algoritmos

Neste capítulo estaremos descrevendo as técnicas utilizadas para a implementação de 2 aplicações na TN310. A primeira é a unidade de decisões globais, operando como identificadora da RoI. Esta implementação, como será descrito, seguiu diferentes fases. Utilizamos a implementação da unidade de decisão global para que nos familiarizássemos com o sistema TN310 e pudéssemos nos sentir mais à vontade quando fôssemos implementar o segundo nível de validação.

A segunda aplicação é o próprio 2ºnível de validação para o experimento ATLAS/LHC. Nesta implementação utilizamos as técnicas conhecidas e entendidas com a implementação da unidade de decisões globais para que nos aproximássemos mais rapidamente da solução do problema. Em todas as soluções expostas, tentamos abordar o sistema TN310 como capaz de suportar as taxas de dados do 2ºnível de validação, isto é, usando o sistema como um autêntico **simulador**.

Neste capítulo nos resumiremos a abordar as implementações, deixando os resultados e ponderações sobre estes para o capítulo 5.

4.1 A Unidade de Decisão Global (GDU) - Identificando a RoI

Duas razões nos levaram a iniciar a implementação do segundo nível de validação pelo sistema de decisões globais. A primeira já foi citada e se deve à necessidade de ambientação com o sistema como um todo. A segunda se deve ao fato do *trigger* do segundo nível ser totalmente dependente da decisão da unidade de decisão global e não haver sentido em simulá-lo sem tal unidade totalmente operacional.

Uma vez que estávamos nos familiarizando com o equipamento, decidimos implementar inicialmente o código em C responsável pela execução da aplicação em apenas um dos 16 nós de processamento disponíveis. Desta forma estaríamos liberando o peso de ter que implementar uma aplicação paralela. De posse deste algoritmo totalmente testado, estaríamos, então, aptos a construir um sistema utilizando o paralelismo inerente ao equipamento.

4.1.1 A Implementação da GDU em 1 nó

Inicialmente descreveremos a função executada pela unidade de decisões globais, antes de propriamente entrarmos em sua implementação. Durante esta discussão nos limitaremos a expor o quadro relativo às dificuldades de implementação, visto que uma discussão do sistema já foi feita na seção 1.3.4.

A GDU deve receber as características extraídas de uma RoI pelos Extratores de Característica e baseada nelas emitir uma opinião sobre a natureza da partícula que excitou aquela RoI. Esta atividade, segundo o discutido na seção 2.1, pode ser realizada utilizando-se redes neurais artificiais [8].

Usando ANN-s. A entrada da rede neural será um vetor contendo todas as características extraídas e esta dará uma saída representativa da probabilidade de uma dada partícula ter excitado aquela região

Tabela 4.1: As eficiências (em %) obtidas durante a fase de teste de treinamento da unidade de decisões globais para os diferentes tipos de decaimento. Os valores para píons são muito precisos pois possuíamos uma amostra em número reduzido deste tipo de partícula, 1 ou 2 por arquivo (para as outras partículas este número passa de 1500 por arquivo).

	$4e^-$	$2e^- + 2j$	$2e^- + 2\mu$	4μ
e^-	97.45	99.50	94.75	-
$jets$	93.40	96.13	96.40	100
π^-	-	0	-	-
μ	-	94.44	100	99.95

de interesse que, como revisado, também é uma característica favorável. Em termos de implementação em C, temos que a rede neural terá um vetor com 12 posições contendo as 12 características extraídas (CE-s) (números reais) e como saída a indicação da probabilidade de encontrarmos elétrons, jatos, píons ou múons (vetor de 4 números reais).

A Rede Utilizada.

Partindo de uma implementação para a GDU em ambiente UNIX utilizando o pacote JETNET[1] e rotinas especiais, pudemos chegar a uma configuração de pesos que satifizesse a um valor mínimo de erro na saída. Esta configuração de rede corresponde a uma rede com 12 entradas, 6 neurônios na camada escondida (hidden layer) e 4 neurônios na camada de saída. É entendido que os neurônios responsáveis pela separação dos dados nos diversos subtipos (elétrons, jatos, píons e múons) são os da camada escondida e a camada de saída somente é responsável por evidenciar, interpretar esta separação.

Os pesos encontrados para a rede (que minimizam o erro na saída) foram distintos para os diferentes tipos de decaimento apontados nas equações 3.7, 3.8, 3.9 e 3.10. Um esforço no sentido de generalizar este resultado obtendo um conjunto de pesos genérico para todos estes tipos de decaimento não foi realizado. Isto se deve a 2 fatores importantes:

1. Estamos interessados em "performance" e não em qualidade dos resultados importando somente o algoritmo como um todo;

2. Existem outros 2 tipos de decaimento a serem treinados, e não haveria sentido em produzir uma rede especializada em somente 4 dos 6 tipos indicados.

Por questões de referência, a tabela 4.1 pode ser consultada para que tenhamos noção da qualidade de resposta do sistema de separação.

A configuração desta rede foi totalmente transportada para o sistema hospedeiro e nos baseamos nesta configuração para construir o ambiente neuronal em questão.

As características extraídas não possuem equivalência quanto à sua magnitude; logo, uma normalização é requerida, mesmo utilizando uma função de ativação não-linear como sugerida em [13]. Também foi utilizado um ajuste de *threshold* nos neurônios. Um diagrama do neurônio utilizado na aplicação pode ser visto na figura 4.1. Já o diagrama da rede com os neurônios pode ser visto na figura 4.2.

A função de ativação e sua implementação. A implementação da função de ativação (utilizamos a tangente hiperbólica) não foi feita utilizando rotinas disponíveis em linguagem C (`tanh()`). Ao

[1] Pacote para simulação e treinamento de Redes Neurais artificiais em múltiplas configurações. A interface é via FORTRAN 77.

Figura 4.1: O neurônio utilizado na rede da figura 4.2.

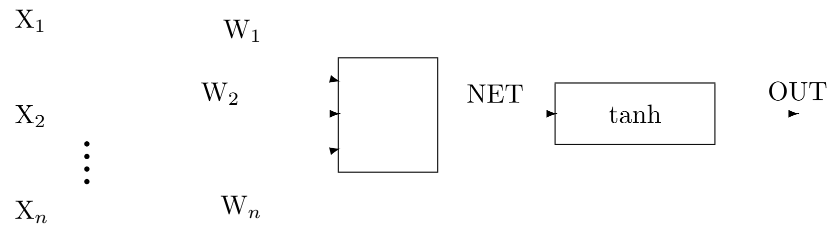
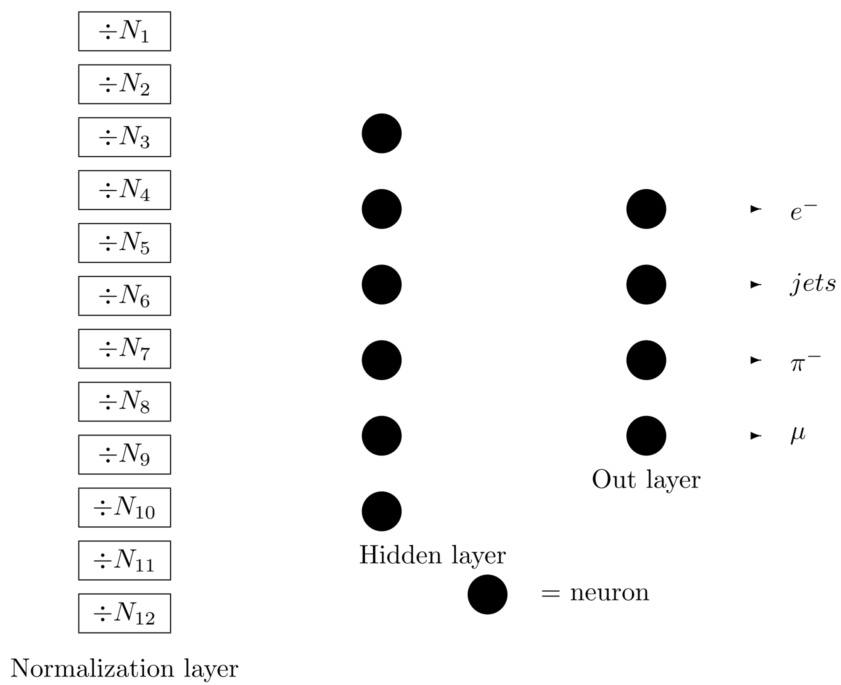


Figura 4.2: A rede para a GDU implementada na TN310.



invés, utilizamos uma *Look-up Table (LuT)*. Este procedimento mostra-se mais rápido pois, ao invés de executarmos operações algébricas, estaremos apenas utilizando uma espécie de conversão de valores segundo uma tabela. Esta tabela pode ser, por exemplo, armazenada em um vetor e, quando quisermos fazer a conversão de algum valor, consultamos a posição correta deste vetor. Um exemplo de consulta na tabela pode ser visto na figura 4.3; nesta figura, vemos a conversão de um valor segundo uma tabela armazenada em um vetor; o código em C equivalente a esta consulta também está incluso.

O código em InMOS C para rodar no ambiente TN310 encontra-se no apêndice D. Uma vez que estamos tentando copiar os resultados encontrados na implementação utilizando o ambiente UNIX (e o JETNET), concordamos que estaríamos satisfeitos se as taxas de eficiência das 2 implementações fossem iguais. Somente consideramos este parâmetro, pois o valor da rede está em sua eficiência de separação. Mínimas diferenças nas saídas são esperadas devido a 2 fatores:

1. O cálculo utilizando expansão de séries para a função de ativação (`tanh()`) é mais preciso que o que utiliza uma tabela de procura (precisão bem determinada e avaliada);

2. Durante a escrita dos pesos para um arquivo, o pacote JETNET os trunca, sendo impossível a replicação exata dos resultados ali obtidos, mesmo se utilizarmos novamente este pacote.

No entanto, como veremos, estas diferenças não influenciaram na qualidade de resposta da rede.

4.1.2 A implementação em até 16 nós

Uma vez tendo implementado e validado o sistema estaríamos aptos a utilizar todo o potencial do equipamento tentando otimizar a execução do algoritmo de decisão global paralelizando-o.

Nosso objetivo é construir um módulo que pudesse ser totalmente transferível para uma aplicação maior, ou seja, construir um módulo de decisão global realocável, isto é, que possa ser utilizado durante o projeto do segundo nível inteiro.

As dificuldades a serem encontradas seriam as relativas às comunicações entre as diversas tarefas alocadas nos nós de processamento, assim como a construção do mapa organizacional da atividade. Dois enfoques poderiam ser utilizados:

1. Tentar otimizar o tempo de execução da rede num processo global utilizando todos os processadores em função de um único algoritmo;

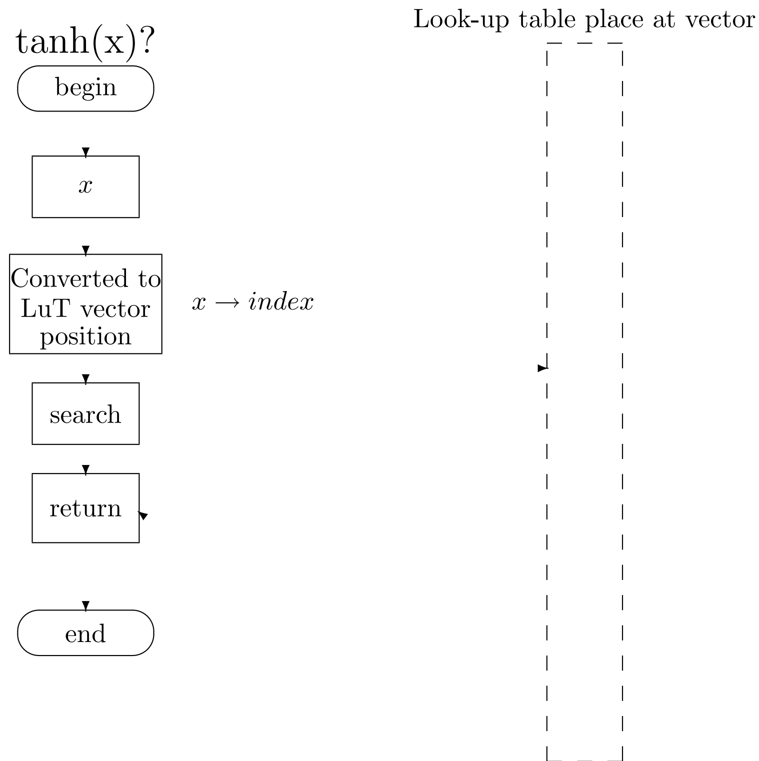
2. Utilizar cada nó de processamento disponível para implementar uma cópia da rede que trabalharia em paralelo.

O segundo enfoque foi o abordado aqui, pois estamos interessados em modularidade, e não em uma otimização que consumisse todos os recursos da máquina. Um segundo motivo é a existência de DSP-s *on-board* em cada HTRAM; seria possível, no futuro, sua utilização para a redução do tempo de processamento sem que precisássemos dedicar mais de um nó com tamanha capacidade de processamento em uma atividade tão simples.

Para utilizarmos o segundo enfoque visualizamos que estávamos diante de uma aplicação tipo *master-slave* (paralelismo de dados) e, assim, a construção de um processo supervisor (*master*) se faria necessária. A abordagem quanto ao paralelismo de dados é intrínseca a esta aplicação, pois estamos lidando como distintas RoI-s, que podem ser processadas independentemente sem que haja perda de informações.

A aplicação seria, então, constituída de uma tarefa supervisora que repassaria os dados para unidades de decisão global modulares que operam sobre estes dados, retornando o vetor de probabilidades destes dados. As unidades de decisão global continuariam a se comportar como a rede descrita na seção 4.1.1. O processo supervisor deve carregar os dados e repassá-los de forma bem determinada às aplicações e, ao mesmo tempo, recolher as saídas. A figura 4.4 mostra um diagrama relacional entre as tarefas da aplicação sugerida. Repare que não utilizamos mais de 16 tarefas, pois somente existem 16 nós independentes na máquina-alvo (TN310); nesta e em outras aplicações desejamos que o potencial de cada nó seja totalmente dedicado à execução de uma única tarefa por motivos de eficiência.

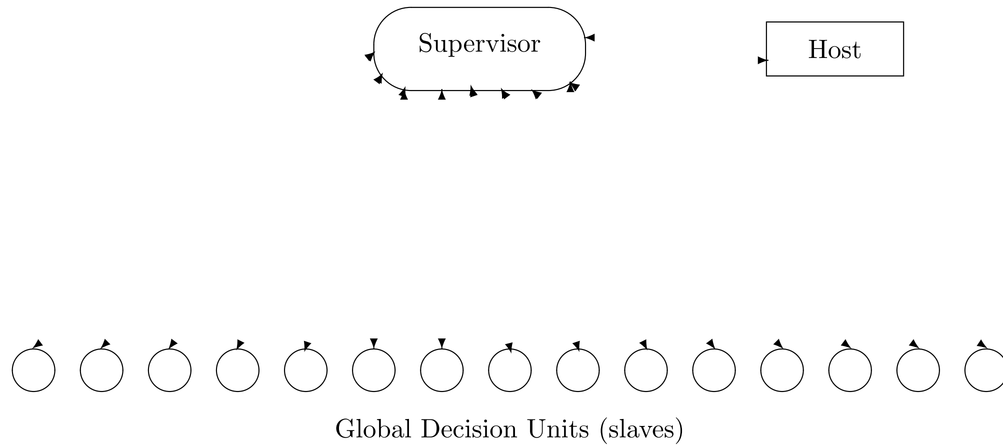
Figura 4.3: Exemplo de *Look-up*.



A função `ativa()` retirada de F.5. Esta função tem como entrada o valor de x (`parâmetro valor`) e o vetor de conversão (passado por ponteiros - `tb`). Retorna o valor convertido.

```
float ativa(float valor, float *tb)
{
 if( valor <= 0 )
 {if ( valor <= -10 ) return -1.0;
  else
  {int indice = 1000*valor+10000;
   return(*(tb+indice));}
 }
 else
 {if ( valor > 10 ) return 1.0;
  else return(*(tb+indice));}
}/* Função ativa */
```

Figura 4.4: Um esquema da GDU rodando em paralelo, com 16 nós de processamento.



O processo supervisor

O processo supervisor é incumbido de várias tarefas, inclusive a de administrar o fluxo de dados e contar o tempo total de aplicação para que seja feita uma estimativa do *speed-up* final. Um fluxograma para o processo supervisor pode ser visto na figura 4.5.

Existem 2 atividades que podem melhor caracterizar o papel do processo supervisor, a primeira é a atividade de distribuição/recolhimento de dados, a segunda é a inicialização dos processos escravos, no nosso caso específico as unidades de decisão global.

Distribuição de dados. Os dados (vetores com 12 características de alguma RoI disparada pelo 1º nível) podem ser passados para as unidades de decisão global de 2 formas distintas:

1. Uma vez inicializado[2] o sistema, esperamos uma espécie de *hand-shake* de alguma unidade de decisão global (indicando que está livre para receber dados), de forma que somente repassamos dados *on-demand*; ou

2. Uma vez inicializado o sistema distribuímos os dados sequencialmente para a primeira unidade, depois para a segunda e assim sucessivamente, ignorando se as unidades estão ou não-livres.

A princípio, a primeira forma parece ser a mais eficiente, pois estaremos lidando com nós de processamento distintos, que podem demorar mais ou menos para finalizar suas atividades (embora idênticas). Para que façamos uso deste método, além da inicialização da aplicação, devemos inicializar o funcionamento dos escravos (cada um com um dado), pois a princípio todos estão livres. Depois desta inicialização, a aplicação supervisora trabalharia sob-demanda, entregando dados novos e recolhendo dados antigos para aplicações que acenassem indicando terem consumido os dados anteriores. Um diagrama esquemático (figura 4.6) mostra o processo de trabalho sob-demanda executável por uma tarefa supervisora genérica.

Embora pareça mais flexível, a primeira forma de implementação não leva em consideração alguns fatores como o tempo de execução do algoritmo da GDU e nem o tempo gasto para a distribuição de dados. De fato, testes realizados (explanados mais a diante) nos mostraram que o tempo gasto no processamento pela unidade de decisões globais (*slave*) era muito inferior ao tempo gasto na distribuição dos dados. Assim, ao distribuir o vetor de características para todas as aplicações pela primeira vez (inicialização de operação) já teríamos a primeira das unidades de decisão global livre; sendo assim,

[2]Isto implica o carregamento, pelas GDU-s, dos pesos da rede e da tabela de conversão para a função de ativação.

Figura 4.5: O fluxo de atividades do processo supervisor.

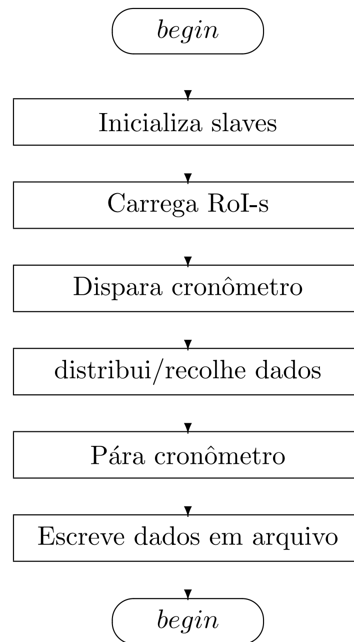


Figura 4.6: Um supervisor trabalhando sob-demanda em uma configuração *master-slave*.

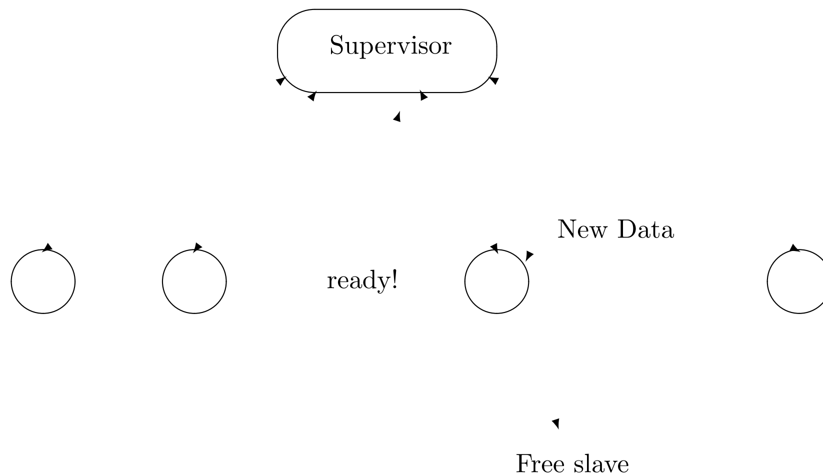
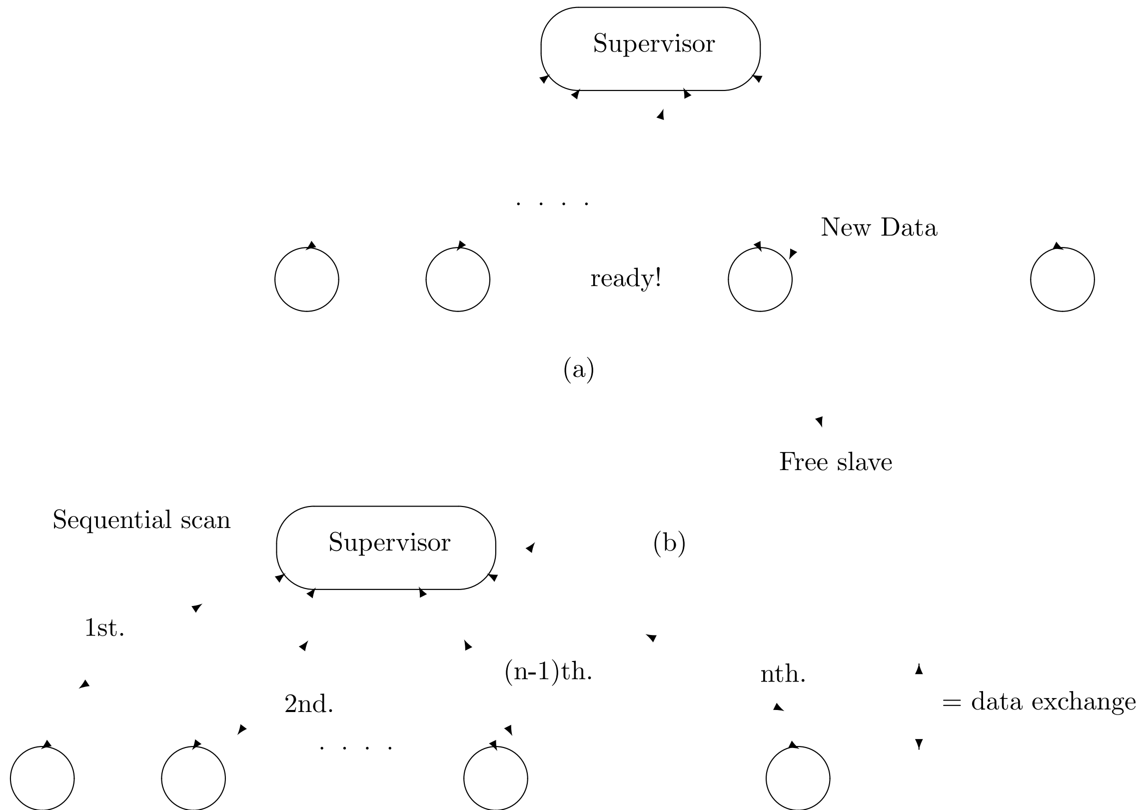


Figura 4.7: A comunicação entre o supervisor e uma GDU, com (a) e sem (b) *hand-shake*.



por que esperar um *hand-shake*? O método mais otimizado seria o de ir distribuindo e recolhendo os dados como com uma “metralhadora”, de forma circular (*Round Robin*), da primeira à ultima unidade, ininterruptamente e sem levantar questões sobre a disponibilidade da tarefa escrava. A otimização chegaria aqui sob a forma de redução do código (nesse caso a checagem de disponibilidade).

É claro que esta abordagem, embora reduza o tempo final de execuçao, não é das mais flexíveis num sentido mais amplo da palavra. Imaginemos o caso em que um dos nós pare de funcionar. Usando a primeira abordagem, aconteceria que apenas perderíamos um dado, e o processo continuaria a funcionar mesmo com o nó defeituoso. Já com a segunda abordagem, se um dos nós parasse de operar, o processo supervisor seria travado, pois estaria, para sempre, esperando uma saída do nó defeituoso[3]. Como estamos usando um sistema onde erros desta natureza ocorrem em escala reduzidíssima, optamos pela segunda abordagem.

Channing In and Out. A comunicação entre as atividades é realizada utilizando-se as funções da tabela 3.2. O fluxograma para a troca de dados com e sem *hand-shake* entre o supervisor e escravos pode ser vista na figura 4.7.

O código comentado do processo supervisor pode ser encontrado no apêndice E. Os resultados, no capítulo 5.

[3]Lembrar que a comunicação com canais em InMOS C é bloqueante às duas tarefas, emissora e receptora de dados.

4.2 Implementação do segundo nível de validação

Uma vez tendo implementado com êxito o sistema de decisão global, estamos aptos a trabalhar no segundo nível de validação como um todo. O segundo nível é composto de várias subtarefas e aplicações, que se interligam formando uma complexa rede de protocolos e dados. Como elucidado na seção 1.3.4, o segundo nível é composto de:

1. Pré-processamento;
2. RoI *Collection*;
3. Extração de Características;
4. Decisão Global em 2 subfases:
   (a) Identificação da partícula excitadora da RoI; e
   (b) Identificação do canal físico de um evento.

Estes se somam a um supervisor formando o sistema de validação. Nossa disponibilidade de equipamento é uma máquina com 16 nós baseados no padrão HTRAM, ou seja, um conjunto de processadores potentes. Dentre as três implementações possíveis para o segundo nível, a arquitetura B parece ser a mais interessante do ponto de vista do equipamento. A arquitetura A não é implementável na TN310 pois exige processamento diferenciado para os extratores de característica, com equipamento altamente granulado[4] a arquitetura C é baseada em chaves muito rápidas (tecnologia ATM), que também não se encontra em nosso poder. A arquitetura B utiliza processadores com alto poder de processamento para realizar um misto entre aplicações de paralelismo de fluxo e paralelismo de dados na execução da tarefa do 2ºnível.

A aplicação do paralelismo de dados vem do fato de aproveitarmos uma grande quantidade de nós de processamento para replicarmos módulos de algumas tarefas, como a dos extratores de característica e das unidades de decisão global. Quanto ao paralelismo de fluxo, ele é aplicado quando se percebe que o processamento de um evento é, em verdade, uma seqüência repetida de processos, i.e.:

1. Extraem-se as características de cada RoI;
2. Decide-se sobre a partícula excitadora;
3. Decide-se sobre o canal físico representativo de um conjunto de RoI-s.

Isto lembra uma linha de montagem, e poderemos então aproveitar esta característica para paralelizarmos também a aplicação em relação ao seu fluxo.

Nossa implementação também parte do princípio que os dados disponíveis já se encontram pré-processados e coletados, estando prontos para serem abastecidos aos Extratores de Característica. Optamos por esta abordagem pelo fato de a demanda da arquitetura B apontar para o uso de rápidos *pipelines* operando na taxa de dados do segundo nível para a execução das unidades de pré-processamento e coleção de RoI-s. Ainda, restringiu-se o funcionamento da aplicação até a identificação da RoI. Posterior identificação do canal físico exigiria treinamento e implementação de novos padrões de rede, tendo em vista vários complicadores como o fato de eventos distintos possuírem diferentes números de RoI-s. A implementação da estrutura como um todo poderia, neste caso, ser reaproveitada visto que tal fase se mistura à própria identificação da RoI.

A figura 2.5 é repetida aqui mostrando quais partes do 2ºnível de *trigger* serão implementadas na TN310 (figura 4.8).

[4] A granularidade é um conceito inerente à sistemas distribuídos: Um sistema é muito granulado (*fine grained*) se possui muitos nós de processamento baseados em processadores de baixa "performance" e pouco granulado (*coarse grained*) se possui poucos nós baseados em processadores de alta "performance" (caso da TN310).

Figura 4.8: As partes a serem implementadas na TN310 do segundo nível de *trigger* do experimento ATLAS/LHC.

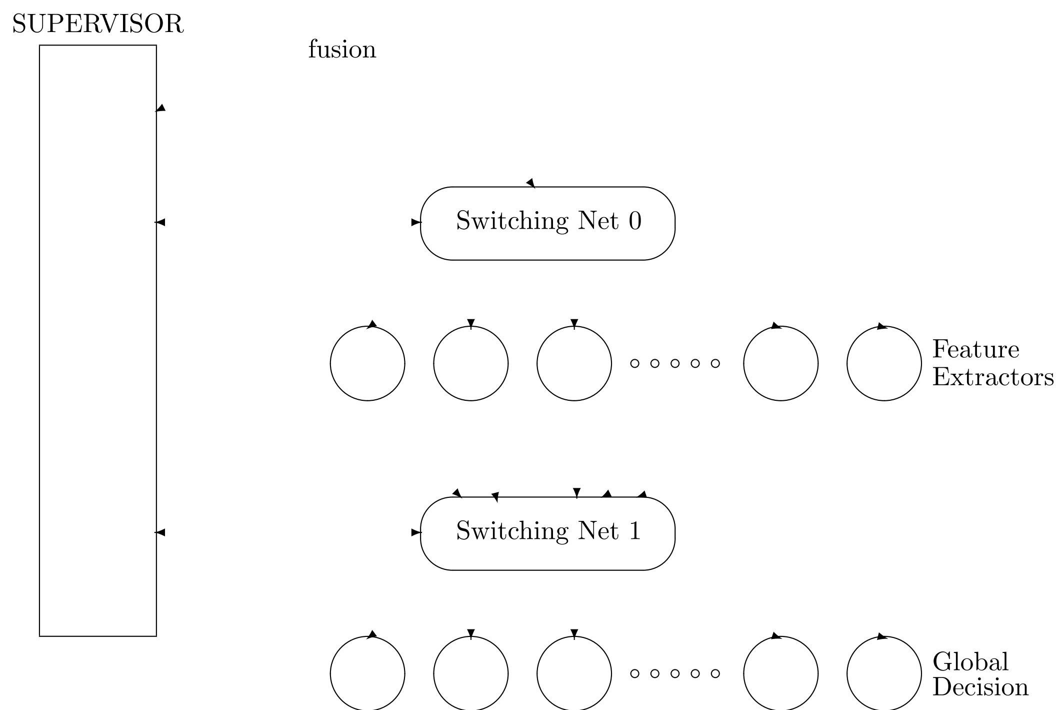
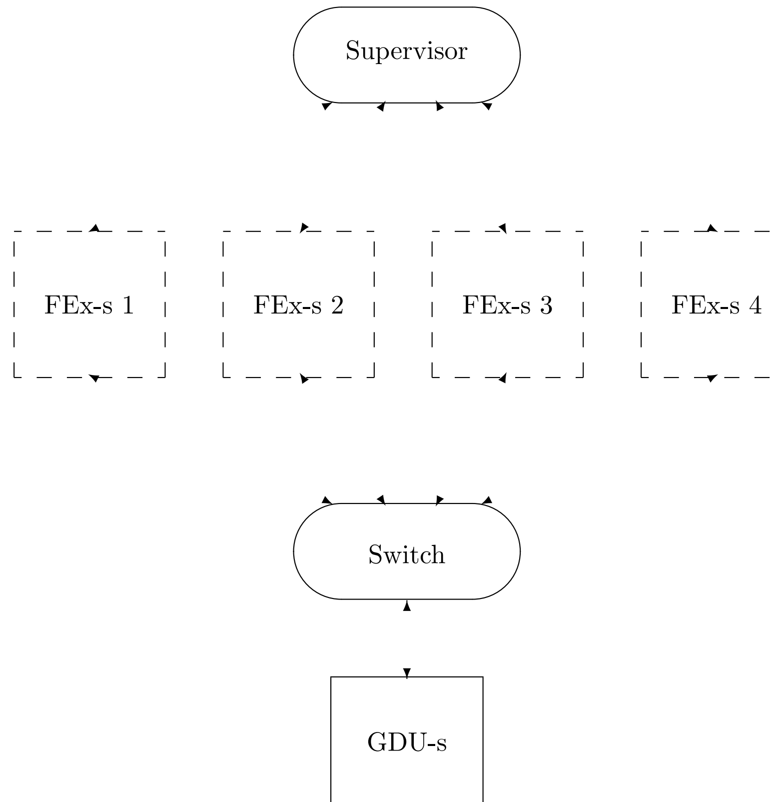


Figura 4.9: Diagrama da implementação genérica do sistema de validação da figura 4.8 no sistema TN310.



Os extratores de característica foram substituídos por um bloco de processamento que consome o tempo de aplicação, mas que, em verdade nada processa. Esta decisão foi tomada pois extratores de característica não foram o alvo principal deste trabalho, embora possam ser encontrados em muitas referências de nosso grupo (Colaboração Internacional CERN/COPPE/UFRJ). Existem vários laboratórios trabalhando na otimização de extratores para diversos subsistemas. A utilização de extratores específicos retiraria a flexibilidade de uma aplicação cuja meta principal seria verificar o funcionamento do equipamento quando exposto a um quadro de impacto como o do sistema de validação. Tendo estes fatores em vista, decidimos por esta substituição.

Um diagrama topológico (e genérico) da implementação da aplicação no sistema TN310 pode ser visto na figura 4.9. Os blocos pontilhados representam processos que foram substituídos pelo tempo de processamento equivalente.

Dados. Uma vez que estaríamos utilizando enfoques que visam a detetar tempos de processamento e "gargalos" durante a execução da aplicação, os dados que estão sendo passados para os extratores de característica são dados pseudoaleatórios, assim como suas saídas. Dados reais somente estarão sendo utilizados nas unidades de decisão global, visto que aproveitaremos a unidade modular previamente construída para preencher tal espaço. A devida substituição dos dados aleatórios por dados válidos para a unidade de decisão global será feita no momento oportuno como será visto e não representa peso significativo no tempo de processamento global.

4.2.1 Simulando o segundo nível

Para simularmos o segundo nível de validação no menor tempo possível alguns aspectos devem ser levados em conta:

- Estamos interessados na máxima otimização das rotinas a serem executadas;
- Interessa-nos, também, que haja uma minimização na comunicação entre as tarefas, pois isto introduz uma seqüencialização da atividade;
- A divisão das tarefas e processadores deve ser feita de forma a balancear ao máximo a carga sobre as aplicações, de tal forma que os dados fluam sem que haja muitos "gargalos".

A máxima otimização do tempo de execução das rotinas pode ser observada segundo algumas regras de programação e alocação do código nos nós de processamento, como é possível ver na seção *Performance Tips* em [15].

A minimização do código do processo de comunicação entre as atividades se resume basicamente a remover processos de comunicação desnecessários, como confirmações de recebimento de dados e *hand-shakes*[5]. Uma vez que estamos lidando com um *pipeline*, fica difícil reduzir a taxa de comunicação com relação aos dados-alvo do *pipe*, no nosso caso as RoI-s.

A divisão das tarefas visa a balancear o sistema de tal forma que partes que executem suas atividades rapidamente fiquem pouco tempo paradas esperando a execução de outras partes da aplicação. Devemos, entretanto, nos lembrar de que a alocação de pelo menos 2 processadores para cada atividade deva ser feita de forma a caracterizar a utilização do paralelismo de dados nas diversas subtarefas do sistema de validação.

Uma vez que o processo de otimização acontece passo a passo, exporemos aqui as implementações e otimizações feitas no decorrer do tempo enfatizando a utilização de novos elementos e conceitos ao longo das diferentes versões do simulador.

Simulador - Versão 1

Nesta versão utilizamos o conhecimento adquirido projetando-se a aplicação para a unidade de decisões globais para chegarmos a um projeto final do sistema. É claro que levamos em consideração todos os aspectos citados na seção 4.2.1.

Uma configuração que atende a todos os pré-requisitos expostos até o presente é exibida no diagrama da figura 4.10. Neste diagrama identifica-se uma tarefa supervisora, desempenhando papel semalhante ao da tarefa supervisora na diagramação das unidades de decisão global. Identificamos também 2 nós de processamento responsáveis pela extração de característica para calorímetros, 3 para os extratores de TRT, 3 para SCT e 3 extratores para os dados da câmara de múons. Esta separação se deve a fatores de balanceamento. Em seguida observam-se, ao invés de uma, 2 aplicações fazendo o papel de *switching network* chamadas de Local Network 0 e Local Network 1 (LN0 e LN1). Ao final, as unidades de decisão global.

O Fluxo de Dados. O fluxo de dados é facilmente entendido e acompanha os seguintes passos:

1. As RoI-s pré-processadas e coletadas estão à disposição da aplicaçao supervisora (lembremos que os dados são pseudo-aleatórios e sem maior importância em seu conteúdo, mas de vital importância em seu tamanho);
2. Estas são repassadas para os extratores de característica de acordo com o subsistema a que pertencem, i.e., dados de calorímetros vão para os extratores de calorímetros e, assim, sucessivamente;

[5] Lembre-se de que a comunicação por canais é um processo bloqueante e que esta fato remove a necessidade de *forced acknolegments* e *hand-shakes*.

Figura 4.10: O diagrama da aplicação a ser simulada na TN310.

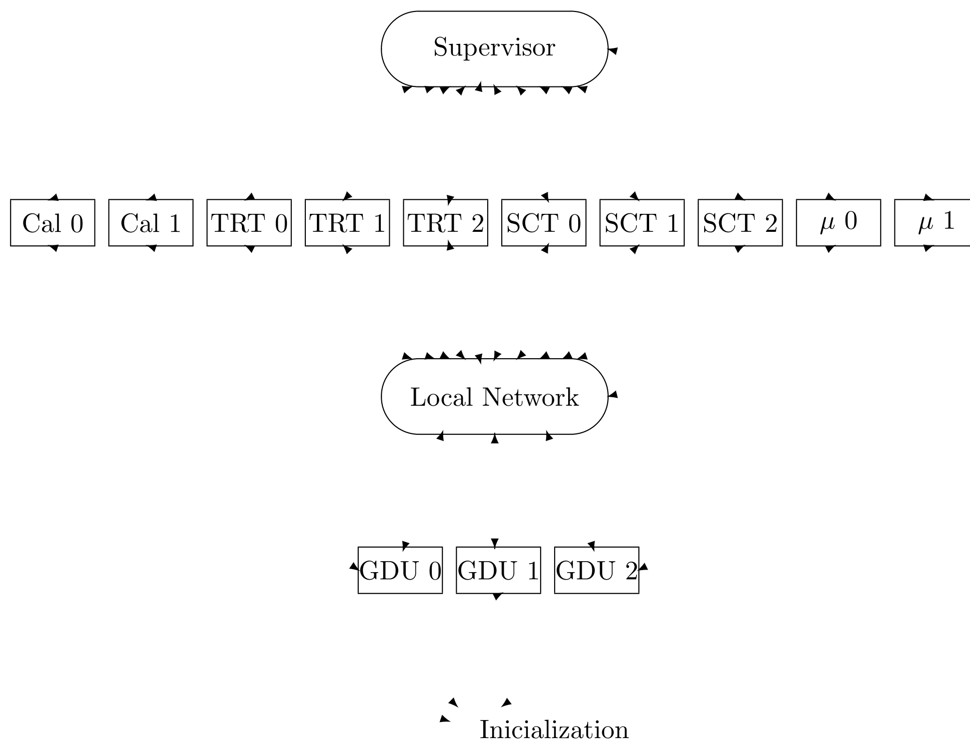
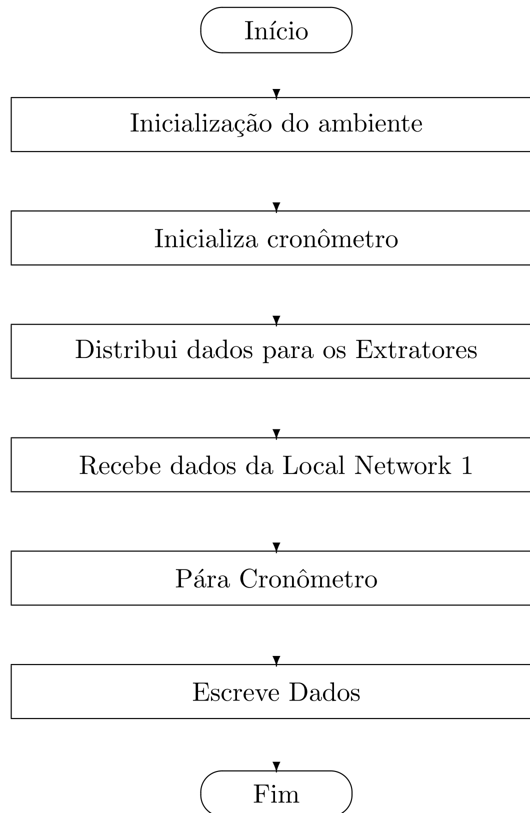


Figura 4.11: Fluxograma do processo supervisor para a simulação do segundo nível de *trigger*.



3. Os dados processados são recolhidos pela LN0, que aguarda até que todos os dados relativos a uma RoI cheguem; quando isto acontece, as características extraídas são repassadas a LN1;

4. A LN1 controla a interação com as unidades de decisão global, i.e., repassa o vetor de características (substituído por dados válidos) e recolhe o vetor de probabilidades, repassando-o ao supervisor por ocasião do final de processamento.

Este processo é repetido no sistema distribuído 500 vezes. Achamos que este número representava uma quantidade de RoI-s suficiente para que pudéssemos estimar o tempo de processamento para uma única RoI.

Para que não nos estendamos demasiadamente na explanação de versões de programas que não representem a versão final, elucidaremos apenas alguns pontos de cada uma destas versões que as caracterizam, alocando, no final do documento, um apêndice onde é possível encontrar a versão final da aplicação comentada.

Processo Supervisor - detalhes de implementação. O processo supervisor da primeira aplicação foi construído baseado no fluxograma da figura 4.11.

A primeira fase do programa consiste em inicializar todo o sistema. Esta fase é responsável por carregar as unidades de decisão global com pesos, vetores de normalizaçào e a tabela para a conversão da tangente hiperbólica, e carregar a LN1 com dados válidos para as unidades de decisão global; o código destas fases segue no escopo da figura 4.12.

Em seguida, a contagem de tempo é disparada; dois contadores são utilizados, 1 para o tempo global e 1 para o tempo de distribuição dos dados. Este segundo contador tem por objetivo determinar qual

Figura 4.12: As instruções-chave que implementam a inicialização do sistema pelo supervisor.

```
...
/* Criação dos canais */
int InputSize = (int) *((long int *) get_param(4));
int OutputSize = (int) *((long int *) get_param(6));
Channel **Input = CreateChannels(get_param(3), InputSize);
Channel **Output = CreateChannels(get_param(5), OutputSize);
Channel *from_ln1 = get_param(7);
Channel *to_ln1 = get_param(8);
int gdInSize = (int) *((long int *) get_param(10));
int gdOutSize = (int) *((long int *) get_param(12));
Channel **to_gd = CreateChannels(get_param(11), gdOutSize);
/* End Channel creation */
...
/* Envia a matriz que contem os vet_ROI's para LN1 */
ChanOut(to_ln1,matriz,sizeof(matriz));
printf(" Sent true global data to Local Network 1.\n");
...
/* Feeding Global Decision Workers */
printf("Loading Global Decision Units...\n");
printf(" -- Particle identification ONLY\n");
/* Lê tabela com valores das tangentes hiperbólicas */
leia_tabela(tab);
/* Lê arquivos de pesos */
leia_dumps(hidd,outlay);
/* Lê o arquivo com os valores de normalização */
leia_norma(normal);
for(i=0;i<gdOutSize;i++)
{
 ChanOut(to_gd[i],tab,sizeof(tab)); /* Tabela com tanh */
 ChanOut(to_gd[i],normal,sizeof(normal)); /* Vetor de normalizacao */
 ChanOut(to_gd[i],hidd,sizeof(hidd)); /* Neuronios da camada escondida */
 ChanOut(to_gd[i],outlay,sizeof(outlay)); /* Neuronios da camada de saida */
 printf("GD Worker #%d, LOADED.\n",i);
}/* end for(i) */
/* Fim do envio de dados para Global Decision Network */
...
```

Tabela 4.2: Os dados enviados a cada vez para os extratores de característica. Valores em bytes.

	Calorímetro	TRT	SCT	Múon
Dados	484 (121 floats)	400 (100 floats)	400 (100 floats)	40 (10 floats)
Cabeçalho	20 (4 integers)	20 (4 integers)	20 (4 integers)	20 (4 integers)
Total (bytes)	504	420	420	60

Tabela 4.3: O cabeçalho enviado junto com cada RoI. Valores em bytes.

	Posição no vetor
# Evento	1
final de processo	2
# RoI	3
# RoI-s no evento	4
registrador	5

é o tempo que é gasto para que passemos os dados para os extratores de característica. Veja a seguir o código.

```
...
printf("Processing, wait till end...\n");
inicio_global = ProcTime();
...
/* Conta tempo de distribuição para cada RoI/subsistema de deteção */
inicio = ProcTime();
...
```

Inicia-se então a distribuição de eventos pelos extratores de característica. Uma vez que estamos utilizando versões simuladas destas aplicações, estaremos passando dados pseudo-aleatórios, i.e., dados que não têm uma significação direta mas que ocupam, em tamanho, valores próximos aos valores de tamanhos reais. Desta forma simularemos com precisão o tempo gasto na comunicação entre o processo supervisor e os extratores de característica. A tabela 4.2 mostra o tamanho dos vetores repassados aos extratores de característica.

A tabela também mostra a inclusão de um cabeçalho. Este cabeçalho é utilizado para que, em qualquer momento ou em qualquer aplicação, conheça-se a procedência de uma dada RoI. O cabeçalho (também com dados fictícios) contém informações como o número do evento, número da RoI, número de RoI-s neste evento e mais 2 campos extras que são usados em algumas partes do programa como indicadores de final de operação ou registradores. A organização do cabeçalho enviado com cada RoI pode ser visto na tabela 4.3.

As rotinas de envio podem ser vistas no escopo da figura 4.13. Nestas rotinas, reparamos a implementação de instruções para que o supervisor trabalhe enviando dados *on-demand* (instrução `ProcAltList(Input);`). Esta instrução recebe como parâmetro de entrada um vetor montado pela rotina `CreateChannels()` e monitora estes canais esperando que algum deseje se comunicar; quando isto acontece, ele repassa à variável `i` a posição do canal, dentro do vetor querendo se comunicar. Por exemplo, se o canal `Input(0)` quiser se comunicar, a função retornará 0 (zero). E, assim, sucessivamente.

Dois problemas podem ocorrer com esta implementação; os dois são contornáveis ainda que demandem acréscimo de código, como foi feito. Inicialmente, exporemos o quadro de conexões da aplicação.

A aplicação supervisora está conectada às aplicões extratoras de características através de 2 dezenas de canais (são 10 extratores, lembre-se). Dez destes canais são apontados para os extratores, logo constituindo canais de saída; os outros 10 são os equivalentes de entrada de dados vindos dos extratores. Estes 2 jogos de canais são aqui chamados de `Input` e `Output` por motivos óbvios. A princípio o fluxo de dados somente ocorre do supervisor para os extratores, logo deixando de haver necessidade do vetor de canais `Output`. Os canais, também a princípio, são conectados de forma estática aos extratores; assim, determinamos que o canal `Input[0]` estaria conectado ao extrator de calorímetro zero, o `Input[1]` ao extrator de calorímetro 1 e, assim, sucessivamente. Desta forma, se quisermos nos comunicar com um extrator, basta que coloquemos o dado no canal daquele extrator.

O primeiro problema da utilização da função `ProcAltList` é que ela espera que a tarefa que estiver livre chame a aplicação supervisora através de um canal. Isto normalmente não deveria acontecer nesta aplicação, pois o fluxo de dados, como explanado, ocorre do supervisor para os extratores, e jamais ao contrário. A utilização de tal função nos força a implementar um canal de comunicações que vá dos extratores para o supervisor, indicando a liberação. Este passo consome mais tempo de processamento, mas deve ser implementado.

O segundo problema da utilização da função `ProcAltList` concerne ao tempo de execução. Existe, como no caso das unidades de decisão global, a necessidade de inicializarmos a operação dos extratores, distribuindo um dado para cada um, pois todos estão livres no início. Se, ao distribuírmos o dado para o último processador, o primeiro já estiver livre, a função `ProcAltList` o identificará e passará novo dado para este extrator. Assim prosseguirá até o final dos dados. Porém, se a aplicação for "viciada" (o que acontece na maioria das vezes) e os primeiros extratores processarem mais rápido do que a função pode vasculhar os canais, os primeiros extratores, i.e., aqueles que estão conectados aos primeiros canais serão sempre beneficiados com novos dados pois a função em questão não possui um código seletivo. Em outras palavras, depois que a função escolhe um canal para comunicação ela "reseta" e começa a vasculhar do primeiro canal; se este estiver livre, ela o escolhe. Em aplicações como esta, o tempo de processamento é menor que o tempo gasto para enviar dados; logo, a utilização de `ProcAltList`, beneficiará somente os primeiros canais.

Para que isto não aconteça, é necessário que façamos uma rotação na posição dos canais dentro do vetor. É isto que a função `RotateChannels` faz. Quando é chamada, a função muda a posição dos canais no vetor. Desta forma, quando o vetor for vasculhado pela função `ProcAltList`, estaremos priorizando vetores que não foram ainda escolhidos.

Um problema direto que surge com isto é que as posições antes fixas para os extratores (segundo os canais a que estavam conectados) não serão mais válidas em detrimento da rotação, e teremos que adicionar uma lógica para que saibamos com quem estamos falando em cada momento da aplicação, como é visto na figura 4.13.

A figura (na realidade é um texto) também mostra o uso da função `send_data()`; esta função manda os dados para os extratores e controla de forma simplificada o fluxo de dados na aplicação. Foram omitidos aqui trechos redundantes do programa e rotinas de checagem de operação. Elas estarão inclusas em um apêndice no final do trabalho.

Esperando os dados da LN1. Depois de enviados todos os dados para os extratores, a aplicação supervisora espera até que a tarefa responsável por colher os dados das unidades de decisão global (LN1) retorne os resultados. Isto é feito numa imensa matriz (depois de todos os dados terem sido processados), pois desta forma otimizaremos a utilização do canal. A seguir, o trecho do programa dedicado a esta tarefa.

```
ChanIn(from_ln1,matriz_prob,sizeof(matriz_prob));
```

A contagem de tempo é parada.

```
final=ProcTime(); /* End final time */
acc=ProcTimeMinus(final,inicio_global); /* Evaluates time spent */
```

Figura 4.13: As rotinas de envio de dados para os extratores.

```
...
/* A próxima instrução implementa a procura de um escravo desocupado
para descarregar dados */
i=ProcAltList(Input);  /* Checks for ready FEx'or */
...
/* Traduz o canal que respondeu em um tipo de Ext.Car., para que não envie
para o Ext.Car. errado. */
if(fex_num==0)
{
 tipo=i; /* se o primeiro canal for o do 1o.
             FEx'or, então não há o que fazer !!*/
}
else /* Faz algoritmo para
        descobrir qual é o tipo do FEx'or */
{
 /* algoritmo omitido */
}
/* End seta tipo */
...
/* dependendo do tipo de Ext.Car. manda um tipo de dado */
switch(tipo)

case 0:
case 1:
 if(hold_calo < NROIS_MAX)
 {
  send_data(Input[i],Output[i],calo_info,
                     &hold_calo,&fim,vetor_calo,121);
 }
 break;
...
RotateChannels(Input,InputSize,1);
        /* Rotaciona canais, para nao ficar em cima de um so */
RotateChannels(Output,OutputSize,1);
          /* Rotaciona out chan's para nao ficar diferente */
fex_num--;
if(fex_num<0) { fex_num=OutputSize-1; }
              /* Decrementa apontador ja que houve rotacao */
...
```

Tabela 4.4: Tempos de processamento simulados para cada tipo de extrator.

Extrator	Tempo
Calorímetro	$10\mu s$
TRT	$400\mu s$
SCT (PreShower)	$250\mu s$
Múon	$5\mu s$

Tabela 4.5: Os dados externalizados pelos extratores de característica para LN0 (em bytes).

Extrator	Dados	Cabeçalho	Total
Calorímetro	24 (6 floats)	20 (5 integers)	44
TRT	8 (2 floats)	20 (5 integers)	28
SCT (PreShower)	12 (3 floats)	20 (5 integers)	32
Múon	4 (1 float)	20 (5 integers)	24

A partir daí, a aplicação escreve os diversos dados recolhidos durante a sua operação e finaliza.

Os extratores de característica. Os extratores de característica são processos mais simples, pois constituem simuladores. Estas tarefas recebem o vetor de dados da aplicação supervisora, esperam um tempo pré-determinado (tabela 4.4) e repassam um vetor cujo tamanho (tabela 4.5) equivale ao das características extraídas por algoritmos reais. Repare que o volume de dados é bem menor. Isto mostra a eficiência dos algoritmos de extração na transformação de todos os dados de uma RoI em poucas variáveis inteligentes.

As redes locais. Estas redes fazem o papel da chave no esquema da figura 2.5. A primeira parte recebe os dados dos extratores de característica e os armazena em uma matriz de forma ordenada, segundo a organização exigida pela unidade de decisão global, ou seja, 6 características de calorímetro, 2 de TRT, 3 de SCT e 1 de Múon. Esta matriz é bidimensional, onde as linhas representam o evento e as colunas, as RoI-s. Cada posição é uma estrutura de dados contendo o cabeçalho e as características já extraídas para aquela RoI.

```
struct _roi_ /* Estruturas para a matriz de dados */
{
 float ext_feat[NFEAT];
 int header[HEADER_SIZE];
};
```

A LN0 deve, a cada vez que recebe uma nova entrada, verificar se todas as características para aquela RoI já não chegaram. Caso tenham chegado todas as características, então estes dados já podem ser repassados para o próximo estágio, a LN1; caso não, elas são armazenadas, aguardando a parte restante das características.

A recepção de dados é feita sob-demanda operando de forma parecida com a implementação feita na aplicação supervisora (usando `ProcAltList`). A única diferença é que o fluxo de dados, aqui, não justifica a utilização de *hand-shake*. A função `recv_data()` é responsável pelo controle do fluxo de dados no programa, ajustando-se ao dado de entrada. Ela o posiciona corretamente na matriz de dados, verificando a natureza de seu cabeçalho informativo.

```
while(final!=4)
```

```
 {
  /* Processador Liberado ?? */
  i=ProcAltList(Input);  /* Checks for ready FEx'or */
  /* Descobre o tipo de FEx'or que requisitou dados */
  if(fex_num==0)
  {
tipo=i; /* se o primeiro canal for o do 1o. FEx'or, então não há o que fazer !!*/
  }
  else /* Descobre qual é o tipo do FEx'or */
  {
   /* omitido */
  }
  /* End seta tipo */

/* Somente precisamos testar processador pois no caso de RoI's analizadas
 por um tipo de rede acabarem, o processador responsável não irá mais acenar
 com ChanOut(); */

switch(tipo)
{
 case 0:
 case 1: recv_data(Input[i], vet_info, matriz, &final, 6, 0); /* CAL */
         break;
 case 2:
 case 3:
 case 4: recv_data(Input[i], vet_info, matriz, &final, 2, 6); /* TRT */
         break;
...
    /* Este é um exemplo de utilização do 5o. elemento do
       header (registrador) se header[4] == 4, isto significa
                    que já recebeu os dados dos 4 extratores */
    if(matriz[vet_info[0]][vet_info[1]].header[4]==4)
    {
     ChanOut(to_ln1,vet_info,sizeof(vet_info));
     ChanOut(to_ln1,&matriz[vet_info[0]][vet_info[1]],sizeof(struct _roi_));
    }/* Dados mandados para LN1 */
```

Como é possível ver, também são utilizados aqui os procedimentos de rotação de canais e "setagem" do tipo de extrator que acena indicando estar com dados prontos.

A segunda parte da rede local, LN1 (*Local Network one*), recebe as características das RoI-s sob a forma de estruturas (cada uma com 12 `floats` contendo as características e 5 inteiros com o cabeçalho) e escolhe uma unidade de decisão global para onde passar estes dados. A escolha aqui também é feita sob-demanda.

Para que pudéssemos utilizar a unidade de decisão global modelada na seção 4.1.2 sem alterações e aproveitando sua modularidade, elaboramos um sistema que, ao passar as características para uma determinada unidade de decisão global, guarda o cabeçalho relativo àquela RoI. Desta forma, somente remetemos as características, e não correremos o risco de cofundir as RoI-s. Partes relevantes do código encontram-se a seguir:

```
...
 /* Input from LN0, first the header, then, the features */
 ChanIn(from_ln0,vet_info,sizeof(vet_info));
 ChanIn(from_ln0,&matriz_dados[vet_info[0]][vet_info[1]],sizeof(struct _roi_));
```

```
...
 /* Changes received data for valid ones */
 for(i=0;i<=11;i++)
 {
  matriz_dados[vet_info[0]][vet_info[1]].ext_feat[i]=matriz_real[linha][i];
 }
...
  /* Any processor freed ?? */
  i=ProcAltList(Input);  /* Checks for ready GD Workers */

  ChanIn(Input[i],&matriz_prob[gd[i][0]][gd[i][1]].outnet[0],
                               4*sizeof(float)); /* Receive data */
  ChanOut(Output[i],&matriz_dados[vet_info[0]][vet_info[1]].ext_feat[0],
                                   12*sizeof(float)); /* Pass new data */
  /* Salva header */

  for(k=0;k<HEADER_SIZE;k++)
  {
matriz_prob[gd[i][0]][gd[i][1]].header[k]=gd[i][k];
                              /* Saves header into right position */
gd[i][k]=vet_info[k];
  }
  /* Does not have to rotate since if an application
     is free it can be used, even though it's the first (or the second)
     one always */
```

A unidade de decisões globais. A unidade de decisões globais é a mesma modelada na seção 4.1.2 e não será adicionado nenhum comentário extra aqui.

Concluindo sobre o modelo. Este modelo, totalmente operacional, foi o 1ºque tentamos implementar. Ele é altamente robusto, visto que opera 100% sob-demanda. Se não for solicitado, não opera. Isto significa que, se partes do sistema falharem, ele não interrompe suas rotinas ou fica travado.

Durante as medidas de "performance" pudemos perceber 2 fatores que seriam responsáveis por uma queda de desempenho: o primeiro é relacionado a extratores cujos dados acabaram. Nestes casos, embora não haja mais dados para serem tratados, eles continuam acenando indicando estarem livres para processarem. Esta não é uma boa característica, pois o supervisor perderá tempo até que descubra que os dados daquele tipo de extrator acabaram. Uma possível alteração seria fazer os processadores de características cujos dados acabaram pararem de acenar; assim, economizaríamos este tempo gasto na verificação de processos terminados.

O segundo fator responsável pela redução de "performance" é mais complexo. Como vimos na seção 3.2.4 a conexão entre os diversos nós de processamento é feita de forma distinta. Estas conexões beneficiam a comunicação entre alguns nós e prejudicam a comunicação com outros. A exemplo, vemos processos rodando em nós (0 e 8) que se comunicam através de 4 chaves STC104, ou processos que se comunicam através de apenas 1 destas chaves. Isto sugere que, para termos "performance" máxima, teremos que realocar os processos de forma a otimizar os processos de comunicação entre as tarefas. Desta forma, será possível reduzir o tempo gasto em comunicação.

Simulador - Versão 2

Nesta versão estaremos priorizando a realocação dos processos de forma ordenada, bem como resolvendo a questão de extratores livres cujos dados já acabaram.

Tabela 4.6: A alocação de processos na primeira versão do programa simulador.

Task	Node
Supervisor	0
Calo.FEx 0	1
Calo.FEx 1	2
TRT.FEx 0	3
TRT.FEx 1	4
TRT.FEx 2	5
SCT.FEx 0	6
SCT.FEx 1	7
SCT.FEx 2	8
Muon.FEx 0	9
Muon.FEx 1	10
LN0	11
LN1	12
GDU 0	13
GDU 1	14
GDU 2	15

A tabela 4.6 pode ser elucidativa quanto à alocação dos processsos feita durante a primeira versão do simulador.

Nesta tabela, vemos que a alocação dos processos foi feita sem que nenhum cuidado fosse tomado quanto à distância entre os nós de processamento. Reparemos que, para que o supervisor se comunique com qualquer dos extratores, é preciso que os dados atravessem pelo menos **2** chaves. Já a comunicação entre os extratores e a rede local (LN0) se dá através de conexões de vários tamanhos, ou seja, através de 1, 2 ou 3 chaves, retirando o balanceamento requerido em aplicações deste tipo, como já mencionado.

Por outro lado, temos que as unidades de decisão global estão bem alocadas em relação à rede local (LN1) visto que para que se comuniquem com ela, os dados terão que atravessar somente uma chave.

O diagrama da figura 4.14 pode ser elucidativo quanto à distância entre os processadores. Já a tabela 4.7 apresenta um novo e eficiente posicionamento para as tarefas desta aplicação. A figura 4.15 mostra o diagrama de figura 4.14 com as alterações propostas por esta última tabela. Estas alterações foram feitas baseadas em algumas características do sistema; são elas:

- Tarefas que demandem alto fluxo de dados devem permanecer maximamente próximas;
- Tarefas que executem suas atividades em tempo menor jamais serão beneficiadas com uma alocação mais favorável do que tarefas mais longas, por questões de balanceamento de carga;
- Uma vez que estamos em um sistema trabalhando sob-demanda e existe a possibilidade de extratores ou unidades de decisão global operarem de forma mais rápida que a distribuição de dados (fazendo com que os primeiros de cada categoria sempre sejam selecionados), os primeiros módulos de uma mesma tarefa devem ser beneficiados quanto à sua proximidade das tarefas que enviem ou recebam dados.

Simulador - Versão 3

Nesta versão, resolvemos utilizar a técnica de distribuição circular exposta na seção 4.1.2 para otimizar a distribuição e recolhimento de dados aos extratores de característica e unidades de decisão global.

Figura 4.14: O diagrama mostra a alocação de processos e a "distância" entre estes na primeira versão do programa simulador.

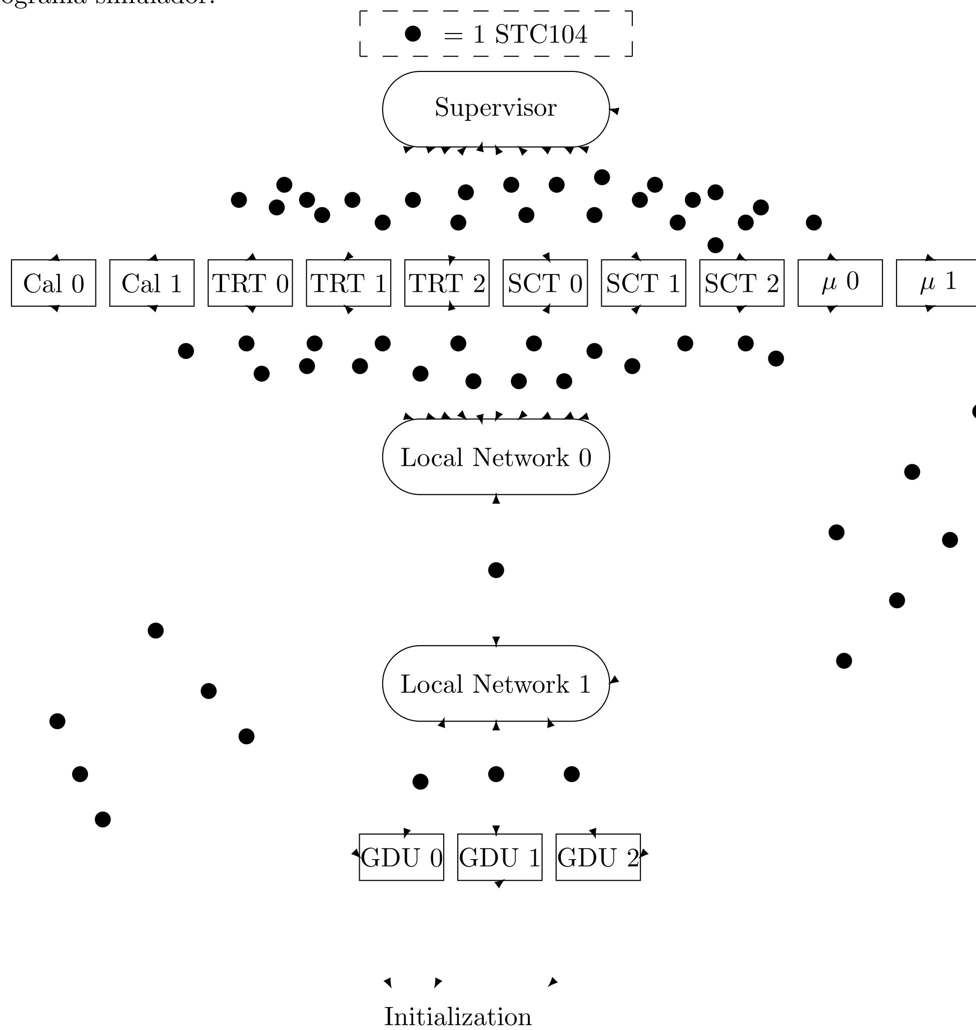


Figura 4.15: O diagrama mostra a alocação de processos e a "distância" entre estes na segunda versão do programa simulador.

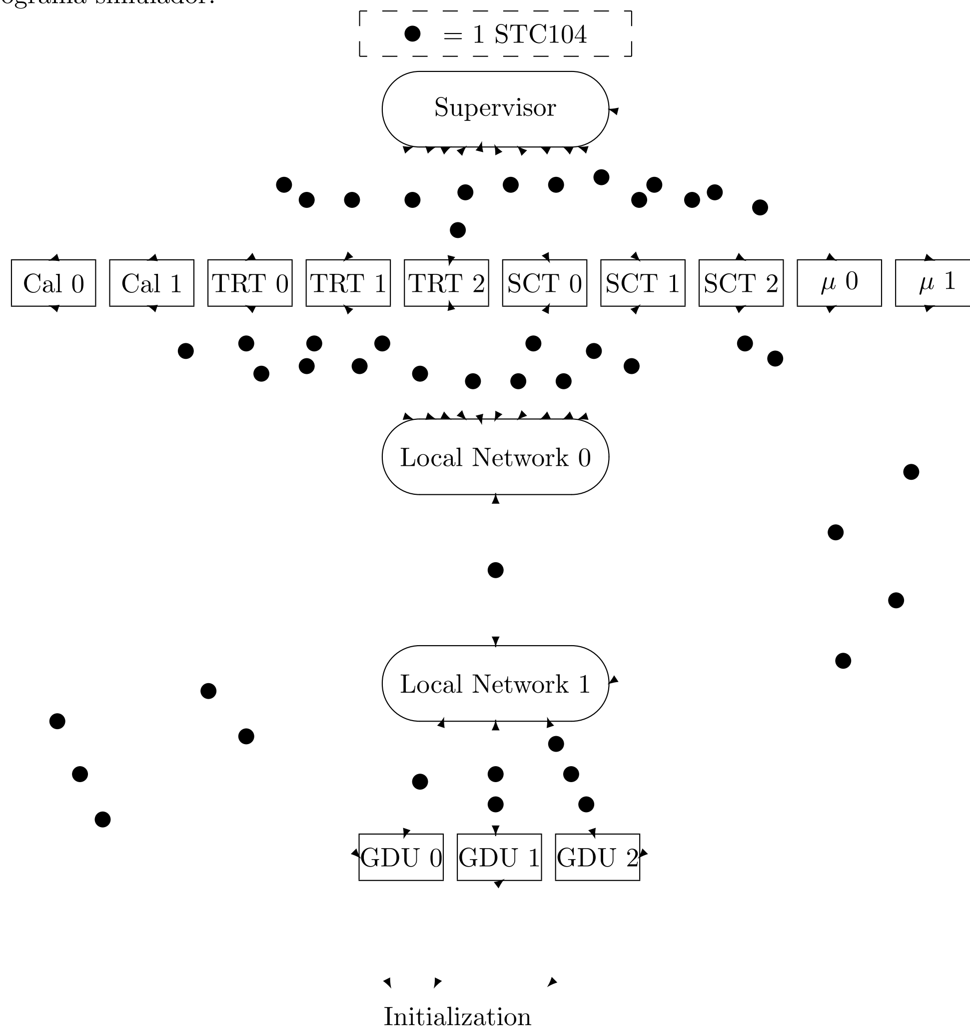


Tabela 4.7: Uma proposição mais eficiente para para a alocação das tarefas na TN310.

Task	Node
Supervisor	1
Calo.FEx 0	4
Calo.FEx 1	7
TRT.FEx 0	2
TRT.FEx 1	3
TRT.FEx 2	9
SCT.FEx 0	5
SCT.FEx 1	6
SCT.FEx 2	10
Muon.FEx 0	11
Muon.FEx 1	12
LN0	13
LN1	14
GDU 0	15
GDU 1	8
GDU 2	0

Esta abordagem, embora não nos leve a um sistema mais robusto, é, sem dúvida, a mais rápida, pois estaremos eliminando algumas funções inerentes a versões anteriores do processo de simulação.

O mapa organizacional das tarefas foi mantido, visto ter sido esta a configuração mais otimizada para a aplicação. Uma vez que esta foi a última versão implementada, resolvemos por anexá-la por completo ao final do documento, no apêndice F. Neste apêndice encontramos o arquivo de configuração (CFS) para a aplicação e os códigos das aplicações para cada tarefa totalmente documentados.

Nesta versão também utilizamos versões otimizadas (`DirectChan-s`) das funções que implementam a comunicação entre tarefas (tabela 3.2).

Capítulo 5

Resultados obtidos e Conclusões

Neste capítulo, estaremos expondo os resultados obtidos com as implementações expostas no capítulo anterior. Para todas as implementações, consideramos o *speed-up* em relação a uma versão simples da aplicação, isto é, rodando em um único processador. Também é feita uma revisão dos tempos de comunicação e afins.

Inicialmente, repassaremos testes realizados sobre o tempo de comunicação do sistema TN310. Estes testes visaram a um entendimento operacional do tempo de processamento dispendido durante comunicações entre aplicações. Na segunda seção, traremos resultados para a aplicação do sistema de decisão global e na seção seguinte, exporemos os resultados e conclusões para a simulação do sistema de validação.

5.1 Teste de comunicações

Em equipamentos com número de nós de processamento considerável é necessário o entendimento pleno do sistema de comunicação utilizado. Quando menciona-se "Entendimento Pleno"quer se acentuar que para uma otimização das rotinas há a necessidade de o programador desenvolver uma "afinidade" muito grande com o equipamento. Esta afinidade traduz-se no entendimento de todos os detalhes de funcionamento do sistema, sejam eles de alto ou de baixo nível.

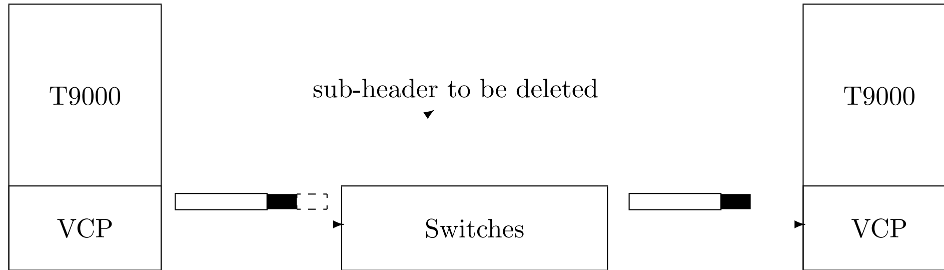
O processo de comunicação entre os diversos nós de processamento no sistema TN310 não é diferente, isto é, ele possui uma interface de alto nível (como as funções da tabela 3.2) e uma interface de baixo nível, inerente ao sistema. A interface de alto nível é responsável pelo processo de comunicação em si, mas os motivos pelos quais alguns tipos de comunicação são diferentes de outros é um ponto que somente pode ser explicado com o entendimento do sistema operando em baixo nível. Desta forma, antes de entrarmos nos resultados destes testes propriamente, abordaremos alguns aspectos na forma de comunicação entre os diversos nós HTRAM do sistema TN310.

5.1.1 Introdução ao sistema de comunicações

O sistema de comunicações da TN310 é totalmente baseado no conceito de chaveamento assíncrono de pacotes (*Asynchronous Packet Switch*). Este tipo de sistema pode aumentar a "performance" em ambientes maiores, como é caso. Ele se baseia na utilização de uma chave rápida (STC104) para que haja a distribuição de pacotes pela rede de forma mais rápida e livre de erros possível. Estas chaves podem rotear até 32 pacotes diferentes ao mesmo tempo, vindo de quaisquer 32 localidades distintas e indo para outras 32 localidades também distintas.

A idéia é, na verdade, bem simples. Quando uma informação deseja ser mandada de um ponto a outro, a tarefa no ponto emissor envia uma mensagem ao processador virtual de canais local indicando o endereço e o tamanho do dado a ser mandado. O processador de canais divide o dado a ser enviado

Figura 5.1: O processo de deleção de cabeçalho em uma rede com chaveamento assíncrono de pacotes.



em vários pacotes de tamanho predeterminado. Após esta divisão, a cada pacote é adicionado um cabeçalho contendo a locação-destino e a rota para o pacote. O pacote é enviado a uma chave que interpreta a primeira parte do cabeçalho e envia o pacote para o nó correto, removendo a parte do cabeçalho referente à rota já informada e deixando exposta a parte do cabeçalho referente ao nó-destino do pacote. Ao chegar ao nó-destino, cada pacote é autenticado (i.e., verifica-se se realmente se está esperando tal informação e se o cabeçalho está correto) e um reconhecimento (*acknowledgement*) é enviado para que o próximo pacote seja mandado, se este for o caso.

No caso de o pacote ter que passar por 2 chaves, 3 cabeçalhos são inseridos a cada pacote. Este será roteado através de 2 chaves e ocorrerão 2 remoções de cabeçalhos (*header deletion*) e apenas um será utilizado novamente no nó destino. Desta forma, é possível garantir que poderemos criar um número muito grande de canais virtuais. Comunicação utilizando canais virtuais nada mais é do que comunicação feita com pacotes de dados intercalados simulando um sistema com maior número de conexões do que realmente existem.

Quando é dito que há uma multiplexação dos canais, quer-se dizer que, uma vez que são dividos em pacotes, os dados a serem enviados podem ser intercalados, gerando uma rede de multiplexação de pacotes e, assim, satisfazendo um grande número de canais simultâneos, chamados virtuais.

A figura 5.1 pode ser elucidativa quanto à remoção de cabeçalhos.

Concluindo. A TN310 é um sistema distribuído com 16 nós de processamento tipo HTRAM totalmente conectados através de uma rede de chaves assíncronas. A comunicação nesta rede é feita através canais descritos em *software* (InMOS C), mas implementados via *hardware* através de um sistema de canais virtuais cujo processador central é o VCP (Processador de Canais Virtuais). O VCP divide a informação a ser mandada em pacotes e a cada pacote adiciona um número de cabeçalhos proporcional ao número de nós por onde o pacote passará. Assim sendo, se o pacote tiver que passar por 2 chaves adicionar-se-ão 3 cabeçalhos (2 para as chaves e 1 para o nó-destino). Os pacotes são enviados intercaladamente com outros pacotes de outras informações que também estão sendo enviadas. Isto cria uma rede de canais virtuais.

5.1.2 Os testes

Levando em consideração que estaremos alocando em cada nó de processamento uma única tarefa seqüencial, isto é, sem que dispare processos concorrentes, podemos chegar à conclusão que todos os DS-Links estarão à disposição da aplicação para a transmissão de dados. Isto maximiza a utilização dos meios de comunicação de cada nó. Estamos interessados em entender 2 pontos na operação do sistema:

1. Qual é o tamanho de cada pacote mandado por vez?

2. Quanto tempo demora para que pacotes sejam enviados a diferentes nós de processamento?

A primeira questão tem um motivo prático: sabendo o tamanho do pacote, podemos maximizar seu uso. A segunda questão se refere ao tempo gasto para que enviemos uma quantidade fixa de dados de um ponto para outro do sistema. Ao saber isto, estaremos aptos a tomar melhores decisões sobre a alocação de processos e avaliar melhor o quadro de operação de uma aplicação.

O teste realizado consiste em mandar dados de diferentes tamanhos para diferentes nós de processamento e medir o tempo gasto durante esta transferência. Saberemos que este tempo representa a melhor condição possível de operação, onde os 4 DS-Links estarão à disposição do processo de comunicação. O envio de dados ocorreu de forma iterativa, embora tenhamos descontado o tempo de iteração no final do processo. A iteração serviu para que pudéssemos avaliar mais precisamente o tempo de comunicação para os diversos tamanhos de dados.

Os resultados encontram-se na tabela 5.1. A tabela expõe na horizontal o tempo (em μsegundos) para que haja a transmissão do número de bytes indicados à direita através de 4 tipos diferentes de conexão. Cada tipo representa o número de chaves existentes entre os nós de processamento. Existem diferentes tipos de conexões, como foi visto na seção 3.2.4. A conexão do tipo 1 representa a conexão entre dois nós onde ocorre apenas uma chave; a do tipo 2, 2 chaves e, assim, sucessivamente, até 4 chaves (conexão entre os nós 0 e 8).

Com este resultado, percebemos que o tamanho do pacote mencionado anteriormente é de 32 bytes. Este valor é fixo para comunicações em qualquer distância. Isto quer dizer que a inclusão de mais ou menos cabeçalhos não influencia no tamanho do pacote enviado.

A aplicação foi feita de forma que as tarefas escravas receptoras de dados sempre estariam disponíveis para o recebimento de dados, nunca atrasando a tarefa-mestra. Um fator interessante é que podemos perceber que, em verdade, o que ocorre é uma distribuição de pacotes. Este fato fica marcante se olharmos a segunda parte da tabela, que indica "tamanho (em pacotes)". Ao compararmos o envio de 1 pacote, logo utilizando 1 dos 4 DS-Links disponíveis, o tempo de demora é um. Porém, se estivermos mandando mais de um pacote, logo utilizando mais de um DS-Link, o tempo por pacote cai, ainda que o tempo total aumente. Isto se deve ao fato de que o VCP processa os dados a serem enviados seqüencialmente, o que obrigatoriamente nos leva a um aumento do tempo, mas vemos que o simples fato de enviar 2 pacotes quase simultaneamente reduz o tempo total cerca de 16% (essa taxa reduz quanto maior o número de chaves na conexão chegando até 11%). Se enviarmos 3 pacotes a taxa de redução chega a 23%, para 4, a 26% e assim segue até que esta taxa atinja o valor de 34%, quando enviamos mais de 300 pacotes utilizando os 4 DS-Links. Esta taxa pode ser mencionada como taxa de saturação, i.e., representa o máximo na otimização de comunicação atingível (para conexões do tipo 1) usando 4 canais simultâneos, ao invés de apenas 1.

Estes resultados mostram que o envio de mais de 4 pacotes nesta situação (uma tarefa por processador) parece ser uma característica interessante em termos de otimização. Os processos podem ser ajustados para que o volume de comunicações não seja menor do que 4 pacotes por vez. Pelo fato da rede estar totalmente conectada através de chaves assíncronas e os DS-Links terem cada um uma conexão com a rede, se usarmos a configuração de uma tarefa (sem processos concorrentes) por processador, nunca teremos congestionamento de dados.

Depois do entendimento do processo de comunicação e troca de dados, veremos as implementações propostas no capítulo 4.

5.2 Resultados para a GDU

5.2.1 Replicando o JETNET

Implementamos 2 processos distintos para a unidade de decisões globais; no primeiro tentamos replicar os resultados obtidos com o pacote JETNET no ambiente do sistema TN310 (InMOS C), usando os pesos, *thresholds* e vetores de normalização achados durante a fase de treinamento neste primeiro ambiente (JETNET). A tabela 5.2 mostra os resultados de eficiência obtidos utilizando o programa do apêndice D. Quando mencionamos eficiência de uma RNA queremos destacar sua capacidade no

Tabela 5.1: Os tempos gastos para os diversos tamanhos de dados comunicados a diferentes nós de processamento. Tempos em microssegundos e tamanho em bytes ou pacotes de 32 bytes.

Tamanho (bytes)	Tamanho (pacotes)	Tipo 1	Tipo 2	Tipo 3	Tipo 4
1 até 32	1	10,2	12,1	13,8	15,5
33 até 64	2	16,9	20,6	24,0	27,6
65 até 96	3	23,5	28,9	34,2	39,5
100	4	30,2	37,5	44,6	51,8
150	5	37,2	46,0	54,7	63,6
200	7	50,5	62,9	75,3	87,6
250	8	57,2	71,4	85,4	99,5
300	10	70,8	88,2	106	123
350	11	77,3	96,7	116	136
400	13	90,6	114	136	159
450	15	104	130	157	183
500	16	111	139	167	195
550	18	124	156	188	219
600	19	131	164	198	231
650	21	114	181	218	255
700	22	151	190	228	261
750	24	164	206	249	291
800	25	171	215	255	303
850	27	184	232	219	327
900	29	197	245	300	351
950	30	204	257	310	363
1000	32	217	274	331	387
1500	47	317	400	484	567
2000	63	424	535	647	758
2500	79	531	670	810	950
3000	94	631	797	964	1130
3500	110	737	931	1130	1320
4000	125	838	1060	1280	1500
4500	141	944	1190	1440	1690
5000	157	1050	1330	1610	1880
5500	172	1150	1450	1760	2060
6000	188	1260	1590	1920	2260
6500	204	1360	1720	2090	2450
7000	219	1460	1850	2240	2630
7500	235	1570	1990	2400	2820
8000	250	1670	2110	2560	3000
8500	266	1790	2250	2720	3190
9000	282	1900	2400	2880	3380
9500	297	2000	2530	3040	3560
10000	313	2120	2670	3200	3760

Tabela 5.2: As eficiências para a rede 12-6-4 da GDU usando InMOS C. Os valores para π^- não foram computados por não representarem um volume de dados expressivo. As marcas "—" indicam que não aconteceram partículas daquele tipo no arquivo de entrada.

	$4e^-$	$2e^- + 2j$	$2e^- + 2\mu$	4μ
e^-	97.45	99.50	94.75	—
$jets$	93.40	96.13	96.40	100
μ	—	94.44	100	99.95

Tabela 5.3: A GDU concorrente, versão 1 - Mapa de alocação de processos.

Tarefa	Processador
Supervisor	0
15 GDU-s	1 ao 15

reconhecimento de padrões, assim, uma eficiência de 90% significa que de cada 100 partículas de um dado tipo a rede consegue **corretamente** identificar 90 e, assim, sucessivamente.

O tempo de processamento médio por RoI foi de 200 microssegundos. Este tempo leva em consideração somente o processamento neuronal da RoI; a leitura e escrita em disco foram realizadas sem contar tempo de processamento. O casamento perfeito entre a tabela 4.1 e a tabela 5.2 mostra a validade da substituição do cálculo utilizando séries (`tanh()`) por uma tabela de conversão. Esta substituição também reduz o tempo de processamento requerido por RoI.

5.2.2 A GDU concorrente

A segunda versão do programa, operando em 15 nós no sistema TN310; foi executada em 2 configurações: na primeira alocamos os processos como descrito na tabela 5.3.

Optamos por esta configuração para demostrarmos que diferentes posicionamentos dos processos podem nos levar a resultados bem diferentes. O tempo de processamento encontrado para uma ROI foi, aproximadamente, de 30,3 microssegundos. Consideramos o fator de agilização da aplicação (*speed-up*) como a relação entre o tempo gasto para processar 1 RoI na aplicação construída em apenas um nó do sistema TN310 e o tempo gasto para processar 1 RoI em qualquer outra configuração da mesma aplicação operando em mais de um nó de processamento do sistema. Neste caso, o *speed-up* da aplicação, quando a comparamos com a versão que roda em apenas 1 nó de processamento este resultado de 30,3 μs por RoI, fica em torno de 6,6.

Se utilizarmos a organização apontada na tabela 5.4, o tempo de processamento por evento cai para cerca de 27 microssegundos por RoI. Isto nos leva a um *speed-up* de 7,4.

Tabela 5.4: A GDU concorrente, versão 2 - Mapa de alocação de processos.

Tarefa	Processador
Supervisor	15
15 GDU-s	0 ao 14

5.2.3 Conclusões sobre a unidade de decisões globais

Os dados recolhidos aqui, sem contar com as eficiências, representam a média aritmética de milhares de iterações. Isto significa que estes dados estão tão próximos dos valores médios quanto foi possível chegar. De forma nenhuma os valores aqui apresentados representam dados espúrios ou ocasionais.

A tabela de conversão. A utilização da tabela de conversão provou-se muito eficiente como forma de substituição da aritmética complexa envolvida no cálculo de funções como a tangente hiperbólica. Neste caso específico, a substituição da função implementada por expansão em série por uma tabela de conversão mostrou-se ideal, visto que consiguimos diminuir o tempo de processamento sem termos reduzido a eficiência de separação da rede.

O tempo de processamento em 1 nó. O tempo de processamento por RoI em 1 nó encontrado (200 μs) é representativo do processamento neural da RoI, não incluindo de forma alguma as rotinas de leitura e escrita dos dados de cada RoI. O processamento neuronal, neste caso específico, consiste de 86 somatórios, 72 multiplicações e 10 ativações. Sabendo que 1 ciclo de um processador dura no mínimo 1 microssegundo[1] e que realizamos 168 operações distintas, sendo que 10 delas são mais complexas (ativação da `NET` dos neurônios) diríamos que o valor de 200μs por RoI é bem satisfatório e coerente.

Alocação de tarefas. Uma diferença de até 10% no tempo de processamento pode ser observado se realocarmos de forma eficiente os processos. Este fator exemplifica que o bom programador é aquele que conhece o equipamento com que trabalha, pois desta forma consegue aproveitar o máximo de cada recurso.

O *Speed-up*. O valor para o *speed-up* de 7,4 é um bom resultado. O valor máximo para esta característica é, no caso específico de 15 escravos concorrentes, 15. Uma vez que o tempo gasto na comunicação introduz uma seqüencialização na aplicação, como visto na seção 3.3.2, é de se esperar que este fator de agilização se reduza drasticamente.

De uma forma geral, a aplicação de várias técnicas como o paralelismo de dados, a conversão de valores por tabelas e a otimização baseada na organização de *hardware* conseguiram multiplicar em até 7,4 vezes a velocidade de execução de uma rede neuronal. A utilização de uma máquina de processamento distribuído neste caso mostrou-se eficaz.

5.3 Resultados para a implementação do segundo nível de *trigger*

Apresentaremos aqui os resultados obtidos com as 3 versões de implementação para o segundo nível de validação do experimento ATLAS/LHC. Destacamos aqui, como fizemos anteriormente, que atingimos a máxima otimização através de várias implementações como se seguem.

5.3.1 Resultados para o algoritmo completo rodando em apenas 1 processador

Foi simulado o processamento completo de uma RoI, 100 vezes (para tirarmos uma média), utilizando-se de apenas um nó de processamento. Este processamento consiste em, sequencialmente, extrair-se as características da RoI e passá-la na rede neural 12-6-4 das unidades de decisão global. O tempo de processamento para cada RoI foi de 1,7 ms em média. Este processamento não inclui, é claro, a

[1] Embora o *clock* seja de 20MHz (50ns por pulso) cada ciclo do *transputer* é executado em 1 microssegundo para processos em alta prioridade e em 64ms para processos em baixa prioridade.

Tabela 5.5: Os resultados obtidos com a implementação da primeira versão do simulador para o sistema de validação. Totais para 500 RoI-s.

	Tempo (ms)	% do tempo total
Distribuição de dados para Extratores	365.079	87,93
Perda com verificações redundantes	29.580	7,12
Total de aplicação	415.189	100

parte relativa às Redes Locais, que só têm sentido de implementação em sistemas distribuídos. Isto se deve ao fato de que Redes Locais representam processos de distribuição e recolhimento de dados, o que não ocorre em uma versão seqüencial da aplicação, pois não há distribuição dos dados por um sistema complexo. Os dados mantêm-se em uma única unidade de processamento por todo o tempo.

Os tempos simulados para os algoritmos de extratores estavam de acordo com os dados da tabela 4.4.

O valor de 1,7 ms será utilizado para calcularmos o *speed-up* das aplicações que se seguem. Utilizaremos o mesmo critério de cálculo utilizado na avaliação da agilização para a unidade de decisões globais: dividiremos o tempo de processamento de uma RoI processada por apenas um nó pelo tempo gasto para processar uma RoI em configurações que utilizem mais de um nó de processamento, como as que se seguem; assim, chegaremos ao que consideramos o *speed-up* da aplicação.

5.3.2 Resultados para a versão 1

Ao executarmos esta versão chegamos aos resultados contidos na tabela 5.5.

Podemos perceber que o maior "gargalo" de nossa aplicação encontra-se na distribuição de dados para os extratores de característica, somando aproximandamente 90% do tempo total da aplicação. Embora existam técnicas para otimizar a comunicação entre tarefas, não há como eliminarmos por completo ou em boa parte o tempo gasto nesta distribuição. Isto se deve a fatores puramente lógicos: a passagem de dados da aplicação supervisora para os extratores de característica é necessária, ou não teremos aplicação. Outro fator interessante a ser lembrado é que jamais conseguiremos otimizar o fluxo de dados nesta aplicação, pois todos os canais com extratores de característica demandam um alto volume de dados, portanto já utilizando mais de um DS-Link por vez.

O tempo de processamento para cada RoI fica em torno de 830 μs. O *speed-up* é:

$$speed - up = \frac{1700}{830} = 2,05,$$

que é um valor baixo. Tentamos a otimização por realocação dos processos como se segue.

5.3.3 Resultados para a versão 2

Ao realocarmos os processos de uma forma mais eficiente, esperamos que os tempos gastos em comunicação se reduzam; desta forma teríamos um aumento do *speed-up* da aplicação. Os resultados estão na tabela 5.6. Tentamos, também, eliminar o tempo gasto com a verificação de extratores cujos os dados já acabaram, tentando minimizar o tempo de processamento.

O tempo gasto com a verificação de extratores parados foi totalmente retirado, ainda que tenhamos sofrido alguma penalidade na redução das distâncias entre os processos, a medida que tivemos que adicionar código para a eliminação deste tempo extra.

O tempo de processamento para cada RoI baixou para 688 μs. Isto significa que o *speed-up* aumentou, vejamos:

$$speed - up = \frac{1700}{688} = 2,47.$$

Tabela 5.6: Os resultados obtidos com a implementação da segunda versão do simulador para o sistema de validação. Totais para 500 RoI-s.

	Tempo (ms)	% do tempo total
Distribuição de dados para Extratores	335.695	97,58
Perda com verificações redundantes	0	0
Total de aplicação	344.015	100

Tabela 5.7: Os resultados obtidos com a implementação da terceira versão do simulador para o sistema de validação. Totais para 500 RoI-s.

	Tempo (ms)	% do tempo total
Distribuição de dados para Extratores	194.427	96
Perda com verificações redundantes	0	0
Total de aplicação	203.453	100

Embora estejamos próximos de um aumento de 20%, estes ajustes ainda não caracterizariam uma boa "performance" pelo equipamento. Visto que uma abordagem tentando maximizar a utilização dos canais não aponta bons resultados, pois o único canal a ser otimizado é para a distribuição de dados para extratores de múons, que não se encontram tão distantes da otimização máxima, direcionamo-nos para a implementação utilizando a distribuição circular de eventos, como abordado na seção 4.2.1.

5.3.4 Resultados para a versão 3

Esta versão tira proveito da sequencialização do processo de distribuição de eventos e de rotinas especiais, otimizadas para a utilização na implementação em alto nível de canais virtuais criados em *hardware*, caso específico do Transputer T9000. Este conjunto de rotinas é conhecido como `DirectChan-s`, e funciona de forma idêntica às rotinas da tabela 3.2, apenas de forma mais rápida.

Os resultados desta versão de implementação encontram-se na tabela 5.7.

O tempo para o processamento de uma RoI atinge, nesta implementação, cerca de 390 μs. O *speed-up*:

$$speed - up = \frac{1700}{390} = 4,36.$$

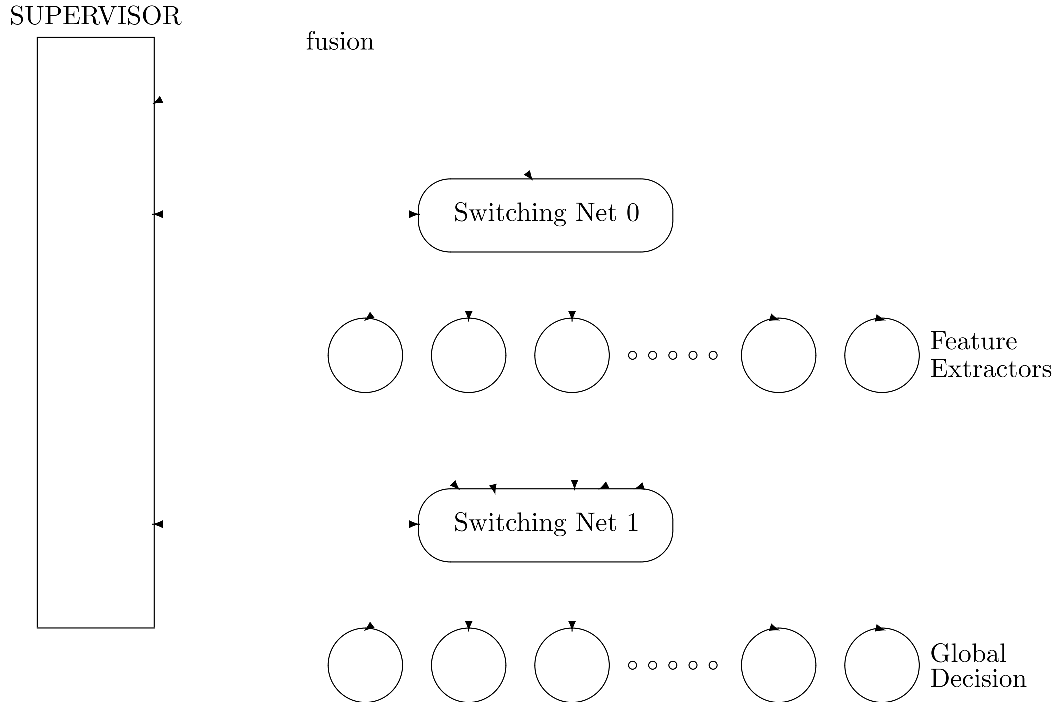
5.3.5 Concluindo sobre o sistema de validação

Cabe aqui uma pequena revisão do processo de paralelização como um todo; para isto repetiremos aqui algumas das figuras mostrando o processo de modelagem e implementação da aplicação.

Inicialmente estávamos diante de uma aplícação de difícil paralelização, mostrada na figura 4.8 e repetida aqui na figura 5.2. Esta aplicação possui 2 níveis de paralelização distintos, um segundo o seu fluxo, pois atua como um *pipeline* processando os dados no sentido dos extratores para as unidades de decisão global; e outro segundo seus dados. Neste caso estaremos atuando sobre um volume de dados cuja natureza e grandeza são as mesmas; poderemos, então, utilizar processamento repetido para que agilizemos o processo como um todo.

A aplicação destas duas técnicas neste último diagrama (figura 5.2) nos leva ao diagrama de tarefas da figura 5.3. Este diagrama foi arranjado de forma a maximizar a utilização dos processadores e minimizar as cargas sobre cada nó de processamento, balanceando o sistema.

Figura 5.2: O diagrama da aplicação implementada no sistema TN310.



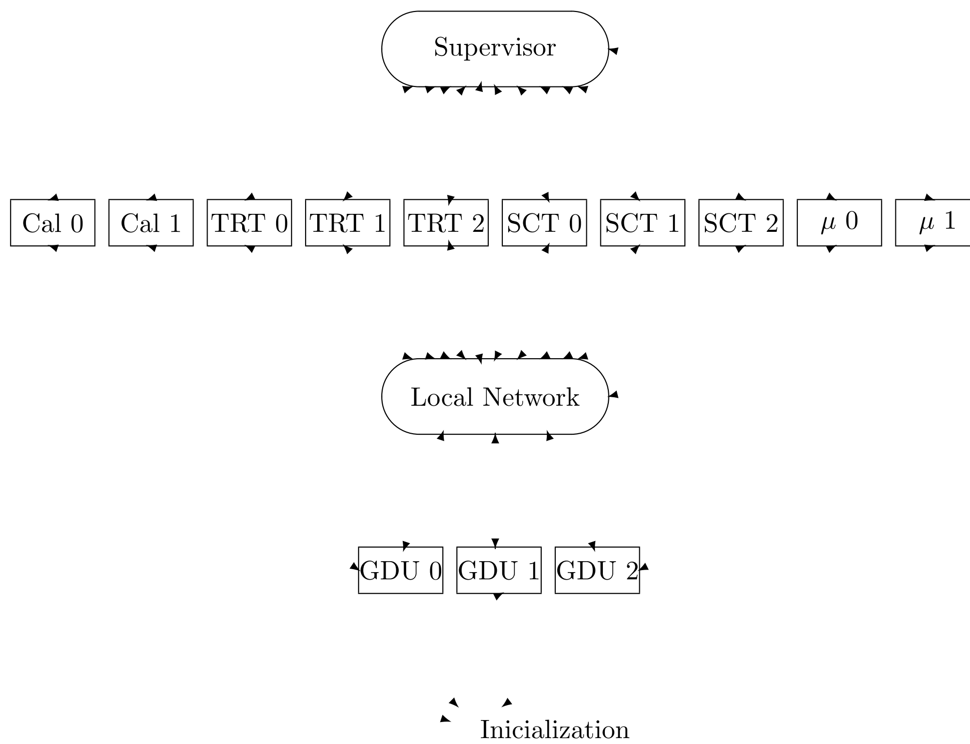
O *speed-up* para tal aplicação virá, portanto, na forma de 2 características adicionadas à execução do sistema, sua paralelização de dados e sua paralelização de fluxo. Se somente pudéssemos observar o paralelismo de dados, o *speed-up* máximo a ser encontrado estaria entre 2 e 3, pois possuímos apenas 2 unidades concorrentes para extratores de calorímetro e múons, ainda que tenhamos operando 3 unidades de processamento para extratores de TRT e 3 para SCT.

Porém, a existência de um paralelismo de fluxo nos leva a um acréscimo substancial no *speed-up* máximo atingível. Este acréscimo é diretamente proporcional ao número de subprocessos do *pipeline* que estão sendo executados paralelamente. Neste caso temos 2 subprocessos paralelos, a extração de características e a decisão global, levando-nos a um *speed-up* máximo entre 4 e 6. Isto pode ser visualizado com um simples exemplo:

> Imaginemos uma linha de montagem para automóveis. Um automóvel demora um tempo x para ser construído. Porém se tivermos y trabalhadores especilizados na montagem de partes exclusivas do automóvel, de forma que consigam terminar todo um veículo sem ajuda de mais trabalhadores, estes conseguirão imprimir uma velocidade maior no processamento de veículos. No caso idealizado, quando a linha de montagem já se encontra operacional (as diversas partes já possuem trabalho a ser feito) teremos a produção de y veículos no mesmo tempo x.

De forma mais genérica, se utilizarmos paralelismo de fluxo em uma aplicação que processa infinitamente, o *speed-up*, quando o número de produtos finais aumenta, tenderá ao número de subtarefas que são executadas simultaneamente. Assim, se 2 passos forem fluxo-paralelizados, o *speed-up* será, quando o número de dados processados for grande o suficiente, 2. Para 3 passos paralelizados, o *speed-up* passa para 3 e, assim, sucessivamente.

Figura 5.3: As tarefas da aplicação.



O resultado obtido para o *speed-up* em nossa aplicação foi de aproximadamente 4,4. Este resultado exprime quase 100% do *speed-up* máximo da aplicação, na configuração exposta. Seria de valor reduzido tentarmos reprojetá-la de forma a aumentarmos o *speed-up* máximo atingível[2], pois o tempo gasto com a distribuição de dados atinge, na máxima otimização, enormes 96% do tempo total de aplicação. Isto reflete que o rearranjo jamais se traduziria em aumento do *speed-up* final de forma significativa, podendo até reduzí-lo por estarmos desbalanceando o peso em cima das subtarefas de nossa aplicacação (deixando apenas uma unidade de decisão global operando, sem aplicarmos paralelismo de dados).

O sistema para o processamento de segundo nível

Por outro lado, o resultado obtido utilizando-se apenas 1 nó de processamento nos mostra que, talvez, para uma aplicação que não esta, fosse mais viável a utilização de muitos processadores operando sobre todo uma RoI (extração de característica e decisão global) do que a distribuição do processo por várias subtarefas. No entanto, isto feriria a configuração proposta para a arquitetura B (não factível, segundo [12]) do segundo nível de validação. De fato, este esquema estaria mais adaptado à arquitetura C para o segundo nível, cuja implementação através de nosso equipamento seja inviável, pois requeriria a utilização de chaves muito rápidas (ATM), que não se encontram disponíveis.

O fato de não possuirmos nós em número grande o suficiente para que consigamos suportar a taxa de fluxo do segundo nível de validação (100MHz), é, possivelmente a causa deste resultado. O tempo de processamento para uma RoI beira os 390 microssegundos; sabendo que o número de RoI-s por evento é, na média, de 5, isto nos dá um tempo de processamento de cerca de 2 ms por evento, que é um valor baixo. Baixo porque não estamos considerando aqui duas importantes etapas do processamento de um evento, o preprocessamento que consta da transformação da RoI geométrica disparada pelo nível 1 na RoI real, como explanado na seção 1.3.4 e a decisão sobre o canal físico do evento, parte integrante da tarefa de decisões globais. Levando isto em consideração, os 2 ms, ainda assim, representam a inexpressiva taxa de 500 Hz, baixíssima para suportar a operação do segundo nível de *trigger*.

Máximos e mínimos

Ao construirmos a aplicação para a unidade de processamento global percebemos que, ainda que utilizando distribuição circular (*Round-Robin*), atingimos um *speed-up* de 7,4 utilizando a configuração da figura 4.4. O *speed-up* máximo previsto, no entanto, era de 15, pois estávamos lidando com 15 escravos.

A questão é: “Porque o máximo que consiguimos foi 7,4?”. A resposta é um tanto complexa e nos basearemos nos resultados da tabela 5.1 para que a respondamos.

A distribuição circular provou-se útil pois, sempre que atingíamos o último escravo, o primeiro já estava livre; assim, jamais precisaríamos questionar a ocupação de um dado escravo já que saberíamos de ante-mão que ele estaria apto a receber novos dados e repassar os dados antigos já processados, se a distruição mantivesse a ordem (circular) inicial. Com este conhecimento é possível formular esta pergunta: “Qual é o número mínimo de escravos que necessito para que o primeiro **sempre** esteja livre quando o supervisor passar dados para o último?”. A resposta desta pergunta pode nos levar a 2 novos caminho de implementação:

1. **Reduzir o número de escravos:** Assim será se o número de processos-escravos necessários para mantermos fluxo de dados ininterrupto no supervisor for menor do que o que estamos utilizando (15). Isto significa que mais de um escravo está “parado” quando dados são distribuídos para o último.

2. **Emular mais escravos:** Assim será se o número de escravos necessários para manter o fluxo de dados constante e ininterrupto no supervisor for maior que o que temos agora (15).

[2]Isto seria possível se realocássemos 2 das 3 unidades de decisão global para trabalharem como os extratores ausentes para os subsistemas de calorímetro e múon.

Caso a resposta seja a primeira, deveremos provar que isto é verdade reduzindo o número de escravos ao número projetado e verificando se não há degradação da "performance" do sistema. Neste caso, mediremos isto pelo *speed-up* da aplicação, se este reduzir, notaremos que houve prejuízo na redução do número de escravos indicando necessidade de maior número destes processos. Caso não reduza comprovaremos que este desenvolvimento é coerente e que estamos atingindo o máximo desta aplicação na configuração proposta.

Caso a resposta seja a segunda, deveremos implementar um sistema de emulação. Durante uma emulação acontece uma espécie de *ensaio* onde simular-se-á o número de nós-de-processamento que serão suficientes para que atinjamos um fluxo de dados ininterrupto na aplicação supervisora. Assim, se chegarmos, por exemplo, que o número de escravos necessário é 30 devemos alocar em cada nó de processamento uma unidade de decisão global que recebe 2 RoI-s, mas que, na realidade, gasta o tempo de processamento de uma única região "emulando" a existência de mais nós-de-processamento, neste caso 2.

Uma unidade de decisão global, segundo a seção 5.2.1, demora 200 μs para processar uma RoI. A passagem de dados de/para escravos, como pode ser visto na aplicação descrita no apêndice E, acontece quando a aplicação supervisora recolhe os dados processados anteriormente pelo escravo e repassa novos dados para serem processados. Numa situação onde existam somente conexões do tipo 1 (isto é, com uma chave apenas ligando os processos supervisor e escravo) o tempo gasto nesta atividade será de, no mínimo, 27 microssegundos. Este número vem do seguinte cálculo: cada unidade de decisão global consome 12 números em ponto flutuante (reais); isto consome 48 bytes, que, segundo a seção 5.1, deve ser divido em 2 pacotes que demorarão (em uma conexão do tipo 1) cerca de 17 μs para que sejam entregues. Já os dados processados são em número de 4 (reais) e consomem 16 bytes que podem ser enviados à aplicação supervisor por através de um único pacote que leva cerca de 10 μs para que seja recebido (talvez um pouco mais). Esta troca de dados consumirá, então, os 27 μs mencionados.

Levando estes dados em consideração necessitaremos de

$$\frac{200\mu s}{27\mu s} \approx 7,4(\Longrightarrow 8)$$

escravos. Isto quer dizer que, se tivermos 8 escravos para distribuirmos dados **depois** de distribuirmos dados para o primeiro escravo, este, e somente este, estará livre quando acabarmos de distribuirmos dados para o último escravo. Seguindo a lógica, o segundo processo escravo, e somente este, estará livre quando acabarmos de distribuir dados para o primeiro e, assim, sucessivamente.

Isto está de acordo com a primeira assunção que fizemos alguns parágrafos acima no texto: o número de escravos é menor que 15, isto significa que não precisaremos emular uma versão da unidade de decisão global concorrente. Devemos então, a partir do esquema da figura 4.4 ir eliminado o número de escravos e verificar quando temos degradação de performance do sistema. De fato, como é possível ver na tabela 5.8 a "performance" do sistema não cai até que usemos menos de 9 escravos na GDU concorrente, re-afirmando a validade dos dados da tabela 5.1 e das assunções de funcionamento do sistema expostas anteriormente.

Outra importante observação é que o valor calculado (7,4) é, exatamente, o *speed-up* máximo para esta configuração e, ao mesmo tempo, o número de escravos (menos 1) necessários para que o supervisor mantenha-se sempre distribuindo dados.

É possível levantarmos as mesmas questões para o segundo nível de validação: "Será que estamos usando processadores demais?". Esta questão deve ser respondida da mesma forma que respondemos a questão similar para a GDU. Aqui, porém, deveremos nos concentrar na parte de distribuição de dados que mais pesa ao funcionamento do sistema como um todo, que é a distribuição de dados para os extratores de característica.

Esta distribuição acontece, diferentemente da que acontece na GDU concorrente, somente no sentido do supervisor para as unidades de extração e leva tempos distintos para diferentes sistemas de extração. A tabela 5.9 pode ser elucidativa quanto aos tempos gastos em conexões do tipo 1 para a pas-

Tabela 5.8: Os resultados para a GDU concorrente quando reduzimos o número de processos escravos.

Número de escravos	Tempo para 1 RoI	*speed-up*
15	27 μs	7,4
10	27 μs	7,4
9	**27** μs	**7,4**
8	**28,5** μs	**7,0**
7	32,5 μs	6,2
5	44,6 μs	4,5
3	74,2 μs	2,7

Tabela 5.9: Tempos gastos na passagem de dados da aplicação supervisora para os extratores.

Tipo de extrator	Tamanho (em bytes)	Tamanho (em pacotes)	Tempo
Calorímetro	504	16 (15,75)	111 μs
TRT	420	14 (13,13)	98 μs
SCT	420	14 (13,13)	98 μs
Múon	60	2 (1,88)	17 μs

sagem de dados do supervisor para os extratores. Já a tabela 5.10 mostra os tempos de processamento de cada unidade.

Para que seja possível atender o maior dos tempos de processamento (desta forma estaremos atendendo também os menores), neste caso o do extrator para o TRT (400 μs, deveremos ter tempos de comunicação com outros extratores quaisquer (inclusive com outros extratores para TRT), o mais próximo possível, ainda que maior, de 400 μs. Para que isto se concretize, **qualquer** configuração que atenda este critério poderá ser a configuração mínima para sistema.

Assim podemos pensar em várias destas configurações como se seguem:

1. **1 Ext.TRT, 2 Ext.SCT, 2 Ext.Calorímetro e 1 Ext.Múon**;
2. **1 Ext.TRT, 3 Ext.SCT, 1 Ext.Calorímetro e 1 Ext.Múon**;
3. **2 Ext.TRT, 1 Ext.SCT, 2 Ext.Calorímetro e 1 Ext.Múon**;
4. **2 Ext.TRT, 1 Ext.SCT, 1 Ext.Calorímetro e 6 Ext.Múon**
5. etc

Se o leitor somar os tempos de comunicação, verá que esta soma é mais que suficiente para cobrir quaisquer dos tempos de processamento. Desta forma, quando o dado for passado para o extrator

Tabela 5.10: O tempo gasto em processamento dos dados em cada tipo de extrator de característica.

Tipo de extrator	Tempo
Calorímetro	10μs
TRT	400μs
SCT (PreShower)	250μs
Múon	5μs

anterior a alguma unidade, esta já estará livre para receber mais dados, sempre. Para testarmos isto, projetamos um sistema muito parecido com o sistema da figura 5.3, porém, constando somente de 2 extratores para cada sub-sistema de deteção (ao invés de 2 para calorímetro, 3 para TRT, 3 para SCT e 2 para Múon) e constando de apenas 2 unidades de decisão global.

A razão de termos utilizado apenas 2 unidades vem do fato que, como temos distribuição circular, o tempo que o sistema demorará até que dados a serem analizados pelas unidades de decisão global estejam prontos, está em torno de 1,4 ms (tempos de processamento dos extratores somados), ainda que a cada 1,4 ms 2 RoI-s estejam prontas quase que simultaneamente. Para evitar "gargalos" 2 unidades de decisão global foram alocadas, desta forma, satisfazendo o fluxo de dados do sistema. Seria possível reduzir, ainda, o número de unidades de decisão global de 2 para 1 se, ao invés de distribuirmos dados para 2 extratores de TRT, em seguida 2 extratores de SCT e, assim, sucessivamente, fizéssemos isto de forma intercalada, ou seja, passássemos os dados para 1 extrator de TRT, em seguida para 1 extrator de SCT, daí para o de calorímetro e para o de múon, somente aí, então, passaríamos dados para os segundos extratores de cada subsistema. Desta forma, a aplicação produziria, ao invés de 2 RoI-s quase que simultaneamente a cada 1,4 ms, 1 RoI a cada 700 μs. Uma vez que a unidade de decisão global demora apenas 200 μs para processar uma RoI completa, utilizar somente uma destas aplicações não criaria "gargalo" neste sistema.

A rodarmos a aplicação sugerida verificamos que o tempo de processamento para uma RoI permaneceu em 390 μs, confirmando nossas expectativas. Isto se traduz no fato de que utilizar 3 unidades de decisão global, 3 extratores para TRT e 3 extratores para SCT durante a simulação constitui-se uma redundância desnecessária. Para a configuração proposta atingimos um *speed-up* máximo de 4,4, sem que utilizemos todo o potencial de processamento da máquina. Isto se deve basicamente a "lentidão" no processo de comunicações quando utilizamos *Transputers* conectados por redes de chaves STC104.

Conclusões gerais e principais resultados

Construímos um sistema de decisão global que consegue reproduzir fielmente os resultados obtidos com pacotes especializados de treinamento e teste de redes neurais em ambientes UNIX (JETNET 2.0). Este sistema é uma aplicação capaz de rodar em uma TN310 operando em um único nó ou em até 15 nós. Para este projeto, de forma a reduzir o tempo de processamento requerido pela unidade, resolvemos por utilizar a ativação neuronal por *Look-up* ao invés de utilizarmos a função residente `tanh` do ANSI C. Isto reduziu o tempo de processamento da unidade sem que tenhamos perdido qualidade no reconhecimento de padrões da rede. O menor tempo de processamento para cada RoI atingido foi 390 microssegundos.

Consideramos como *speed-up* a relação entre o tempo de processamento para um dado (RoI) no sistema operando em apenas 1 nó e o tempo de processamento de um dado no sistema operando em mais de 1 nó. Encontramos um *speed-up* máximo de 7,4 para a aplicação em questão.

Em aplicações como esta uma unidade supervisora é responsável pela distribuição e recolhimento dos dados processados pelas tarefas escravas. Verificamos (tabela 5.8) que a utilização de 15 nós de processamento é desnecessária, já que o supervisor (em casos da unidade estar rodando com mais de um nó) permanecerá distribuindo dados ininterruptamente se o número de unidades de decisão global (aplicações escravas) não for inferior a 9. Desta forma conclui-se que utilizar 16 nós de processamento para a aplicação nesta configuração é desnecessário.

Isto se deve basicamente ao fato da tecnologia em questão (*transputers* acoplados via STC104-s) não constituirem uma base eficiente o bastante para que o *speed-up* aumente com o número de nós-escravos. Estas conclusões abrangem somente o caso específico da aplicaçao na configuração mostrada na figura 4.4 e cujo volume de dados é tal como o descrito na seção 5.3.5.

Depois de construirmos a unidade de decisões globais passamos ao projeto de parte do segundo nível de validação para o experimento ATLAS/LHC (figura5.2). Esta aplicação **deve** ser paralelizada para que atinjamos o menor tempo de processamento possível para uma RoI utilizando o equipamento disponível (TN310). Para isto utilizamos técnicas de paralelismo de fluxo e dados na construção da aplicação vista na figura 5.3. Conseguimos chegar a um tempo de processamento por RoI de aproxi-

madamente 390 μs. Isto, como vimos na seção 5.3.5, traduz o limite do sistema para as configurações expostas na seção anterior. O *speed-up* encontrado foi de 4,4 quando comparamos esta aplicação com outra que executava as mesmas funções em um só nó de processamento (versão *single*).

Não obstante a estes fatos, provamos que a utilização de recursos híbridos de paralelização de aplicações mostra-se em vantagem contra a aplicação de apenas um dos métodos de paralelização expostos. O conhecimento do sistema como um todo nos levou a uma otimização significativa na velocidade de processamento; consiguimos atingir quase 100% do *speed-up* máximo desta aplicação utilizando os recursos do sistema de forma a maximizar a configuração proposta. Por final, a distribuição circular de eventos por processos escravos, pode, dependendo do caso onde se emprega[3], ser muito mais vantajosa do que a utilização de processamento sob-demanda. Obtivemos cerca de 40% de redução no tempo de distribuição, comparando a abordagem 3 com a abordagem utilizada na segunda versão do programa para o segundo nível de validação.

No próximo capítulo (6), estaremos discutindo algumas possíveis extensões deste trabalho e a possibilidade da inclusão de mais tarefas neste processamento.

[3]Neste caso específico temos o tempo de processamento das unidades extratoras muito menor do que o tempo que demoramos para passar os dados para os diversos escravos. Ainda, foi detectado que o tempo de *hand-shake* para qualquer dos extratores era um valor alto, cerca de 45 μs, que eram adicionados a cada tempo de canal.

Capítulo 6

Discussões

Este capítulo é dedicado à discussão sobre possíveis extensões deste trabalho. Estaremos expondo aqui algumas das características ainda não exploradas no equipamento disponível TN310, partes não incluídas na aplicação e possibilidades de emulação e expansão da aplicação.

6.1 Usando os DSP-s *on-board*

Esgotamos quase todos os recursos que nos levariam à otimização das rotinas de execução do sistema de validação. Maximizamos o processo de comunicação entre as tarefas de forma lógica e coerente e entendemos as diversas facetas e mecanismos de operação da TN310, aplicando estes recursos na redução do tempo de execução de nossa aplicação.

Dentre os recursos não utilizados estão os DSP-s embutidos em cada nó de processamento HTRAM. Estes DSP-s atuam como velocíssimos coprocessadores matemáticos e somente têm acesso ao mundo exterior graças a uma mémoria que compartilham com o *transputer* (T9000) da HTRAM.

DSP-s, como processadores matemáticos, podem ser utilizados de 2 formas em nossa aplicação: a primeira, fazendo com que todo o processamento matemático seja deslocado para estas unidades e, desta forma, reduzindo o tempo de processamento final; a segunda, utilizando os DSP-s como nós de processamento central de nossas tarefas, de tal forma que o tempo de execução de cada tarefa seja o mínimo possível.

A segunda forma de aplicação parece inviável, pois estaríamos tentando realocar o centro de processamento de uma HTRAM do *transputer* para o DSP. Pagaríamos então o preço por realocar tais atividades em um nó secundário, que se comunica através de outro nó ao meio exterior, aumentando o tempo de comunicação. Por outro lado, a implementação das tarefas no DSP teria que ser feita em *assembly* pois não existem rotinas de fluxo para tal processador como é descrito no manual [19][1].

A primeira forma de utilização do DSP é a mais coerente. Nesta abordagem, tentaríamos utilizar as bibliotecas disponíveis para o coprocessamento via DSP para dimunuir o tempo de processameto nas unidades que demandam grande quantidade de operações matemáticas, como é o caso das unidades de decisão global. Coprocessamento significa processar de forma auxiliar; isto implica que o nó de processamento central permaneceria como tal, ainda que operações matemáticas que demandassem alta velocidade fossem realocadas no DSP.

A primeira abordagem poderia utilizar as bibliotecas descritas no manual ([19]) para que o processamento matemático fosse deslocado do *transputer* para o DSP, dificultando menos a implantação do coprocessamento. Esta poderia ser considerado como mais uma forma de paralelização de nossa aplicação.

[1] Em verdade, não há nem referências de como implantar um código assembly no DSP. Suspeitamos de que o uso do DSP esteja restrito ao uso como coprocessador, mas nunca como processador.

A redução no tempo de processamento de cada unidade de decisão global, provocada pela utilização do DSP, irá reduzir o *speed-up* máximo da aplicação visto que estaremos agilizando o processamento sequencial ainda que o processamento distribuído não seja beneficiado pois estaremos presos na questão da distribuição de dados como visto na seção 5.3.5. Veja que o *speed-up* é exatamente o tempo de processamento em um único nó (que irá reduzir) dividido pelo tempo de transmissão de 3 pacotes (2 indo para o escravo e 1 voltando a cada vez). Este último não reduzirá pois é inerente ao sistema, o primeiro, no entanto, deve reduzir com o uso dos DSP-s *on-board* para realizar o co-processamento matemático do sistema, reduzindo, assim, o *speed-up*.

6.2 Fundindo aplicações e aumentando o número de processos por nó

Uma outra forma de tentarmos aumentar/otimizar a execução de tarefas pode vir pela maximização do processamento em cada nó. O arquivo de configuração exibido no apêndice F nos mostra que, no caso dos extratores de característica, apenas uma pequena parcela da memória total de cada nó dedicado a esta atividade está sendo utilizado (entre *stack*, *heap*, código e área estática o programa deve ocupar cerca de 50 Kilobytes de um total de 8 Megabytes disponíveis). É possível, então, que coloquemos tantas instâncias quantas forem possíveis de cada tipo de extrator em um nó, maximizando a utilização de tal nó.

Isto nos faz perceber 2 pontos distintos na execução de tal abordagem:

1. Com o aumento de tarefas rodando paralelamente, teremos uma possível diminuição do tempo de processamento global;
2. Pagaremos por este aumento, pois o número de DS-Links por nó é fixo; um maior número de extratores (ou unidades de decisão global) por nó implica uma maximização tão intensa no uso dos canais que atingiremos rapidamente a saturação.

Em outras palavras, a distribuição de dados é necessária; não podemos, em nenhuma configuração, deixar de fazê-la. Isto implica que, uma vez que estamos próximos da otimização máxima do uso de canais[2] para para vários das subtarefas, entre elas o supervisor (alto fluxo de dados), os extratores para calorímetro (16 pacotes por vez), SCT (14 pacotes por vez) e TRT (14 pacotes por vez), esta mudança somente atingiria de forma significativa os extratores de múons e as unidades decisão global. Como vimos, isto não representará uma alteração significativa do tempo de processamento global.

Uma outra possbilidade de otimização viria da fusão de algumas das tarefas da aplicação, aumentando o uso de cada nó e, ao mesmo tempo, aumentando o número de processadores disponíveis para processos mais lentos. Esta otimização poderia, por exemplo, ser realizada com a fusão do processo supervisor e das redes locais em um único processador. As diversas tarefas contituiriam parte de uma única tarefa que dispararia processos concorrentes (usando *time-sharing*) no mesmo processador. Isto mostrar-se-ia interessante, neste caso, pois o volume de dados transportados pelas redes locais[3] é diminuto e estaríamos eliminando a necessidade de comunicarmos os dados da rede local ao supervisor quando o processamento acabasse.

6.3 Anexando processos

Como trabalhos de extensão sugerimos a anexação das diversas fases de processamento que foram retiradas deste trabalho. Entre elas, podemos destacar o pré-processamento de Regiões de Interesse, a complementação da unidade de decisões globais e a adição de alguns extratores de característica reais.

[2] Esta discussão concerne ao uso de canais reais, os DS-Links.

[3] Um enfoque de fusão apenas das 2 redes locais (LN1 e LN0) também parece viável.

O pré-processamento, como já descrito na seção 1.3.4, constitui-se da parte do algoritmo de segundo nível responsável pela simplificação dos dados a serem processados pelos extratores de característica. Esta unidade, embora seja modelada pela arquitetura B com diferentes tipos de processadores, formando um *pipe*, pode ser implementada, por motivos de emulação, no sistema como um todo, aumentando a sua complexidade e fazendo com que possamos estimar de melhor forma o desempenho do sistema TN310 na execução da atividade de validação. O mesmo podemos dizer quanto aos extratores de característica.

A implementação da segunda fase de decisão global no processo final também parece ser um bom trabalho de continuação. Neste, teremos que dimensionar uma rede que possa receber até 100 entradas distintas para que determine o canal físico que representa um evento. Estas 100 entradas são resultado de cada evento possuir um máximo de 25 RoI-s; assim, se temos 4 saídas para cada RoI, um máximo de 100 entradas será suficiente para analisarmos qualquer evento. A rede em questão deverá ser dimensionada de forma a satisfazer o espaço de variáveis a que se destina, isto é, deverá possuir tantas saídas quantos foram os canais físicos a serem identificados.

O número de neurônios na camada intermediária estará diretamente relacionado com a linearidade do espaço de dados como discutido em seções anteriores. A rede deverá ser capaz de distinguir eventos com diferentes números de RoI-s.

A fase de treinamento pode ser realizada utilizando-se o JETNET, e a implementação no sistema TN310 deve seguir a implementação da rede para a identificação de partículas, usando, preferencialmente, as funções de ativação já construídas para esta aplicação (figura 4.3).

Referências Bibliográficas

[1] An introduction to particle physics. Internet WWW site (http://hepwww.rl.ac.uk/Pub/Phil/ppintro.html).

[2] University of Salamanca Nuclear Physics Group. Introduction to nuclear physics. Internet WWW site (http://www.usal.es/gfn/nucl_i.html).

[3] http://wwwlhc1.cern.ch/detectors/atlas/.

[4] The history of the lhc project. Internet WWW site (http://nicewww.cern.ch/lhcp/plo/history.htm).

[5] Level 2 Requirements group. Atlas level 2 trigger user requirements document. ATLAS/DAQ Note 79, CERN, December 1997. **Avaible in:** http://atlasinfo.cern.ch/Atlas/GROUPS/DAQTRIG/NOTES/note79/urdv1.ps.Z.

[6] Lívio P. Mapelli. Global architecture for the atlas daq and trigger - part 1 - functional model. RD13 Technical Note 136, CERN, January 1995. **Avaible in:** http://rd13doc.cern.ch/public/doc/Note136.html.

[7] Atlas level 1 trigger home page. Internet WWW site (http://atlasinfo.cern.ch/Atlas/GROUPS/DAQTRIG/LEVEL1/level1.html).

[8] J. M. Seixas, L. P. Calôba, A. L. Braga, E. Duarte, and E. Ribeiro. Global decisions for a neural triggering system in a high event rate environment. 1995.

[9] J. M. Seixas, B. Kastrup, L. P. Calôba, and R. Weber. Neural feature extraction for calorimeters in a distributed processing environment. 1997.

[10] R.K. Bock, J. Carter, and I.C. Legrand. What can artificial neural networks do for the global second level trigger. ATLAS/DAQ Note 11, CERN, March 1994. **Avaible in:** http://atlasinfo.cern.ch/Atlas/GROUPS/DAQTRIG/NOTES/note11/daq-11.ps.

[11] R.K. Bock, J. Carter, and I.C. Legrand. Real time, data driven architectures simulated in concurrent c++ for the lhc second-level trigger. *Third International Workshop on Software Engineering, Artificial Inteligence and Expert Systems for High Energy and Nuclear Physics*, 1993.

[12] R.K. Bock. Atlas level 2 trigger system requirements. Draft no.3, November 1995.

[13] Philip D. Wasserman. *Neural Computing Theory and Practice*. Van Nostrand Reinhold, New York, 1st. edition, 1989.

[14] Telmat's tn310 system user's course transparencies and annotations, September 1996. Ministered by Réné Pathenay.

[15] Telmat Multinode. *T9000 Ansi C ToolSet User Guide*. Analog Devices, Inc., France, 1995.

[16] Telmat Multinode. *IMS S7397 T9000 IBM PC Interface Software User Manual.* Analog Devices, Inc., France, 1995.

[17] Telmat Multinode. *T9000 ToolSet Reference Manual.* Analog Devices, Inc., France, 1995.

[18] Telmat Multinode. *T9000 ANSI C ToolSet Language and Libraries Reference Manual.* Analog Devices, Inc., France, 1995.

[19] Telmat Multinode. *DSP-HTRAM Software User Manual.* Analog Devices, Inc., France, 1995.

Apêndice A

Exemplo comentado de um arquivo de descrição de redes

Este apêndice transcreve o arquivo do PC hospedeiro em `C:\hardfile\tn310-2p.ndl` que contém informação de hardware para a configuração do sistema TN310 utilizando somente 2 dos 16 nós de processamento baseados em HTRAM. Os comentários virão em letras em letras comuns como qualquer texto deste documento, enquanto que o texto do arquivo estará em `letra tipo máquina-de-escrever, como aqui.`

Neste tipo de arquivo os comentários começam com `‘‘--’’` e uma vez identificado tal seqüência de caracteres todos os dados até o final da linha são ignorados.

```
---------------------------------------------------------------------------
-- TN310 Machine content --
---------------------------------------------------------------------------

-- Host 2008: EDGEDevice8
--Card 0 ,  Entry on EXTERNAL  by EDGE
--Link[0] (, Data Boot Link)
---------------------------------------------------------------------------
-- Host 3000: EDGEDevice1000
--Card 0 ,  Entry on EXTERNAL  by EDGE
--Link[Up] (, Control link)
---------------------------------------------------------------------------

-- Mother Card Number: 0 :

-- Node  0:  Dsp[0]
-- Node  2:  Dsp[1]

---------------------------------------------------------------------------
-- Processor(s) generals informations --
-- Group Dsp : -- 2 Processor(s) 'Dsp': T9000
--                 Type DSP , 8M , 20MHz, Size 4 , Id:, SubType:
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- Network description Language file for a TN310 system.
```

```
-- File created by the NDL Generator for TN series - version 2.0
-- (c) Copyright TELMAT Multinode - february 1996
---------------------------------------------------------------------------
```

Esta parte é gerada pelo programa `c:\t9bin\tmndl.exe` durante a confecção do arquivo NDL. Ela descreve de uma forma simplificada a rede a ser descrita. O programa em questão escreve um arquivo NDL a partir de um arquivo MCF, padrão criado pela TELMAT para simplificar a confecção de arquivos NDL. Entende-se que as informações contidas em arquivos MCF são instâncias simplificadas da descrição da rede que serão desenvolvidas no arquivo NDL. As informações acima, são as únicas que um arquivo MCF carrega. Todo o resto é desenvolvido pelo programa `tmndl.exe`.

A descrição em questão abrangerá a utilização de 2 nós de processamento HTRAM cujos nomes são Dsp[0] e Dsp[1]. Cada nó do tipo Dsp possui um transputer, 8Megabytes de memória RAM, opera a 20MHz e é tamanho 4.

Inclue algumas definições utilizadas no decorrer do arquivo, como o valor para K ou M.

```
#INCLUDE "stdndl.inc"
```

Define algumas variáveis como os endereços das diversas áreas da memória em `DspMemoryFlags` e a configuração da memória RAM on-chip que o transputer T9000 possui, neste caso para 8Kilobytes memória rápida e 8Kilobytes de cache.

```
VAL SysMem  IS 4*K :  -- Size of system RAM
VAL UsrBase IS #80000000 :  -- Start address of user RAM

VAL LinkSpeedMultiply IS 10 :
VAL LinkSpeedDivide IS 1 :
VAL ControlSpeedDivide IS 8 :

--
-- Nodes Definitions
--

VAL DspCacheSize IS 8 :
VAL DspMem IS (8*M) - SysMem :  -- Size of user RAM
VAL DspSysBase IS UsrBase+DspMem :  -- Start address of system RAM
VAL DspMemoryFlags IS [[DspSysBase, SysMem, RAM+SYSTEM],
                      [UsrBase, DspMem, RAM+USER],
                      [#00000000, 32, RAM+RESERVED],
                      [#7FFE0000, #20000, ROM]] :
[2]NODE Dsp :

VAL HOST.CONNECTION IS 16 : -- Data Link to the Host System

VAL DIR IS 4 :   -- Number of directions
VAL C104LinkSpeedMultiply IS 20 :
VAL C104LinkSpeedDivide IS 1 :
ARC HostLink :
[2][DIR] ARC LinkDsp :
```

Declara as chaves que utilizará na rede.

```
-- Switch declarations <Card 0>
NODE ControlSwitch0 :
NODE ControlPart0 :
```

```
NODE DataPart0Dir0 :
NODE DataPart0Dir1 :
NODE DataPart0Dir2 :
NODE DataPart0Dir3 :
[DIR]NODE DataSwitch0 :
--
```

Algumas características das chaves.

```
NETWORK
  DO
    -- Set default attributes
    SET DEFAULT (link.speed.multiply, link.speed.divide, control.speed.divide :=
                 LinkSpeedMultiply , [LinkSpeedDivide], [ControlSpeedDivide])

    --
    -- Set router attributes for Switches
    --
    --
    -- ControlSwitch0 initialisation
    --
    SET ControlSwitch0 (type := "C104")
    SET ControlSwitch0 (link.speed.multiply := C104LinkSpeedMultiply )
    DO J = 0 FOR 32
      SET ControlSwitch0 (link.speed.divide[J] := C104LinkSpeedDivide )
    SET ControlSwitch0 (link.groups := [[0,1],[2,3],[4,5],[6,7]])
    --
    -- Control Partition of Controlswitch <Card 0>
    SET ControlPart0 (type := "C104PARTITION")
    SET ControlPart0 (node := ControlSwitch0)
    SET ControlPart0 (links := [[20,20],[21,21],[28,28],[31,31]])
    --
    -- Data Partition 0 of Control switch <Card 0>
    SET DataPart0Dir0 (type := "C104PARTITION")
    SET DataPart0Dir0 (node := ControlSwitch0)
    SET DataPart0Dir0 (links := [[8,8],[0,1],[HOST.CONNECTION,HOST.CONNECTION]])
    --
    -- Data Partition 1 of Control switch <Card 0>
    SET DataPart0Dir1 (type := "C104PARTITION")
    SET DataPart0Dir1 (node := ControlSwitch0)
    SET DataPart0Dir1 (links := [[9,9],[2,3]])
    --
    -- Data Partition 2 of Control switch <Card 0>
    SET DataPart0Dir2 (type := "C104PARTITION")
    SET DataPart0Dir2 (node := ControlSwitch0)
    SET DataPart0Dir2 (links := [[10,10],[4,5]])
    --
    -- Data Partition 3 of Control switch <Card 0>
    SET DataPart0Dir3 (type := "C104PARTITION")
    SET DataPart0Dir3 (node := ControlSwitch0)
    SET DataPart0Dir3 (links := [[11,11],[6,7]])
    --
    --
```

```
-- Set the (Site 0/1) intervals for Dir 0 <Card 0>
SET DataPart0Dir0 (interval.separator[0] := 0)
SET DataPart0Dir0 (link.select[0] := 8)
SET DataPart0Dir0 (delete[8] := TRUE)
SET DataPart0Dir0 (interval.separator[1] := 1)

-- Set the (other Site) intervals for Dir 0 <Card 0>
SET DataPart0Dir0 (link.select[1] := 0)
SET DataPart0Dir0 (interval.separator[2] := 2)

-- Set the (Site 0/1) intervals for Dir 1 <Card 0>
SET DataPart0Dir1 (interval.separator[0] := 0)
SET DataPart0Dir1 (link.select[0] := 9)
SET DataPart0Dir1 (delete[9] := TRUE)
SET DataPart0Dir1 (interval.separator[1] := 1)

-- Set the (other Site) intervals for Dir 1 <Card 0>
SET DataPart0Dir1 (link.select[1] := 2)
SET DataPart0Dir1 (interval.separator[2] := 2)

-- Set the (Site 0/1) intervals for Dir 2 <Card 0>
SET DataPart0Dir2 (interval.separator[0] := 0)
SET DataPart0Dir2 (link.select[0] := 10)
SET DataPart0Dir2 (delete[10] := TRUE)
SET DataPart0Dir2 (interval.separator[1] := 1)

-- Set the (other Site) intervals for Dir 2 <Card 0>
SET DataPart0Dir2 (link.select[1] := 4)
SET DataPart0Dir2 (interval.separator[2] := 2)

-- Set the (Site 0/1) intervals for Dir 3 <Card 0>
SET DataPart0Dir3 (interval.separator[0] := 0)
SET DataPart0Dir3 (link.select[0] := 11)
SET DataPart0Dir3 (delete[11] := TRUE)
SET DataPart0Dir3 (interval.separator[1] := 1)

-- Set the (other Site) intervals for Dir 3 <Card 0>
SET DataPart0Dir3 (link.select[1] := 6)
SET DataPart0Dir3 (interval.separator[2] := 2)

-- Set the (Host system Data boot link) intervals
SET DataPart0Dir0 (link.select[2] := HOST.CONNECTION)
SET DataPart0Dir0 (delete[HOST.CONNECTION] := TRUE )
SET DataPart0Dir0 (interval.separator[3] := 3)

--
-- Data switches initialisations of <Card 0>
--
-- DataSwitch0 initialisation
--
DO I = 0 FOR DIR
```

```
    DO
      SET DataSwitch0[I] (type := "C104")
      SET DataSwitch0[I] (link.speed.multiply := C104LinkSpeedMultiply )
      DO J = 0 FOR 32
        SET DataSwitch0[I] (link.speed.divide[J] := C104LinkSpeedDivide )
  SET DataSwitch0[0] (link.groups :=  [[0,1]])
  SET DataSwitch0[1] (link.groups :=  [[0,1]])
  SET DataSwitch0[2] (link.groups :=  [[0,1]])
  SET DataSwitch0[3] (link.groups :=  [[0,1]])

      -- Set the (Site 0/1) intervals of <Card 0>
  SET DataSwitch0[0] (interval.separator[0] := 0 )
  SET DataSwitch0[0] (link.select[0] := 0 )
  SET DataSwitch0[1] (interval.separator[0] := 0 )
  SET DataSwitch0[1] (link.select[0] := 0 )
  SET DataSwitch0[2] (interval.separator[0] := 0 )
  SET DataSwitch0[2] (link.select[0] := 0 )
  SET DataSwitch0[3] (interval.separator[0] := 0 )
  SET DataSwitch0[3] (link.select[0] := 0 )
  SET DataSwitch0[0] (interval.separator[1] := 1 )
  SET DataSwitch0[1] (interval.separator[1] := 1 )
  SET DataSwitch0[2] (interval.separator[1] := 1 )
  SET DataSwitch0[3] (interval.separator[1] := 1 )

      -- Set the (other Sites) intervals of <Card 0>
  SET DataSwitch0[0] (link.select[1] := 2)
  SET DataSwitch0[0] (delete[2] := TRUE)
  SET DataSwitch0[0] (interval.separator[2] :=2 )
  SET DataSwitch0[1] (link.select[1] := 2)
  SET DataSwitch0[1] (delete[2] := TRUE)
  SET DataSwitch0[1] (interval.separator[2] :=2 )
  SET DataSwitch0[2] (link.select[1] := 2)
  SET DataSwitch0[2] (delete[2] := TRUE)
  SET DataSwitch0[2] (interval.separator[2] :=2 )
  SET DataSwitch0[3] (link.select[1] := 2)
  SET DataSwitch0[3] (delete[2] := TRUE)
  SET DataSwitch0[3] (interval.separator[2] :=2 )

      -- Set the (Host System Data boot link) intervals
  SET DataSwitch0[0] (link.select[2] := 0)
  SET DataSwitch0[0] (interval.separator[3] := 3 )
```

Inicializa e define algumas características dos nós de processamento, entre elas o nó raiz e e quantidade de memória disponível em cada nó.

```
  -- Set Processors default attributes
  DO I = 0 FOR 2
    DO
      SET Dsp[I] (type,memory := "T9000",DspMemoryFlags)
      SET Dsp[I] (memconfig := "d4-8-20.mem")
      SET Dsp[I] (cachesize := DspCacheSize)
      -- SET Dsp[I] (local.rom,pmi.config.inrom,cache.config.inrom,linkset.inrom
                     := TRUE,TRUE,TRUE,TRUE)
```

```
--
--
SET Dsp[0] (root := TRUE)
--
```

Conecta os processadores e chaves através de *links* de controle e *links* de dados. Depois finaliza com a instrução `‘‘:’’`.

```
-- Initialize Host Device(s)
--
-- CONTROL PIPELINE CONNECTIONS
--
CONNECT host[control] TO ControlSwitch0[control.up]

-- Control connections on processors <Card 0>
CONNECT ControlSwitch0[control.down] TO ControlPart0[link][31]
CONNECT ControlPart0[link][20] TO Dsp[0][control.up]
CONNECT ControlPart0[link][21] TO Dsp[1][control.up]
CONNECT ControlPart0[link][28] TO DataSwitch0[0][control.up]
DO I = 0 FOR (DIR-1)
  CONNECT DataSwitch0[I][control.down] TO DataSwitch0[I+1][control.up]

--
-- DATA TREE CONNECTIONS
--
CONNECT host[data] TO DataPart0Dir0[link][HOST.CONNECTION] WITH HostLink
--
DO I = 0 FOR DIR
  DO
    --
    IF
      I=0
        CONNECT DataPart0Dir0[link][8] TO Dsp[0][link][I] WITH LinkDsp[0][I]
      I=1
        CONNECT DataPart0Dir1[link][9] TO Dsp[0][link][I] WITH LinkDsp[0][I]
      I=2
        CONNECT DataPart0Dir2[link][10] TO Dsp[0][link][I] WITH LinkDsp[0][I]
      I=3
        CONNECT DataPart0Dir3[link][11] TO Dsp[0][link][I] WITH LinkDsp[0][I]
      TRUE
        SKIP
    CONNECT DataSwitch0[I][link][2] TO Dsp[1][link][I] WITH LinkDsp[1][I]
    --
    -- Connections between Ctrl and Data Switches
CONNECT DataPart0Dir0 [link][0] TO DataSwitch0[0][link][0]
CONNECT DataPart0Dir0 [link][1] TO DataSwitch0[0][link][1]
CONNECT DataPart0Dir1 [link][2] TO DataSwitch0[1][link][0]
CONNECT DataPart0Dir1 [link][3] TO DataSwitch0[1][link][1]
CONNECT DataPart0Dir2 [link][4] TO DataSwitch0[2][link][0]
CONNECT DataPart0Dir2 [link][5] TO DataSwitch0[2][link][1]
CONNECT DataPart0Dir3 [link][6] TO DataSwitch0[3][link][0]
CONNECT DataPart0Dir3 [link][7] TO DataSwitch0[3][link][1]
--
```

```
    -- Hosts connections to Edges
    --
:
-- End of File
```

Apêndice B

Exemplo comentado de arquivo CFS

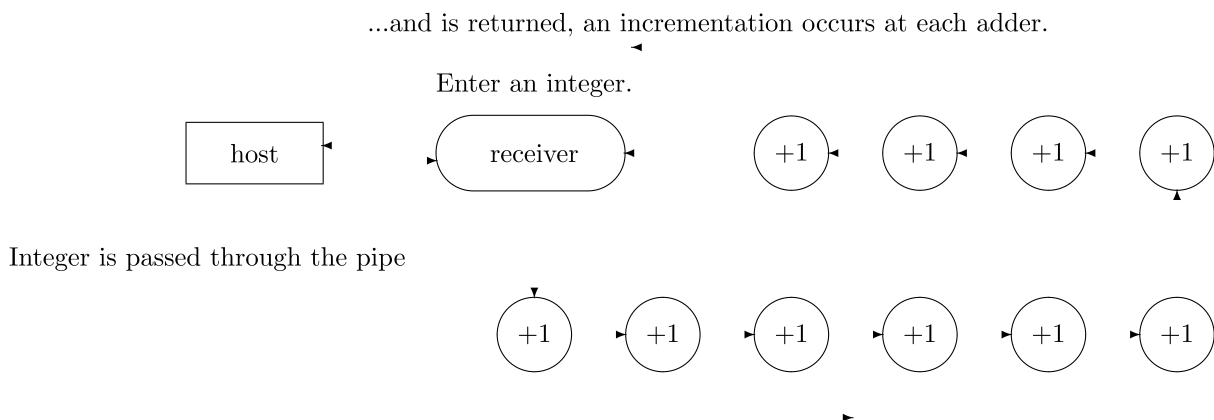
Neste apêndice apresentamos um arquivo de configuração da aplicação detalhadamente comentado. A aplicação à que se destina é simples: a tarefa principal, neste arquivo chamada de *receiver*, recebe do usuário um número inteiro e o repassa através de um canal para a primeira de 10 aplicações *adder*. Estas últimas têm a função de somar um ao número em seus canais de entrada e re-passar o número resultante ao próximo somador até que o último repasse o resultado obtido para a aplicação *receiver*. O diagrama na figura B.1 pode ser elucidativo quanto à topologia da aplicação. É uma das funções do arquivo de configuração da aplicação garantir esta topologia durante a execução.

Repare que somente uma aplicação fica conectada ao sistema hospedeiro[1], isto se deve ao fato desta conexão ser feita somente através de 2 canais de *hardware* unidirecionais. Não há necessidade da aplicação estar alocada em um nó especial, ainda que, como é possível se observar na figura 3.15, o nó mais apropriado para esta aplicação é o nó 0 (zero). Isto é verdade graças a existência de apenas uma chave conectando o nó em questão ao sistema hospedeiro, o que não é verdade para qualquer outro nó como visto na figura.

Deve-se também notar que o diagrama esboçado na figura B.1 de nenhuma forma é relacionado ou dependente do diagrama da figura 3.15. Isto é observável pela quantidade de processadores uti-

[1] A menos que utilize-se uma aplicação multiplexadora, todas as aplicações somente podem possuir 1 única tarefa conectada ao sistema hospedeiro.

Figura B.1: A topologia de conexões entre as tarefas da aplicação *PIPELINE*.



lizados na aplicação(11) comparado como o número de processadores disponíveis na inicialização do sistema(16). Ainda, também temos que o número de canais disponíveis por nó e totalizando 64 canais não são totalmente utilizados na aplicação. Durante a criação do arquivo de inicialização uma mensagem avisa o usuário que alguns dos nós não estão sendo utilizados ainda que isto não prejudique em nada a execução da aplicação.

Com isto em mente podemos re-afirmar(seção 3.2.5) que a programação de uma aplicação é feita em 2 níveis distintos: um no qual descreve-se a configuração do sistema em baixo nível usando NDL e o outro em descrevemos a aplicação com suas tarefas a serem executadas usando C Concorrente. O arquivo de configuração funciona como um "cimento" a estas duas descrições que devem operar fundidas num único arquivo de inicialização.

Agora o arquivo `pipeline.cfs`

Os comentários são como em C.

O arquivo de incialização a ser utilizado, e seu tipo. O tipo pode ser `ndl`, arquivo texto ou `nif` uma versão binária e mais confiável do arquivo ndl. Esta versão pode ser produzida utilizando-se os programas indl.exe e inif.exe localizados em `c:\d7394a\tools` as instruções podem ser encontradas em [17].

```
\# network "tn310.ndl" "ndl"
```

Declaração de algumas constantes, `pipe` é o número de somadores e processors o número de processadores disponíveis na máquina

```
val pipe 10;
val processors 16;
```

Declara as tarefas da aplicação (palavra-chave `process`), cada tarefa tem um espaço de stack(variáveis) bem definido assim como um espaço de heap. A palavra chave `interface` designa a passagem de parâmetros para a aplicação, os parâmetros incluem canais(palavras-chave `input`-canal de entrada e `output`-canal de saída), vetores, inteiros ou qualquer tipo conhecido no T9000 C ToolSet. Canais são sempre unidirecionais. Ao fim designa-se o nome da tarefa.

Tarefas iguais podem ser tratadas como vetores aqui. As aplicações `adder` são em número de 10 ao total, embora a declaração seja única.

```
process(stacksize=500k, heapsize=1000k, interface(input HostIn,
output HostOut, input From_pipe, output To_pipe)) receiver;

process(stacksize=500k, heapsize=1000k, interface(input NumIn,
output NumOut)) adder[pipe];
```

Declara canais sem aplicação específica

```
input hostin;
output hostout;
```

Atribui canais ao sistema hospedeiro, somente podem existir 2

```
place hostin on host;
place hostout on host;
```

Conecta o hospedeiro à tarefa `receiver`

```
connect receiver.HostOut to hostout;
connect receiver.HostIn to hostin;
```

Conecta o `receiver` ao *pipe*

```
connect receiver.From_pipe to adder[pipe-1].NumOut;
connect receiver.To_pipe to adder[0].NumIn;
```

Conecta cada um dos somadores entre si formando uma linha

```
rep i=1 for (pipe-1)
connect adder[i-1].NumOut to adder[i].NumIn;
```

Diz que utilizar-se-á o código em `receiver.c`(compilado e "linkado" às diversas bibliotecas na forma de `receiver.lku`) para a tarefa `receiver`

```
use "receiver.lku" for receiver;
```

Repete o procedimento anterior para cada somador usando `adder.c`

```
rep i=1 for pipe
use "adder.lku" for adder[i-1];
```

Identifica nó onde será alocada a tarefa `receiver`

```
place receiver on Dsp[0];
```

Faz o mesmo para cada somador, repare que a distribuição de tarefas ao longo dos processadores de dá de acordo com a constante `processors`. O comando `i%processors` retorna o resto da divisão de `i` por 16, assim se tivermos `pipe` maior que 15 não haverá problema pois será alocado em processadores válidos(o resto da divisão por 16 é menor que 16 sempre). Isto comprova a versatilidade de tal sistema de programação, o número de tarefas não é restrito ao número de nós de processamento, ou seja, é possível alocar mais de uma tarefa(mesmo com diferentes códigos) em um mesmo processador desde que se respeite a quantidade de memória alocada para cada uma. A exemplo, não é possivel alocar 5 tarefas em um mesmo processador se utilizarem um **total** de 2Megabytes cada pois cada nó somente possui 8Megabytes. Tal tentativa **não** é frustrada com um erro de compilação, mas aborta a execução aplicação.

```
rep i=1 for pipe
place adder[i-1] on Dsp[i%processors];
```

Apêndice C

Exemplo de arquivo Makefile

Aqui apresentaremos o exemplo comentado de um arquivo tipo `makefile`. Estes arquivos são utilizados juntos com o programa `make.exe` para facilitar o processo de compilação de programas em geral. Aqui o direcionamos para que compile aplicações em InMOS C(C Concorrente para o sistema TN310). A aplicação em questão utiliza 8 tipos diferentes de códigos em C. Isto, em maneira nenhuma, limita o número de processos a serem criados pois alguns deles podem usar o mesmo código como vimos no apêndice B.

Definimos inicialmente algumas constantes como o compilador `ICC` e *flags* em `ICCOPT`.

```
ICC=icc
ILINK=ilink
HCCONF=inconf
ICOLLECT=icollect
INIF=inif
IMAP=imap
ILINKOPT=-T9000 -GAMMAE -MO $(<:%.tco=%.dku)
HCCONFOPT=-GA
ICOLLECTOPT=-p level2.map
INIFOPT=
IMAPOPT=
ICCOPT=-GAMMAE -T9000 -G
SOURCES = master.c time_cal.c time_trt.c time_pre.c time_muo.c ln0.c ln1.c global.c
OBJECTS= $(SOURCES:%.c=%.tco)

.SUFFIXES : .c .tco
```

Agora define-se as diretivas de compilação, a diretiva `all` é a *default* sendo executada se o programa `make.exe` for chamado sem parâmetros. Um processo interessante aqui é que as únicas diretivas realmente independentes são as de compilação dos arquivos em C, todas as outras dependem da execução de outras diretivas(indicado ao lado do sinal ":"). Assim, se instruirmos o aplicativo para gerar `all`(ou `level2.btl`) teremos uma compilação recursiva que passará por todas as fases anteriores e necessárias a execução desta fase, i.e., haverá compilação do código em C, ele será "linkado", configurado e aí sim será coletado num arquivo com extensão `btl`.

```
all :  level2.btl

.c.tco :
$(ICC) $< $(ICCOPT) -o $@ -p $(<:%.c=%.mco)
```

```
level2.btl : level2.cfb
$(ICOLLECT) level2.cfb $(ICOLLECTOPT)

level2.cfb: level2.cfs master.lku time_cal.lku time_trt.lku time_pre.lku
            time_muo.lku ln0.lku ln1.lku global.lku
$(HCCONF) level2.cfs $(HCCONFOPT)

master.lku : master.tco
$(ILINK) master.tco -o master.lku -f cdebug.lnk $(ILINKOPT)

time_cal.lku : time_cal.tco
$(ILINK) time_cal.tco -o time_cal.lku -f cdebugrd.lnk $(ILINKOPT)

time_trt.lku : time_trt.tco
$(ILINK) time_trt.tco -o time_trt.lku -f cdebugrd.lnk $(ILINKOPT)

time_pre.lku : time_pre.tco
$(ILINK) time_pre.tco -o time_pre.lku -f cdebugrd.lnk $(ILINKOPT)

time_muo.lku : time_muo.tco
$(ILINK) time_muo.tco -o time_muo.lku -f cdebugrd.lnk $(ILINKOPT)

ln0.lku : ln0.tco
$(ILINK) ln0.tco -o ln0.lku -f cdebugrd.lnk $(ILINKOPT)

ln1.lku : ln1.tco
$(ILINK) ln1.tco -o ln1.lku -f cdebugrd.lnk $(ILINKOPT)

global.lku : global.tco
$(ILINK) global.tco -o global.lku -f cdebugrd.lnk $(ILINKOPT)

map : level2.btl
$(IMAP) level2.map -o level2.bcm  $(IMAPOPT)

clean:
 del  *.tco
 del  *.lku
 del  *.cfb
 del  *.btl
 del  *.map
 del  *.bcm
 del  *.mco
 del  *.dku
 del  *.bak
```

A diretiva `clean` comanda o apagamento de todos os arquivos gerados durante a compilação.

Apêndice D

GDU em InMOS C para 1 nó HTRAM

Neste apêndice estaremos descrevendo a unidade de decisões globais implementada para 1 nó HTRAM utilizando InMOS C para o sistema TN310. O arquivo de configuração não será exibido visto sua simplicidade. Os comentários estão em `letras comuns`, junto com o resto do programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <process.h>
#include <math.h>

/*================================================================*/
/* Diretivas DEFINE do programa, incluindo nomes e constantes      */
/*================================================================*/
#define NUMERO_DE_EVENTOS 5000
#define ARQUIVO_DE_EVENTOS "tst_resu\\4mu\\input.txt"
#define ARQUIVO_DE_PESOS "tst_resu\\4mu\\dump.txt"
#define TABELA_PARA_TANH "tanh.txt"
#define ARQUIVO_DE_NORMALIZACAO "tst_resu\\4mu\\normal.txt"
#define ARQUIVO_DE_SAIDA ".\\4mu.txt"

/* Tipos estruturados globais */
struct _hidd_ /* Estruturas para os neurônios da hidden layer */
{
 float threshold;
 float peso[12];
 float saida;
};

struct _outlay_ /* Estruturas para os neurônios da camada de saída */
{
 float threshold;
 float peso[12];
 float saida;
 };
```

```
/* Declaração de protótipos das funções utilizadas na aplicação */
float ativa(float valor, float *tb, float threshold); /* Função que
 calcula tanh por look-up */
void leia_tabela(float *tb);
void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol);
void leia_evento(FILE *fev, float *ev, int linha);
void leia_norma(float *norm);


/*=========================================*/
/* Programa Principal                      */
/*=========================================*/
int main(void)
{
 /* Arquivos utilizados por main */
 FILE *outnet, *fevento;

 /* Declaraçao de constantes utilizadas por main */

 const int NEVENTOS = NUMERO_DE_EVENTOS ; /* O número de eventos
 a serem analisados */

 /* Declaração de variáveis locais a main */
 int a,i,j,k; /* variáveis de controle */
 float evento[12]; /* O evento corrente */
 float normal[12]; /* O vetor de normalização */
 struct _hidd_ hidd[12]; /* Os neurônios da camada escondida */
 struct _outlay_ outlay[4]; /* Os neurônios da camada de saída */
 float tab[20001]; /* A tabela de valores para tanh */
 clock_t ev_tempo; /* variável de controle para o tempo do evento atual,
 em pulsos de clock */
 float tempo = 0.0; /* Acumulador para o tempo, em segundos */

 /* Avisa ao usuário que ele deve resetar a TN310
                                antes de começar. */
 printf("Voce \"resetou\" a TN310 antes de disparar a
                                    aplicação ???\n");
 printf("Se nao, selecione file/exit no menu acima e
                                comece de novo...\n");
 printf("\t<<<<<<<<<<<< Deseja continuar ?
                  Tecle <enter> >>>>>>>>>>>>\n");
 getchar();

 /* Lê tabela com valores das tangentes hiperbólicas */
 leia_tabela(tab);

 /* Lê arquivos de pesos */
 leia_dumps(hidd,outlay);

 /* Lê o arquivo com os valores de normalização */
 leia_norma(normal);
```

```
if((fevento=fopen(ARQUIVO_DE_EVENTOS,"r"))==NULL)
{
 printf("Não consegui abrir o arquivo de eventos,
                          terminando aplicação.");
 getchar();
 exit(-1);
} /* if: fim da abertura do arquivo de eventos */

/* Abre o arquivo de saída e verifica sua integridade */
if((outnet=fopen(ARQUIVO_DE_SAIDA,"a"))==NULL)
{
 puts("O arquivo de saída não foi aberto,
                terminando aplicação...\n");
 getchar();
 exit(-1);
}

printf("Iniciando processamento da GDU");
/* Inicia processsamento */
for(a=1; a <= NEVENTOS; a++)
{
 /* Lê evento atual */
 leia_evento(fevento, evento, a);

 /* Dispara cronômetro */
 ev_tempo = clock();

 /* Normaliza o evento */
 for(k=0;k<=11;k++)
 {
  evento[k] /= normal[k];
 }

 /* Processamento Neuronal do evento */

 /* Processamento da Hidden Layer
     ou Neurônios da camada escondida */
 for(i=0;i<=5;i++)
 {
  /*variáveis locais a este for (var i) */
  float soma=0;

  for(j=0;j<=11;j++)
  {
   /* Realiza a soma dos produtos internos da
      saída da camada de entrada com os pesos */
   soma+=(evento[j]*hidd[i].peso[j]);

  }/* for var j (2) */

  /* Passa pela função de ativação */
  hidd[i].saida=ativa(soma, tab, hidd[i].threshold);
```

```
  }/* for var i (2) */

  /* Processamento da Output Layer ou Neurônios da
     camada de saída */
  for(i=0;i<=3;i++)
  {
   /*variáveis locais a este for (var i) */
   float soma=0;

   for(j=0;j<=5;j++)
   {
    /* Realiza a soma dos produtos internos da saída da
       camada escondida com os pesos */
    soma+=(hidd[j].saida*outlay[i].peso[j]);

   }/* for var j (3) */

   /* Passa pela função de ativação 'ativa' */
   outlay[i].saida=ativa(soma, tab, outlay[i].threshold);

  }/* for var i (3) */

 ev_tempo = clock() - ev_tempo;
 tempo = tempo + (float)ev_tempo/CLOCKS_PER_SEC_HIGH;

 /* Imprime no arquivo de saida outnet.txt. */
 for(i=0;i<=3;i++)
 {
  fprintf(outnet,"%1.10f ",outlay[i].saida);
 }/* for var i (4) */

 fprintf(outnet,"\n");

 }/*for var a*/

 printf("\n\n\tTempo de processamento dos %d eventos:
              %e segundos.\n",NEVENTOS,tempo/NEVENTOS);
 fclose(outnet); /* Fecha o arquivo de saída */
 fclose(fevento); /* Fecha o arquivo de eventos */

 /* O usuário deverá teclar enter para terminar a aplicação */
 printf("\n\tTecle <enter> para terminar a aplicação...");
 getchar();

}/*main*/

/*====================================================*/
/* Funções chamadas pela rotina main                  */
/*====================================================*/
float ativa(float valor, float *tb, float threshold)
{
```

```
 /* Soma threshold do neurônio */
 valor+=threshold;

 /* Testa se valor é negativo */
 if( valor <= 0 ) /* 0 */
 {
  /* Caso seja, retorna o -1 para valor<-10 ou indexa a matriz tb
  segundo uma álgebra para identificação da binagem do histograma */
  if ( valor <= -10 ) /* 1 */
  {
   return -1.0;
  }/* then 1 */
  else
  {
   /* Acha a parte inteira da expressão à direita, em outras, trunca
   na terceira casa decimal (estou multiplicando por 1000) "valor" */
   int indice = 1000*valor+10000;
   return(*(tb+indice));
  }/* else 1 */

 }/* then 0 */
 else
 {
  /* Caso não, retorna o 1 para valor>10 ou indexa a matriz tb
  segundo uma álgebra para identificação da binagem do histograma */
  if ( valor > 10 ) /* 2 */
  {
   return 1.0;
  }/* then 2 */
  else
  {
   /* Acha a parte inteira da expressão à direita, em outras, trunca
   na terceira casa decimal (estou multiplicando por -1000) "valor" */
   int indice = 1000*valor+10000;
   return(*(tb+indice));
  }/* else 2 */

 }/* else 0 */

}/* Função ativa */

void leia_tabela(float *tb)
{
 /* Declaração de arquivos */
 FILE *tabela;

 /* Declaração de variáveis */
 int count;
 float lixo;/* Valores de x que serão lidos e jogados fora */

 /* Rotina Principal */
 tabela = fopen(TABELA_PARA_TANH,"r");
```

```
 if(tabela == NULL)
 {
  printf("Não consegui abrir tabela de valores para tanh,
           terminando aplicação.\n");
  exit(-1);
 }
 for(count=0;count<=20000;count++)/* Faz um for de 0 a 10000 */
 {
  fscanf(tabela,"%f %f",&lixo,(tb+count));
 }/* for */
 fclose(tabela);
}/* leia_tabela */

void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol)
{
 /* Arquivos utilizados */
 FILE *fid;

 /* Variáveis Locais */
 int count, count2;

 /* Rotina Principal */
 fid = fopen(ARQUIVO_DE_PESOS,"r");
 if(fid == NULL)
 {
  printf("Não consegui abrir o arquivo de pesos,
                        terminando aplicação.\n");
  exit(-1);
 } /* if */

 /* Lê os dumps da camada escondida */
 for(count=0;count<=11;count++)
 {
  for(count2=0;count2<=5;count2++)
  {
   fscanf(fid,"%f",&hd[count2].peso[count]);
  }/* for count2 */
 }/* for count */

 /* Lê threshold dos neurônios da camada escondida */
 for(count=0;count<=5;count++)
 {
  fscanf(fid,"%f",&hd[count].threshold);
 }/*for count (2o.) */

 /* Lê os dumps da camada de saída */
 for(count=0;count<=5;count++)
 {
  for(count2=0;count2<=3;count2++)
  {
   fscanf(fid,"%f",&ol[count2].peso[count]);
  }/* for count2 (2o.) */
```

```
 }/* for count (3o.) */

 /* Lê threshold dos neurônios da camada de saída */
 for(count=0;count<=3;count++)
 {
  fscanf(fid,"%f",&ol[count].threshold);
 }/*for count (4o.) */
}/* leia_pesos */

void leia_evento(FILE *fev, float *ev, int linha)
{
 /* Declaração de variáveis */
 int count;

 for(count=1;count<=12;count++)
 {
  fscanf(fev,"%f ",(ev+count-1));
 }/* for */

 printf("\tEvento %d lido.\n",linha); /* A variável
                     linha é o ponteiro de eventos */
}/* leia_eventos */

void leia_norma(float *norm)
{
 /* Declaração de arquivos */
 FILE *fid;

 /* Declaração de variáveis */
 int count;

 /* Rotina Principal */
 if((fid=fopen(ARQUIVO_DE_NORMALIZACAO,"r"))==NULL)
 {
  printf("Não consegui abrir o arquivo de normalização,
                              terminando aplicação.");
  getchar();
  exit(-1);
 } /* if */

 for(count=1;count<=12;count++)
 {
  fscanf(fid,"%f",(norm+count-1));
 }/* for */
 fclose(fid);
}/* leia_norma */
```

Apêndice E

O supervisor para GDU rodando em 16 processadores

Este apêndice é dedicado à comentários sobre o programa supervisor, partes que já foram apresentadas como a de leitura de arquivos e inicialização de variáveis(apêndice D) não serão discutidos com tanta ênfase como é o caso das partes novos, principalmente inerentes à comunicação. Além dos comentários no programa(`em letras de máquina`) será constante a inserção de texto comum para destacarmos o funcionamento de alguns pontos.

Inicialmente podemos perceber a inserção de 2 novas bibliotecas: process.h e channel.h, por razões óbvias.

```
/* Supervisor para a rede de decisão global */
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <misc.h>
#include <time.h>
#include <channel.h>

/*===============================================================*/
/* Diretivas DEFINE do programa, incluindo nomes e constantes    */
/*===============================================================*/

#define NUMERO_DE_EVENTOS 3238
#define ARQUIVO_DE_EVENTOS "..\\seq\\tst_resu\\2e2j\\input.txt"
#define ARQUIVO_DE_PESOS "..\\seq\\tst_resu\\2e2j\\dump.txt"
#define TABELA_PARA_TANH "..\\seq\\tanh.txt"
#define ARQUIVO_DE_NORMALIZACAO "..\\seq\\tst_resu\\2e2j\\normal.txt"
#define ARQUIVO_DE_SAIDA ".\\teste.txt"

/* Tipos estruturados globais */
struct _hidd_ /* Estruturas para os neurônios da hidden layer */
{
 float threshold;
 float peso[12];
 float saida;
};
```

```
struct _outlay_ /* Estruturas para os neurônios da camada de saída */
{
 float threshold;
 float peso[12];
 float saida;
 };


/* Declaração de protótipos das funções utilizadas na aplicação */
void leia_tabela(float *tb);
void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol);
void leia_norma(float *norm);
```

A função declarada a seguir, `CreateChannels()`, é uma função especial para a manipulação de um conjunto de canais declarados no arquivo CFS. Ela tem como entrada o endereço inicial de um vetor de canais e como saída um vetor de canais válidos no programa, i.e., ela declara vetores de canais criados no arquivo CFS. Isto indica que esta função é um ponteiro(outra notação para vetores em C) de ponteiros(os próprios canais). Uma peculiaridade é que o vetor que esta função retorna tem sua última posição apontando para `NULL`. Isto se deve à fatores de implementação de outras funções e inerente ao próprio InMOS C.

O atributo `static` somente serve para alocar a função em uma área especial da memória, não ocupando área de `stack`.

```
static Channel **CreateChannels (Channel *Channels[], int ChannelsSize);

/*===================================*/
/* Programa Principal                */
/*===================================*/

int main()
{

/* ********************************** */

/* ---------------------------- */
/* Begin Declaração de variáveis */
/* ---------------------------- */
```

A criação dos canais. É feita utilizando-se dos parâmetros cedidos pelo arquivo CFS. A função `CreateChannels` pode ser vista aqui sendo utilizada em sua plenitude.

```
 /* Criação dos canais */

 int InputSize = (int) *((long int *)get_param(4));
 int OutputSize = (int) *((long int *)get_param(6));
 Channel **Input = CreateChannels(get_param(3),InputSize);
 Channel **Output = CreateChannels(get_param(5),OutputSize);

 /* Arquivos utilizados por main */
 FILE *outnet, *fevento;

 /* Declaraçao de constantes utilizadas por main */
```

```
const int NEVENTOS = NUMERO_DE_EVENTOS ;
        /* O número de eventos a serem analisados */

/* Declaração de variáveis locais a main */

int a,i,j,k,l,m; /* variáveis de controle */
float evento[NUMERO_DE_EVENTOS][12]; /* Todos os eventos */
float saida[NUMERO_DE_EVENTOS][4]; /* Todas as saídas */
float normal[12]; /* O vetor de normalização */
struct _hidd_ hidd[12]; /* Os neurônios da camada escondida */
struct _outlay_ outlay[4]; /* Os neurônios da camada de saída */
float tab[20001]; /* A tabela de valores para tanh */
clock_t inicio, fim; /* variável de controle para o tempo do
                             evento atual , em pulsos de clock */
float acc = 0;
float tempo = 0; /* Acumulador para o tempo, em segundos */

printf("<globalm.c> Começando Global Decision Network,
                                 aguarde carregamento...\n");
printf("<globalm.c> \t\t<<<<<Particle identification >>>>>\n");

/* Lê tabela com valores das tangentes hiperbólicas */
leia_tabela(tab);
printf("<globalm.c> Li tabela.\n\n");

/* Lê arquivos de pesos */
leia_dumps(hidd,outlay);
printf("<globalm.c> Li pesos.\n\n");

/* Lê o arquivo com os valores de normalização */
leia_norma(normal);
printf("<globalm.c> Li normalização.\n\n");

/* Lê o arquivo com as ROI's */
fevento=fopen(ARQUIVO_DE_EVENTOS,"r");
if (fevento==NULL)
{
  printf("\n Arquivo de entrada vazio!!!\n");
  abort();
}

for(k=0;k<=NEVENTOS-1;k++)
  {
   for(i=0;i<=11;i++)
    {
      fscanf(fevento,"%f",&evento[k][i]);
    }
  }

printf("<globalm.c> Abrindo arquivo de saída...\n");
/* Abre o arquivo de saída e verifica sua integridade */
if((outnet=fopen(ARQUIVO_DE_SAIDA,"a"))==NULL)
```

```
 {
  puts("<globalm.c> O arquivo de saída não foi aberto,
                                  terminando aplicação...\n");
  getchar();
  exit(-1);
 }
 printf("<globalm.c> Arquivo de saída ok, começando
                                  processamento global...\n\n");
```

Começa o processamento em si, inicialmente configura os escravos, distribui pesos, tabelas etc.

```
 /* Envia dados de processamento para os slaves */
 /* ------------------------------------------- */

 printf("<globalm.c> Dados enviados para slaves:\n");
 for(i=0;i<=OutputSize-1;i++)
 {
  ChanOut(Output[i],tab,sizeof(tab));
 /* Tabela com tanh */
  ChanOut(Output[i],normal,sizeof(normal));
 /* Vetor de normalizacao */
  ChanOut(Output[i],hidd,sizeof(hidd));
 /* Neuronios da camada escondida */
  ChanOut(Output[i],outlay,sizeof(outlay));
 /* Neuronios da camada de saida */
  printf("Slaves carregados!\n");
 }/* end for(i) */
 printf("\n");

/* Fim do envio de dados para Global Decision Network */
/* -------------------------------------------------- */
/* Global Network pronta */
printf("Todas as Global Networks(slaves) estão carregadas,
                              prosseguir ? Pressione Enter.");
getchar();
```

Começa o processamento real, isto é, a processar RoI-s, a inicialização deve ser feita, todos os slaves estão disponíveis agora. Embora estejamos utilizando distribuição sequencial de eventos(sem ser sob-demanda) esta inicialização não é necessária ainda que a mantemos para não reduzir a flexibilidade do programa.

```
/* Passa dados novos para slaves,
      sem confirmação pois são os primeiros 15 */

 inicio=clock(); /* inicia contagem de tempo */
 for(a=0;a<=14;a++)
 {
  ChanOut(Output[a],evento[a],12*sizeof(float));
 }/* end for a */
```

Começa um *loop* até que o número pré-determinado de eventos se acabe. Repare que antes de passar dados novos o supervisor recolhe os antigos e os deposita numa matriz, na posição correta.

```
i=0;
for(a=15; a <= NEVENTOS-1; a++)
{
 ChanIn(Input[i],saida[a-15],4*sizeof(float));
                                  /* Recebe saída */
 ChanOut(Output[i],evento[a],12*sizeof(float));
                            /* Passa nova entrada */
 i++;
 if(i==15)
 {
  i=0;
 }
}/* End for a */
```

As últimas saídas recebem processamento diferenciado já que o supervisor não mais passará dados novos. Ininterruptamente usamos a variável "`i`" que já foi ajustada para o último processador que recebeu uma RoI.

```
/* Recebe as últimas saídas */
for(a=0;a<=14;a++)
{
 ChanIn(Input[i],saida[NEVENTOS-15+a],4*sizeof(float));
                                        /* Recebe saída */
 i++;
 if(i==15)
 {
  i=0;
 }
}

fim=clock(); /* finaliza tempo */
acc=(float)(fim-inicio); /* acumula tempo */
tempo=(float)(acc/1.0e6); /* Converte para segundos */

/* End of application, writing outnet to a file */
printf("<globalm.c> Acabei aplicação,
      escrevendo saída para arquivo...\n");
for(i=0;i<=NEVENTOS-1;i++)
{
 for(j=0;j<=3;j++)
 {
  fprintf(outnet,"%1.10f ",saida[i][j]);
 }
 fprintf(outnet,"\n");
}
```

Imprime o tempo total de aplicação, podemos calcular o *speed-up* com ele. Finaliza.

```
/* Imprime tempo total de aplicaçao */
printf("<globalm.c> Tempo de aplicação = %e",tempo);
fclose(fevento);
fclose(outnet);
getchar();
```

```
}/* End Main */

/*================================================*/
/* Funções chamadas pela rotina main            */
/*================================================*/

void leia_tabela(float *tb)
{
 /* lê toda tabela */

 /* Declaração de arquivos */
 FILE *tabela;

 /* Declaração de variáveis */
 int count;
 float lixo; /* Valores de x que serão lidos e jogados fora */

 /* Rotina Principal */
 tabela = fopen(TABELA_PARA_TANH,"r");
 if(tabela == NULL)
 {
  printf("<globalm.c> Não consegui abrir tabela de
            valores para tanh, terminando aplicação.\n");
  exit(-1);
 }
 for(count=0;count<=20000;count++) /* Faz um for de 0 a 20000 */
 {
  fscanf(tabela,"%f %f",&lixo,(tb+count));
 }/* for */
 fclose(tabela);
}/* leia_tabela */

void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol)
{
 /* Arquivos utilizados */
 FILE *fid;

 /* Variáveis Locais */
 int count, count2;

 /* Rotina Principal */
 fid = fopen(ARQUIVO_DE_PESOS,"r");
 if(fid == NULL) /* A flexibilização do programa incluirá
     a possibilidade de fornecer o nome do
     arquivo de pesos na chamada do programa. */
 {
  printf("<globalm.c> Não consegui abrir o arquivo de pesos,
                                     terminando aplicação.\n");
  exit(-1);
 } /* if */
```

```
 /* Lê os dumps da camada escondida */
 for(count=0;count<=11;count++)
 {
  for(count2=0;count2<=5;count2++)
  {
   fscanf(fid,"%f",&hd[count2].peso[count]);
  }/* for count2 */
 }/* for count */

 /* Lê threshold dos neurônios da camada escondida */
 for(count=0;count<=5;count++)
 {
  fscanf(fid,"%f",&hd[count].threshold);
 }/*for count (2o.) */

 /* Lê os dumps da camada de saída */
 for(count=0;count<=5;count++)
 {
  for(count2=0;count2<=3;count2++)
  {
   fscanf(fid,"%f",&ol[count2].peso[count]);
  }/* for count2 (2o.) */
 }/* for count (3o.) */

 /* Lê threshold dos neurônios da camada de saída */
 for(count=0;count<=3;count++)
 {
  fscanf(fid,"%f",&ol[count].threshold);
 }/*for count (4o.) */
}/* leia_pesos */

void leia_norma(float *norm)
{
 /* Declaração de arquivos */
 FILE *fid;

 /* Declaração de variáveis */
 int count;

 /* Rotina Principal */
 if((fid=fopen(ARQUIVO_DE_NORMALIZACAO,"r"))==NULL)
 {
  printf("Não consegui abrir o arquivo de normalização,
                                terminando aplicação.");
  getchar();
  exit(-1);
 } /* if */

 for(count=1;count<=12;count++)
 {
 fscanf(fid,"%f",(norm+count-1));
 }/* for */
```

```
 fclose(fid);
}/* leia_norma */

/* a funcao abaixo crias os canais de
                comunicacao com os slaves */
```

A função a seguir cria um vetor de canais baseado em um endereço, cedido pelo arquivo CFS através de `get_param()` e um inteiro indicando qual é o número de posições que terá este vetor.

```
static Channel **CreateChannels (Channel *Channels[],
                                        int ChannelsSize)
{
    /* Cria um ponteiro duplo que aponta para NULL */
    Channel **CopyChannels = NULL;

   /* Aloca espaço de ambiente para n+1 canais
      transformando o ponteiro duplo em um vetor de
      ponteiros, que é a mesma coisa, só que agora
                          existe espaço disponível */
   if ((CopyChannels = malloc((ChannelsSize + 1)
                      * sizeof(Channel *))) == NULL)
abort();
   else
    {
int ChannelsIndex = 0;

   /* Atribui a cada posição do vetor de
       canais um dos canais criados no arquivo CFS */

while (ChannelsIndex++ < ChannelsSize)
    CopyChannels[ChannelsIndex - 1]
               = Channels[ChannelsIndex - 1];

 /* Atribui NULL à última posição */
    CopyChannels[ChannelsSize] = NULL;
    }
    /* Retorna para a função chamadora, o novo vetor */
    return(CopyChannels);
}/* End CreateChannels */
```

Apêndice F

Versão final do simulador

Neste apêndice encontra-se uma cópia do código utilizado para programar o sistema de validação do segundo nível para o experimento ATLAS/LHC na TN310. A modularidade do problema nos levou a criação de 8 blocos diferentes de código em InMOS C. Estes blocos são implantados no sistema via arquivo "bootável" gerado a partir da fusão do código C com o arquivo CFS também listado aqui. O código dos arquivos é listado me `letra de máquina` assim como comentários inseridos.

F.1 Arquivo CFS (`level2.cfs`)

```
/* Configuração da rede de transputers */

#network "tn310.ndl" "ndl"

/* Descrição das tarefas */

process (stacksize=500k, heapsize=4000k, priority=high,
   interface(input host_in, output host_out, input in[10],
   int in_size = 10, output out[10],int out_size = 10,
   input f_ln1, output t_ln1, input f_gd[3],int gdin_size=3,
                     output t_gd[3], int gdout_size=3))master;

process (stacksize=2k, heapsize=20k, priority=high,
          interface(input in[2],int in_size = 2,
output out[2],int out_size = 2,int Ciclos = 5))cal[2];

process (stacksize=2k, heapsize=20k, priority=high,
          interface(input in[2],int in_size = 2,
output out[2],int out_size = 2,int Ciclos = 400))trt[3];

process (stacksize=2k, heapsize=20k, priority=high,
          interface(input in[2],int in_size = 2,
output out[2],int out_size = 2,int Ciclos = 200))pres[3];

process (stacksize=2k, heapsize=20k, priority=high,
          interface(input in[2],int in_size = 2,
output out[2],int out_size = 2,int Ciclos = 10))muon[2];
```

```
process (stacksize=400k, heapsize=4000k, priority=high,
          interface(input in[10],int in_size = 10,
output out[10],int out_size = 10, input f_ln1, output t_ln1))ln0;

process (stacksize=600k, heapsize=6000k, priority=high,
          interface(input in[3],int in_size = 3,
output out[3],int out_size = 3, input f_master, output t_master,
                                   input f_ln0, output t_ln0))ln1;

process (stacksize=100k, heapsize=1000k, priority=high,
          interface(input in[2],int in_size = 2,
               output out[2],int out_size = 2))gl[3];

/* End Descrição das tarefas */


/* Declaração dos canais do PC-Host */

input HostInput;
output HostOutput;

/* Master <-> PC-Host, std_in e std_out por default */
connect HostInput to master.host_in;
connect master.host_out to HostOutput;

/* Master <-> Redes Neurais dos Detetores */
connect master.out[0] to cal[0].in[0];
connect cal[0].out[0] to master.in[0];
connect master.out[1] to cal[1].in[0];
connect cal[1].out[0] to master.in[1];
connect master.out[2] to trt[0].in[0];
connect trt[0].out[0] to master.in[2];
connect master.out[3] to trt[1].in[0];
connect trt[1].out[0] to master.in[3];
connect master.out[4] to trt[2].in[0];
connect trt[2].out[0] to master.in[4];
connect master.out[5] to pres[0].in[0];
connect pres[0].out[0] to master.in[5];
connect master.out[6] to pres[1].in[0];
connect pres[1].out[0] to master.in[6];
connect master.out[7] to pres[2].in[0];
connect pres[2].out[0] to master.in[7];
connect master.out[8] to muon[0].in[0];
connect muon[0].out[0] to master.in[8];
connect master.out[9] to muon[1].in[0];
connect muon[1].out[0] to master.in[9];

/* Local NetWork_0 <-> Redes Neurais dos Detetores */
connect ln0.out[0] to cal[0].in[1];
connect cal[0].out[1] to ln0.in[0];
connect ln0.out[1] to cal[1].in[1];
connect cal[1].out[1] to ln0.in[1];
```

```
connect ln0.out[2] to trt[0].in[1];
connect trt[0].out[1] to ln0.in[2];
connect ln0.out[3] to trt[1].in[1];
connect trt[1].out[1] to ln0.in[3];
connect ln0.out[4] to trt[2].in[1];
connect trt[2].out[1] to ln0.in[4];
connect ln0.out[5] to pres[0].in[1];
connect pres[0].out[1] to ln0.in[5];
connect ln0.out[6] to pres[1].in[1];
connect pres[1].out[1] to ln0.in[6];
connect ln0.out[7] to pres[2].in[1];
connect pres[2].out[1] to ln0.in[7];
connect ln0.out[8] to muon[0].in[1];
connect muon[0].out[1] to ln0.in[8];
connect ln0.out[9] to muon[1].in[1];
connect muon[1].out[1] to ln0.in[9];

/* Master <-> Local Network_1 */
connect master.t_ln1 to ln1.f_master;
connect ln1.t_master to master.f_ln1;

/* Local NetWork_0 <-> Local NetWork_1 */
connect ln0.t_ln1 to ln1.f_ln0;
connect ln1.t_ln0 to ln0.f_ln1;

/* Master <-> Global Decision Units */
connect master.t_gd[0] to gl[0].in[0];
connect gl[0].out[0] to master.f_gd[0];
connect master.t_gd[1] to gl[1].in[0];
connect gl[1].out[0] to master.f_gd[1];
connect master.t_gd[2] to gl[2].in[0];
connect gl[2].out[0] to master.f_gd[2];

/* Local NetWork_1 <-> Global Decision Units */
connect gl[0].out[1] to ln1.in[0];
connect ln1.out[0] to gl[0].in[1];
connect gl[1].out[1] to ln1.in[1];
connect ln1.out[1] to gl[1].in[1];
connect gl[2].out[1] to ln1.in[2];
connect ln1.out[2] to gl[2].in[1];

/* Descrição das aplicações + alocação em transputers */
/* -------------------------------------------------- */

use "master.lku" for master;
place master on Dsp[1];

use "time_cal.lku" for cal[0];
place cal[0] on Dsp[4];

use "time_cal.lku" for cal[1];
place cal[1] on Dsp[7];
```

```
use "time_trt.lku" for trt[0];
place trt[0] on Dsp[2];

use "time_trt.lku" for trt[1];
place trt[1] on Dsp[3];

use "time_trt.lku" for trt[2];
place trt[2] on Dsp[9];

use "time_pre.lku" for pres[0];
place pres[0] on Dsp[5];

use "time_pre.lku" for pres[1];
place pres[1] on Dsp[6];

use "time_pre.lku" for pres[2];
place pres[2] on Dsp[10];

use "time_muo.lku" for muon[0];
place muon[0] on Dsp[11];

use "time_muo.lku" for muon[1];
place muon[1] on Dsp[12];

use "ln0.lku" for ln0;
place ln0 on Dsp[13];

use "ln1.lku" for ln1;
place ln1 on Dsp[14];

use "global.lku" for gl[0];
place gl[0] on Dsp[15];

use "global.lku" for gl[1];
place gl[1] on Dsp[8];

use "global.lku" for gl[2];
place gl[2] on Dsp[0];

/* Implementa canal de software, já declarado, */
/* em um link real.                            */
place HostInput on host;
place HostOutput on host;
```

F.2 Supervisor (master.c)

```
/* Simulation Approach 3 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <process.h>
#include <misc.h>
#include <time.h>
#include <channel.h>

/*============================================================*/
/* Diretivas DEFINE do programa, incluindo nomes e constantes */
/*============================================================*/

/* Define numero de ROI's para cada RN de detetores */
#define NROIS_MAX 500 /* Tem que ser múltiplo de 10 pois
              é o número de linhas das RoB's simuladas */
#define NO_EVENTOS 500
#define NO_ROIS_per_EV 10

/* Define arquivos com pesos, normalizaçao
                    e dados para Global Decision */
#define ARQUIVO_IN_GLOBAL "..\\input.txt"
#define TABELA_PARA_TANH "..\\tanh.txt"
#define ARQUIVO_DE_PESOS "..\\dumps.txt"
#define ARQUIVO_DE_NORMALIZACAO "..\\normal.txt"
#define ARQUIVO_DE_SAIDA "outnet3.txt"
#define ARQUIVO_DE_CHECK "time.txt"
#define NFEAT 12 /* Define number of features per RoI */
#define NOUTNET 4 /* Define the number of outputs
                                from the Global Network */
#define HEADER_SIZE 5 /* number of fields on the header */
#define NCAL 2 /* number of processors running
                                      CALORIMETER FEx'or */
#define NTRT 3 /* number of processors running TRT FEx'or */
#define NPRE 3 /* number of processors running SCT FEx'or */
#define NMUO 2 /* number of processors running MUON FEx'or */

/* Tipos estruturados globais */

struct _hidd_ /* Estruturas para os neurônios da hidden layer */
{
 float threshold;
 float peso[NFEAT];
 float saida;
};

struct _outlay_ /* Estruturas para os neurônios da camada de saída */
{
 float threshold;
 float peso[NFEAT];
 float saida;
 };

struct _on_ /* Estruturas para a matriz de dados de saída */
{
 float outnet[NOUTNET];
```

```
 int header[HEADER_SIZE];
};

struct _check_ /* Estruturas para checagem de funcionamento */
{
 int canal;
 int tipo;
 int num;
 int tempo;
};

/* Declaração de protótipos das funções utilizadas na aplicação */
void leia_tabela(float *tb);
void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol);
void leia_norma(float *norm);
static Channel **CreateChannels (Channel *Channels[],
                                       int ChannelsSize);
static void RotateChannels (Channel *Channels[],
                            int ChannelsSize, int Count);
static void RearrangeChannels (Channel *Channels[],int ChannelsSize);
void send_data(Channel *In, Channel *Out, int *det_info,
     int *hold_det, int *fim, float *vetor_det, int out_size, int ndet);

int main()
{

/* ***************************************** */

/* ---------------------------- */
/* Begin Declaração de variáveis */
/* ---------------------------- */


  /* Criação dos canais */

  int InputSize = (int) *((long int *) get_param(4));
  int OutputSize = (int) *((long int *) get_param(6));
  Channel **Input = CreateChannels(get_param(3), InputSize);
  Channel **Output = CreateChannels(get_param(5), OutputSize);

  Channel *from_ln1 = get_param(7);
  Channel *to_ln1 = get_param(8);
  int gdInSize = (int) *((long int *) get_param(10));
  int gdOutSize = (int) *((long int *) get_param(12));
  Channel **to_gd = CreateChannels(get_param(11), gdOutSize);

  /* End Channel creation */

   int i, j, k, fim; /* vars de controle */
   int hold_calo=0,hold_trt=0,hold_pres=0,hold_muon=0;
            /* vars de controle para a quantidade de ROI's enviadas */
   float matriz[NO_EVENTOS][NFEAT];
```

```
  struct _on_ matriz_prob[NO_EVENTOS][NO_ROIS_per_EV];
  struct _check_ vetor_check[4*NROIS_MAX];

  FILE *arq_ROIs;
  FILE *outnet_file;
  FILE *check_file;

  /* Vars Global Network */
  float normal[NFEAT]; /* O vetor de normalização */
  struct _hidd_ hidd[NFEAT]; /* Os neurônios da camada escondida */
  struct _outlay_ outlay[NOUTNET]; /* Os neurônios da camada de saída */
  float tab[20001]; /* A tabela de valores para tanh */

  /* Vars de tempo */
  int inicio, inicio_global, final, acc;
  int chan_calo=0, chan_trt=0, chan_pres=0, chan_muon=0;
  float chan_tot, fwaste, facc;
  unsigned long wasted=0; /* Guarda tempo perdido */


  /* Matrizes que simulam dados das RoB's */
  float vetor_calo[121], vetor_trt[100], vetor_pres[100], vetor_muon[10];


  int calo_info[HEADER_SIZE]; /* Declara vetor com dados da RoI */
  int trt_info[HEADER_SIZE];
  int pres_info[HEADER_SIZE];
  int muon_info[HEADER_SIZE];

  int tipo=0; /* Tipo de processador livre
          0-Calo, 1-TRT, 2-PreS, 3-Múon; default é calo */
  int fex_num=0; /* qual dos canais é o
       primeiro FEx'or (ordem inicial), geralmente calo */

  /* Current Date */
  time_t time_calendar;
  struct tm *time_struct;

  /* Process Organization from Dsp[0] to Dsp [15] */
  char proc[16][10] = { "global 2",
"master",
"trt 0",
      "trt 1",
      "calo 0",
     "pres 0",
      "pres 1",
"calo 1",
"global 1",
"trt 2",
"pres 2",
"muon 0",
   "muon 1",
```

```
    "ln 0",
    "ln 1",
 "global 0"};


/* -------------------------- */
/* End Declaração de variáveis */
/* -------------------------- */

/* **************************** */

/*--------------------*/
/* Início do programa */
/*--------------------*/

 /* Date init */

 time(&time_calendar);
 time_struct = localtime(&time_calendar);

 /* Inicializa matrizes RoB's */
 for(j=0;j<121;j++) vetor_calo[j]=j;
 for(j=0;j<100;j++)
 {
  vetor_trt[j]=j;
  vetor_pres[j]=j;
 }
 for(j=0;j<10;j++) vetor_muon[j]=j;
 /* End initialization */

printf(" Simulation Approach 03 for level 2 application\n");
printf("-----------------------------------------------\n");

/* le input.txt, dados a serem processados pela
            Global Decision Unit, mandados para LN1 */

 arq_ROIs=fopen(ARQUIVO_IN_GLOBAL,"r");
 if (arq_ROIs==NULL)
 {
   printf("\n<master.c>Arquivo de entrada vazio!!!\n");
   abort();
 }

 for(k=0;k<NO_EVENTOS;k++)
   {
    for(i=0;i<=11;i++)
     {
       fscanf(arq_ROIs,"%f",&matriz[k][i]);
     }
   }

/* Envia a matriz que contem os vet_ROI's para LN1 */
```

```
ChanOut(to_ln1,matriz,sizeof(matriz));
printf(" Sent true global data to Local Network 1.\n");


/* Feeding Global Decision Workers */

printf("Loading Global Decision Units...\n");
printf(" -- Particle identification ONLY\n");

/* Lê tabela com valores das tangentes hiperbólicas */
leia_tabela(tab);

/* Lê arquivos de pesos */
leia_dumps(hidd,outlay);

/* Lê o arquivo com os valores de normalização */
leia_norma(normal);

for(i=0;i<gdOutSize;i++)
{
 ChanOut(to_gd[i],tab,sizeof(tab)); /* Tabela com tanh */
 ChanOut(to_gd[i],normal,sizeof(normal)); /* Vetor de
                                             normalizacao */
 ChanOut(to_gd[i],hidd,sizeof(hidd)); /* Neuronios da
                                         camada escondida */
 ChanOut(to_gd[i],outlay,sizeof(outlay)); /* Neuronios da
                                          camada de saida */
 printf("GD Worker #%d, LOADED.\n",i);
}/* end for(i) */

 printf("\n");

/* Fim do envio de dados para Global Decision Network */
/* ------------------------------------------------- */

/* ***************************************************** */
/*   Processamento simulado para o segundo nível de trigger  */
/*               do experimento LHC/ATLAS                  */
/* ***************************************************** */

/* Etiqueta e envia dados, etiqueta e se este
                       é o ultimo (=1) ou não (=0) */

/* Inicializa variáveis de controle */
for(i=0;i<=4;i++)
{
 calo_info[i] = 0;
  trt_info[i] = 0;
 pres_info[i] = 0;
 muon_info[i] = 0;
}
```

```
k = 0; /* Where to begin */
j = 0; /* counter for check vector */
fim = 0; /* Indica final do processamento */

/* I want to begin by TRT and SCT, doing initial rotation */
/* Now => Input == trt0 trt1 trt2 pres0 pres1 pres2
                                  calo0 calo1 muon0 muon1 */
RearrangeChannels(Input, InputSize);
RearrangeChannels(Output, OutputSize);

printf("Processing, wait till end...\n");

inicio_global = ProcTime();

while(fim!=4) /* Soma fim dos 4 tipos de dados: calo, trt, pres e muon */
{
 for(i=k;i<InputSize;i++)
 {
  inicio = ProcTime(); /* Begins counter */
  switch(i)
 {
  case 0:
  case 1:
  case 2:
 if(hold_trt < NROIS_MAX)
 {
  vetor_check[j].num=hold_trt;
          /* Guarda numero da RoI em matriz para checagem */
  send_data(Input[i],Output[i],trt_info,&hold_trt,
                                         &fim,vetor_trt,100,NTRT);
 }
 else
 {
  vetor_check[j].num = -5; /* Indica que nao
                                          houve processamento */
  k = 3;
 }
 break;
  case 3:
  case 4:
  case 5:
 if(hold_pres < NROIS_MAX)
 {
  vetor_check[j].num=hold_pres;
          /* Guarda numero da RoI em matriz para checagem */
  send_data(Input[i],Output[i],pres_info,&hold_pres,
                                         &fim,vetor_pres,100,NPRE);
 }
 else
 {
  vetor_check[j].num = -5; /* Indica que nao
                                          houve processamento */
```

```
  k = 6;
 }
 break;
  case 6:
  case 7:
 if(hold_calo < NROIS_MAX)
 {
  vetor_check[j].num=hold_calo;
          /* Guarda numero da RoI em matriz para checagem */
  send_data(Input[i],Output[i],calo_info,&hold_calo,
                                      &fim,vetor_calo,121,NCAL);
 }
 else
 {
  vetor_check[j].num = -5; /* Indica que nao
                                        houve processamento */
  k = 8;
 }
 break;
  case 8:
  case 9:
 if(hold_muon < NROIS_MAX)
 {
  vetor_check[j].num=hold_muon;
         /* Guarda numero da RoI em matriz para checagem */
  send_data(Input[i],Output[i],muon_info,
                        &hold_muon,&fim,vetor_muon,10,NMUO);
 }
 else
 {
  vetor_check[j].num = -5;
           /* Indica que nao houve processamento */
  k = InputSize;
 }
 break;
  default: vetor_check[j].num = -5;
           /* Indica que nao houve processamento */
 } /* End switch */

  final=ProcTime(); /* End of cycle */
  acc=ProcTimeMinus(final,inicio);
  /* begin save check data */
  if(vetor_check[j].num != -5)
  {
vetor_check[j].canal = i; /* Channel number */
vetor_check[j].tipo = i; /* FEx'or type, RoI already saved */
/* Evaluates time in ticks */
vetor_check[j].tempo = acc; /* Saves time */
j++; /* Increments counter */
  }
  else
  {
```

```
wasted += acc; /* holds wasted time */
  }
  /* end save check data */
 }/* End for */

} /* End WHILE */


/* End Etiqueta e envia dados */

DirectChanIn(from_ln1,matriz_prob,sizeof(matriz_prob));
final=ProcTime(); /* End final time */
acc=ProcTimeMinus(final,inicio_global);
printf("LN1 is over!\n");

fclose(arq_ROIs);

/* Escrevendo saída para arquivo */

outnet_file = fopen(ARQUIVO_DE_SAIDA,"w");
if(outnet_file == NULL)
{
 printf("Couldn't open outnet file, closing application.\n");
 getchar();
 exit(-1);
}

for(i=0;i<500;i++)
{
 j=i%10;
 for(k=0;k<=3;k++)
 {
  fprintf(outnet_file,"%f ",matriz_prob[i][j].outnet[k]);
 }
 fprintf(outnet_file,"\n");
}

fclose(outnet_file);

/* Escrevendo dados de check para arquivo */

check_file = fopen(ARQUIVO_DE_CHECK,"w"); /* overwrite if
                                           already exists */
if(check_file == NULL)
{
 printf("Couldn't open time file, closing application.\n");
 getchar();
 exit(-1);
}

/* Doing some statistics */
for(j=0;j<(4*NROIS_MAX);j++)
```

```
{
 switch(vetor_check[j].tipo)
 {
  case 0:
  case 1:
  case 2:
  chan_trt += vetor_check[j].tempo;
          break;
  case 3:
  case 4:
  case 5:
  chan_pres += vetor_check[j].tempo;
  break;
  case 6:
  case 7:
  chan_calo += vetor_check[j].tempo;
          break;
  case 8:
  case 9:
  chan_muon += vetor_check[j].tempo;
          break;
 }
}

chan_tot = chan_calo + chan_trt + chan_pres +
    chan_muon + wasted; /* Total distribution time */
fwaste = (float)wasted ;
facc = (float)acc;

/* Print time file */
fprintf(check_file," Simulation Approach 03 for level 2 application\n");
fprintf(check_file," Current Date is %s\n",asctime(time_struct));
fprintf(check_file," Statistical Info for Time\n");
fprintf(check_file,"--------------------------\n\n");
fprintf(check_file,"         Value   |  Time  | %% compared to | %% compared to\n");
fprintf(check_file,"                 |        | tot.dist.time | tot. app. time\n");
fprintf(check_file,"-----------------+--------+---------------+---------------\n");
fprintf(check_file,"%14s   |%7d |%11.2f    |%11.2f\n",
 "calo",chan_calo,100*(chan_calo/chan_tot),100*(chan_calo/facc));
fprintf(check_file,"%14s   |%7d |%11.2f    |%11.2f\n",
 "trt",chan_trt,100*(chan_trt/chan_tot),100*(chan_trt/facc));
fprintf(check_file,"%14s   |%7d |%11.2f    |%11.2f\n",
 "pres",chan_pres,100*(chan_pres/chan_tot),100*(chan_pres/facc));
fprintf(check_file,"%14s   |%7d |%11.2f    |%11.2f\n",
 "muon",chan_muon,100*(chan_muon/chan_tot),100*(chan_muon/facc));
fprintf(check_file,"%14s   |%7lu |%11.2f    |%11.2f\n",
 "wasted",wasted,100*(fwaste/chan_tot),100*(fwaste/facc));
fprintf(check_file,"%14s   |%7.0f |%11.2f    |%11.2f\n",
 "used in dist.",chan_tot-wasted,100*((chan_tot-wasted)/chan_tot),
  100*((chan_tot-wasted)/facc));
fprintf(check_file,"%14s   |%7.0f |%11.2f    |%11.2f\n",
 "tot dist.time",chan_tot,100*(chan_tot/chan_tot),
```

```
 100*(chan_tot/facc));
fprintf(check_file,"%14s   |%7d |       ----    |%11.2f\n\n",
 "tot app.time",acc,100*(acc/facc));

fprintf(check_file," Process Organization Map is:\n");
fprintf(check_file," Node (Dsp) | Process Type \n");
fprintf(check_file,"------------+--------------\n");
for(j=0;j<16;j++)
 {
  fprintf(check_file,"%6d      | %10s \n",j,proc[j]);
  fprintf(check_file,"------------+--------------\n");
 }
fprintf(check_file,"\n RoI transmission time table follows...\n");
fprintf(check_file," FEx'or Type Legend: 0-2 TRT, 3-5
   SCT(PreShower), 6-7 Calorimeter, 8-9 Muon\n\n");
fprintf(check_file," Channel | Type | RoI |  Time\n");
fprintf(check_file,"---------+------+-----+-------\n");

for(j=0;j<(4*NROIS_MAX);j++)
{
 fprintf(check_file,"%8d | %4d | %3d | %5d\n",vetor_check[j].canal,
      vetor_check[j].tipo,vetor_check[j].num,vetor_check[j].tempo);
 fprintf(check_file,"---------+------+-----+-------\n");
}

fclose(check_file);

/* End write check */

printf("Final de aplicação, hit enter to finish");
getchar();

} /* End Main */


/* ------------------------- */
/* Funcoes chamadas por main */
/* ------------------------- */

void leia_tabela(float *tb)
{

 /* Declaração de arquivos */
 FILE *tabela;

 /* Declaração de variáveis */
 int count;
 float lixo; /* Valores de x que serão lidos e jogados fora */

 /* Rotina Principal */
 tabela = fopen(TABELA_PARA_TANH,"r");
 if(tabela == NULL)
```

```
 {
  printf("Não consegui abrir tabela de valores para tanh,
                                terminando aplicação.\n");
  exit(-1);
 }
 for(count=0;count<=20000;count++)
             /* Faz um for de 0 a 20000 */
 {
  fscanf(tabela,"%f %f",&lixo,(tb+count));
 }/* for */
 fclose(tabela);

}/* leia_tabela */

void leia_dumps(struct _hidd_ *hd, struct _outlay_ *ol)
{
 /* Arquivos utilizados */
 FILE *fid;

 /* Variáveis Locais */
 int count, count2;

 /* Rotina Principal */
 fid = fopen(ARQUIVO_DE_PESOS,"r");
 if(fid == NULL)
 {
  printf("Não consegui abrir o arquivo de pesos,
                       terminando aplicação.\n");
  exit(-1);
 } /* if */

 /* Lê os dumps da camada escondida */

 for(count=0;count<=11;count++)
 {
  for(count2=0;count2<=5;count2++)
  {
   fscanf(fid,"%f",&hd[count2].peso[count]);
  }/* for count2 */
 }/* for count */

 /* Lê threshold dos neurônios da camada escondida */
 for(count=0;count<=5;count++)
 {
  fscanf(fid,"%f",&hd[count].threshold);
 }/*for count (2o.) */

 /* Lê os dumps da camada de saída */

 for(count=0;count<=5;count++)
 {
  for(count2=0;count2<=3;count2++)
```

```
  {
   fscanf(fid,"%f",&ol[count2].peso[count]);
  }/* for count2 (2o.) */
 }/* for count (3o.) */

 /* Lê threshold dos neurônios da camada de saída */
 for(count=0;count<=3;count++)
 {
  fscanf(fid,"%f",&ol[count].threshold);
 }/*for count (4o.) */
}/* leia_pesos */

void leia_norma(float *norm)
{
 /* Declaração de arquivos */
 FILE *fid;

 /* Declaração de variáveis */
 int count;

 /* Rotina Principal */
 if((fid=fopen(ARQUIVO_DE_NORMALIZACAO,"r"))==NULL)
 {
  printf("Não consegui abrir o arquivo de normalização,
          terminando aplicação.");
  getchar();
  exit(-1);
 } /* if */

 for(count=1;count<=12;count++)
 {
  fscanf(fid,"%f",(norm+count-1));
 }/* for */
 fclose(fid);
}/* leia_norma */

/* a funcao abaixo crias os canais de comunicacao com os slaves */
static Channel **CreateChannels (Channel *Channels[],
                                         int ChannelsSize)
{
    Channel **CopyChannels = NULL;

    if ((CopyChannels = malloc((ChannelsSize + 1)
                      * sizeof(Channel *))) == NULL)
        abort();
   else
    {
        int ChannelsIndex = 0;

        while (ChannelsIndex++ < ChannelsSize)
            CopyChannels[ChannelsIndex - 1] =
                     Channels[ChannelsIndex - 1];
```

```
        CopyChannels[ChannelsSize] = NULL;
    }
    return(CopyChannels);
} /* End CreateChannels */

/* a função abaixo rotaciona os canais, i.e.,
         reverte a ordem de uma matriz de canais */
static void RotateChannels (Channel *Channels[],
                            int ChannelsSize, int Count)
{
    while (Count-- > 0)
    {
        int ChannelsIndex = 0;

        Channels[ChannelsSize] =
           Channels[ChannelsIndex]; /* NULL terminated */

        while (ChannelsIndex++ < ChannelsSize)
            Channels[ChannelsIndex - 1] =
                            Channels[ChannelsIndex];

        Channels[ChannelsSize] = NULL;
    }
}/* End RotateChannels */

static void RearrangeChannels (Channel *Channels[],int ChannelsSize)
{
 Channel *ch1, *ch2;
 RotateChannels(Channels,ChannelsSize,2);
                  /* Rotates Channels twice */

 /* Change calos and muons, muons last */
 ch1 = Channels[6];
 ch2 = Channels[7];
 Channels[6] = Channels[8];
 Channels[7] = Channels[9];
 Channels[8] = ch1;
 Channels[9] = ch2;

}/* End RearrangeChannels */

void send_data(Channel *In, Channel *Out, int *det_info,
 int *hold_det_ptr, int *fim_ptr, float *vetor_det,
 int out_size, int ndet)
{
 int hold_det = *hold_det_ptr; /* Passa valor para
  variável, que pode ser usada com operador de incremento ++ */
 int fim = *fim_ptr; /* Passa valor para variável,
  que pode ser usada com operador de incremento ++ */

 if(DirectChanInChar(In) == 'p')
```

```
  /* holds necessary channel - ProcAlt needs it */
 {
  DirectChanOut(Out,vetor_det,out_size*sizeof(float));
  if(++hold_det==NROIS_MAX)
  {
   fim++; /* Increments fim indicating end of this data type */
   det_info[2] = 1;
  }
  if((hold_det>(NROIS_MAX-ndet))&&(hold_det!=NROIS_MAX))
       det_info[2]=2; /* sets det_info[2] to finish FEx'ors */
  DirectChanOut(Out,det_info,HEADER_SIZE*sizeof(int));
  if(++det_info[0]==NO_EVENTOS)det_info[0]=0;
       /* Reseta posição do evento, se for o caso */
  if(++det_info[1]==NO_ROIS_per_EV)det_info[1]=0;
          /* Reseta posição da ROI, se for o caso */
  if(++det_info[3]==25)det_info[3]=0; /* Numero da RoI */
 } /* End if */
 else hold_det++; /* Não deixa de incrementar ponteiro
        ainda que hold_det > NROIS_MAX, para robustez */

 *hold_det_ptr = hold_det; /* Atualiza valor apontado */
 *fim_ptr = fim; /* Atualiza valor apontado */
} /* End send_data */
```

F.3 Extratores de característica

F.3.1 Calorímetro (time_cal.c)

```
#include <time.h>
#include <stdlib.h>
#include <channel.h>
#include <process.h>
#include <misc.h>
#define Entrada 121
#define Saida 6

/* a funcao abaixo crias os canais de comunicacao
   com o master */
static Channel **CreateChannels (Channel *Channels[],
                                       int ChannelsSize)
{
    Channel **CopyChannels = NULL;

    if ((CopyChannels = malloc((ChannelsSize + 1)
                          * sizeof(Channel *))) == NULL)
abort();
   else
    {
int ChannelsIndex = 0;

while (ChannelsIndex++ < ChannelsSize)
```

```
    CopyChannels[ChannelsIndex - 1] =
                      Channels[ChannelsIndex - 1];

CopyChannels[ChannelsSize] = NULL;
    }
    return(CopyChannels);
}

int main(void)

{
 int InputSize = (int) *((long int *) get_param(2));
 int OutputSize = (int) *((long int *) get_param(4));
 Channel **Input = CreateChannels(get_param(1), InputSize);
 Channel **Output = CreateChannels(get_param(3), OutputSize);
 int Ciclos = (int) *((long int *) get_param(5));

 int fim=0; /* Variavel de controle */
 float vet_in[Entrada]; /* Numero de entradas do algoritmo */
 float vet_out[Saida]; /* Numero de saidas do algoritmo */
 int vet_info[5]; /* vetor com informações do processo,
                         ev, roi e se é o último ou não */

while(fim < 1) /* Opera até o último dado */
{
 DirectChanOutChar(Output[0],'p');
 DirectChanIn(Input[0],vet_in,sizeof(vet_in));  /* Data in */
 DirectChanIn(Input[0],vet_info,sizeof(vet_info));
                                             /* Header in */
 fim = vet_info[2]; /* stop signing to master and ln0 */
 if(fim == 2) vet_info[2] = 0;

 /* Process data */

 ProcWait(Ciclos);

 /* End Process data */

 /* Begin Send output to LN0 */

 DirectChanOut(Output[1],vet_info,sizeof(vet_info));
                                 /* Send header to LN1 */
 DirectChanOut(Output[1],vet_out,sizeof(vet_out));
                                      /* Send data */

 /* End Send output to LN0 */

} /* End while */
ChanInChar(Input[0]); /* Waits forever */
} /*end main*/
```

F.3.2 TRT (time_trt.c)

A única alteração com relação ao programa da seção F.3.1 é:

```
#define Entrada 100
#define Saida 2
```

F.3.3 SCT (time_sct.c)

A única alteração com relação ao programa da seção F.3.1 é:

```
#define Entrada 100
#define Saida 3
```

F.3.4 Múon (time_muo.c)

A única alteração com relação ao programa da seção F.3.1 é:

```
#define Entrada 10
#define Saida 1
```

F.4 Rede Local

F.4.1 LN0 (ln0.c)

```
#include <stdlib.h>
#include <math.h>
#include <process.h>
#include <misc.h>

/* Diretivas define do programa */
#define NEVENTO_MAX 500
#define NROI_per_EV_MAX 10
#define NFEAT 12
#define HEADER_SIZE 5

/* Define structures usadas por main */

struct _roi_ /* Estruturas para a matriz de dados */
{
 float ext_feat[NFEAT];
 int header[HEADER_SIZE];
};

/* End structure declaration */

/* Declare prototypes for functions */

static Channel **CreateChannels (Channel *Channels[],
                                     int ChannelsSize);
static void RotateChannels (Channel *Channels[],
                           int ChannelsSize, int Count);
static void RearrangeChannels (Channel *Channels[],
```

```
                                     int ChannelsSize);
void recv_data(Channel *In, int *det_info,
 struct _roi_ mat[NEVENTO_MAX][NROI_per_EV_MAX], int *final_ptr,
 int in_size, int pos);

/* End prototype declaration */

int main ()
{
  /* Begin Channel creation */

  int InputSize = (int) *((long int *) get_param(2));
  int OutputSize = (int) *((long int *) get_param(4));
  Channel **Input = CreateChannels(get_param(1), InputSize);
  Channel **Output = CreateChannels(get_param(3), OutputSize);

  /* Channel *from_ln1 = get_param(5); Not used */
  Channel *to_ln1 = get_param(6);

  /* End Channel creation */

  int i, j=0, k=0;
  struct _roi_ matriz[NEVENTO_MAX][NROI_per_EV_MAX];
  int final=0;
  int vet_info[HEADER_SIZE];
  int tipo=0; /* Tipo de processador livre 0,1-Calo, 2,3,4-TRT,
                 5,6,7-PreS, 8,9-Múon; default é calo */
  int fex_num=0; /* qual dos canais é o primeiro FEx'or
                    (ordem inicial), 0 = CALO */

/* End variable declaration */

/* Initialize matriz.header[4] */
for(i=0;i<NEVENTO_MAX;i++)
 {
  for(j=0;j<NROI_per_EV_MAX;j++)
 {
  matriz[i][j].header[4]=0;
 }
 }
/* End matriz.headar[4] initialization */

/* New order 3x TRT, 3x SCT, 2x CALOS and 2x MUONS */

RearrangeChannels(Input, InputSize);
RearrangeChannels(Output, OutputSize);


while(final!=4)
 {

  for(i=k;i<InputSize;i++) /* Sequential receiving */
```

```
  {
 switch(i)
 {
  case 0:
  case 1:
  case 2: if(final < 1) recv_data(Input[i], vet_info,
                matriz, &final, 2, 6);  /* TRT */
                  else k=3;
                  break;
  case 3:
  case 4:
  case 5: if(final < 2) recv_data(Input[i], vet_info,
                matriz, &final, 3, 8);  /* SCT */
            else k=6;
  break;
  case 6:
  case 7: if(final < 3) recv_data(Input[i], vet_info,
                matriz, &final, 6, 0);  /* CAL */
                  else k=8;
  break;
  case 8:
  case 9: if(final < 4) recv_data(Input[i], vet_info,
                matriz, &final, 1, 11); /* MU  */
                  else k=InputSize;
  break;
 } /* End switch */

 /* Test if header equals to 4, if yes send data to LN1 */
    if(matriz[vet_info[0]][vet_info[1]].header[4]==4)
    {
     DirectChanOut(to_ln1,vet_info,sizeof(vet_info));
     DirectChanOut(to_ln1,&matriz[vet_info[0]][vet_info[1]],
      sizeof(struct _roi_));
    }/* End data parsing to LN1 */

  }/* End for */

 } /* End while */

} /* End Main */

/* Funções chamadas por main */

/* a funcao abaixo crias os canais de
   comunicacao com os slaves */
static Channel **CreateChannels (Channel *Channels[],
                                       int ChannelsSize)
{
    Channel **CopyChannels = NULL;

    if ((CopyChannels = malloc((ChannelsSize + 1)
                      * sizeof(Channel *))) == NULL)
```

```
        abort();
   else
    {
        int ChannelsIndex = 0;

        while (ChannelsIndex++ < ChannelsSize)
            CopyChannels[ChannelsIndex - 1] =
                      Channels[ChannelsIndex - 1];

        CopyChannels[ChannelsSize] = NULL;
    }
    return(CopyChannels);
} /* End CreateChannels */

/* a função abaixo rotaciona os canais, i.e.,
      reverte a ordem de uma matriz de canais */
static void RotateChannels (Channel *Channels[],
                      int ChannelsSize, int Count)
{
    while (Count-- > 0)
    {
        int ChannelsIndex = 0;

        Channels[ChannelsSize] = Channels[ChannelsIndex];
                                   /* NULL terminated */

        while (ChannelsIndex++ < ChannelsSize)
            Channels[ChannelsIndex - 1] =
                         Channels[ChannelsIndex];

        Channels[ChannelsSize] = NULL;
    }
}/* End RotateChannels */

static void RearrangeChannels (Channel *Channels[],
                                     int ChannelsSize)
{
 Channel *ch1, *ch2;
 RotateChannels(Channels,ChannelsSize,2);
                        /* Rotates Channels twice */

 /* Change calos and muons, muons last */
 ch1 = Channels[6];
 ch2 = Channels[7];
 Channels[6] = Channels[8];
 Channels[7] = Channels[9];
 Channels[8] = ch1;
 Channels[9] = ch2;

}/* End RearrangeChannels */

void recv_data(Channel *In, int *det_info,
```

```
 struct _roi_ mat[NEVENTO_MAX][NROI_per_EV_MAX],
 int *final_ptr, int in_size, int pos)
{

 int count;
 int final = *final_ptr;
 /* recebe vetor com informações do evento */
 DirectChanIn(In,det_info,HEADER_SIZE*sizeof(int));
 /* Recebe 2 valores das RN do TRT */
 DirectChanIn(In,&mat[det_info[0]][det_info[1]].ext_feat[pos],
  in_size*sizeof(float));
 mat[det_info[0]][det_info[1]].header[4]++;
                  /* Increment feature counter */
 if(mat[det_info[0]][det_info[1]].header[4]==1)
                     /* First time, save header */
 {
  for(count=0;count<=3;count++)
  {
mat[det_info[0]][det_info[1]].header[count] = det_info[count];
                                                 /* save header */
  }
 }
 final += det_info[2]; /* Acumula que estamos
                  no final dos dados para TRT */
 *final_ptr = final;

} /* End recv_data */
```

F.4.2 LN1 (ln1.c)

```
#include <stdlib.h>
#include <math.h>
#include <process.h>
#include <misc.h>

/* Define directives for this application */
#define NEVENTO_MAX 500 /* Define maximum
                            number of RoI's */
#define NFEAT 12 /* Define number of
                               features per RoI */
#define NOUTNET 4 /* Define the number of
                 outputs from the Global Network */
#define NROI_per_EV_MAX 10 /* Define maximum
                       number of RoI's per event */
#define HEADER_SIZE 5 /* number of fields
                                on the header */
#define GDU0 0 /* Channel number to comunicate
                                 with Global Worker 0 */
#define NUM_OF_GDW 3 /* Defines the number of Global
                                      Decision Workers */

/* Define structures used by main */
```

```
struct _roi_ /* Data structure for GDW data matrix */
{
 float ext_feat[NFEAT];
 int header[HEADER_SIZE];
};

struct _on_ /* Data structure for output matrix */
{
 float outnet[NOUTNET];
 int header[HEADER_SIZE];
};

/* End structure declaration */

/* Define functions used by main application */

static Channel **CreateChannels (Channel *Channels[],
                                       int ChannelsSize);

/* End define prototypes */

int main()
{
  /* Begin Channel Declaration */
  int InputSize = (int) *((long int *) get_param(2));
                                         /* GD Workers */
  int OutputSize = (int) *((long int *) get_param(4));
  Channel **Input = CreateChannels(get_param(1), InputSize);
  Channel **Output = CreateChannels(get_param(3), OutputSize);

  Channel *from_master = get_param(5);
  Channel *to_master = get_param(6);
  Channel *from_ln0 = get_param(7);
  /* End Channel Declaration */

  /* Begin simple variable declaration */

  int i,j,k, linha=0;
  int fim=0, inicio=0;
  int gd[NUM_OF_GDW][HEADER_SIZE]; /* Headers Acumulator */
  int vet_info[HEADER_SIZE]; /* evento e RoI control matrix */
  struct _roi_ matriz_dados[NEVENTO_MAX][NROI_per_EV_MAX];
  struct _on_ matriz_prob[NEVENTO_MAX][NROI_per_EV_MAX];
  float matriz_real[NEVENTO_MAX][NFEAT]; /* where is going
               to be stored valid Global Decision features */

  /* End variable declaration */


/* Initializes matriz_prob */
for(i=0;i<NEVENTO_MAX;i++)
```

```
 {
  for(j=0;j<NROI_per_EV_MAX;j++)
    {
     for(k=0;k<NOUTNET;k++)
       {
matriz_prob[i][j].outnet[k]=10;
       }
    }
 }
/* End Initializes matriz_prob */


/* Receives valid GD features */
ChanIn(from_master,matriz_real,sizeof(matriz_real));

i = GDU0; /* resets channel pointer */

/********************/
/* Begin processing */
/********************/

while(fim!=1)
{

 DirectChanIn(from_ln0,vet_info,sizeof(vet_info));
 DirectChanIn(from_ln0,&matriz_dados[vet_info[0]]
                   [vet_info[1]],sizeof(struct _roi_));

 fim = vet_info[2]; /* notifies end of simulation */

 /* Substitutes processed data for valid data to GDW */
 for(j=0;j<NFEAT;j++)
 {
  matriz_dados[vet_info[0]][vet_info[1]].ext_feat[j]=
                                 matriz_real[linha][j];
 }

 /* Run over and over the same valid features,
                       variability == NEVENTO_MAX */
 if(++linha==NEVENTO_MAX) linha=0;

/* Pass data to Global Decision Units */
 if(inicio < OutputSize) /* first time */
 {
  DirectChanOut(Output[inicio],&matriz_dados[vet_info[0]]
               [vet_info[1]].ext_feat[0],NFEAT*sizeof(float));
  for(j=0;j < HEADER_SIZE;j++)
  {
gd[inicio][j]=vet_info[j]; /* Holds header according
                                to destination processor */
  }/* end for j */
```

```
  inicio++;

 } /* end if inicio */

 else /* Other times */
 {
   DirectChanIn(Input[i],&matriz_prob[gd[i][0]][gd[i][1]].
                       outnet[0],4*sizeof(float)); /* Receive data */
   DirectChanOut(Output[i],&matriz_dados[vet_info[0]]
    [vet_info[1]].ext_feat[0],12*sizeof(float)); /* Pass new data */

   /* saves header */
   for(k=0;k<HEADER_SIZE;k++)
   {
    matriz_prob[gd[i][0]][gd[i][1]].header[k]=gd[i][k];
                      /* Saves header into right position */
    gd[i][k]=vet_info[k]; /* Holds header according
                                 to destination processor */
   }/* end saving header */

   /* relocates i, points to other GDW's */
   if(++i == OutputSize) i=0;

 }/* End else */

}/* End WHILE */

/* Receives last data */

for(i=0;i < InputSize;i++)
{

 j = ProcSkipAltList(Input); /* Verifies "InputSize" times
             if there's a GD worker willing to output data */
 if(j != -1)
 {
  DirectChanIn(Input[j],&matriz_prob[gd[j][0]][gd[j][1]].outnet[0]
                             ,4*sizeof(float)); /* Receive data */
  for(k=0;k<HEADER_SIZE;k++)
  {
   matriz_prob[gd[j][0]][gd[j][1]].header[k]=gd[j][k];
                         /* Saves header into right position */
  }/* End for k */
 }/* End if j != -1 */

}/* End for i */

/* End of process, have to output results to master
        application, time counter is ticking there ! */
DirectChanOut(to_master,matriz_prob,sizeof(matriz_prob));

} /* End Main */
```

```
/****************************/
/* Functions called by main */
/* ------------------------ */
/****************************/

/* Creates a vector of channels were the last position
                          is NULL, for implementation reasons */
static Channel **CreateChannels (Channel *Channels[],
                                         int ChannelsSize)
{
    Channel **CopyChannels = NULL;

    if ((CopyChannels = malloc((ChannelsSize + 1)
                          * sizeof(Channel *))) == NULL)
        abort();
   else
    {
        int ChannelsIndex = 0;

        while (ChannelsIndex++ < ChannelsSize)
            CopyChannels[ChannelsIndex - 1] =
                             Channels[ChannelsIndex - 1];

        CopyChannels[ChannelsSize] = NULL;
    }
    return(CopyChannels);
} /* End CreateChannels */
```

F.5 Decisão Global (global.c)

```
#include <stdlib.h>
#include <math.h>
#include <process.h>
#include <misc.h>
#include <channel.h>
#include <time.h>

/* Tipos estruturados globais */

struct _hidd_ /* Estruturas para os
                neurônios da hidden layer */
{
 float threshold;
 float peso[12];
 float saida;
};

struct _outlay_ /* Estruturas para os
                neurônios da camada de saída */
{
```

```
 float threshold;
 float peso[12];
 float saida;
 };



/* Declaração de protótipos das funções utilizadas
   na aplicação */
float ativa(float valor, float *tb, float threshold);
  /* Função que calcula tanh por look-up */
float ativa2(float valor, float *tb, float threshold);
  /* Função que calcula tanh math.h */
static Channel **CreateChannels(Channel *Channels[],
                                     int ChannelsSize);


/*===============================================*/
/* Programa Principal                            */
/*===============================================*/

int main(void)
{
 /* Abertura dos canais de main */
 int InputSize = (int) *((long int *) get_param(2));
 int OutputSize = (int) *((long int *) get_param(4));
 Channel **Input = CreateChannels(get_param(1), InputSize);
 Channel **Output = CreateChannels(get_param(3), OutputSize);

 /* Declaração de variáveis locais a main */

 int i,j,k; /* variáveis de controle */
 float evento[12]; /* O evento corrente */
 float normal[12]; /* O vetor de normalização */
 struct _hidd_ hidd[12]; /* Os neurônios da
                           camada escondida */
 struct _outlay_ outlay[4]; /* Os neurônios da
                              camada de saída */
 float tab[20001]; /* A tabela de valores para tanh */
 float out_vect[4];
 int fim=0; /* Testa fim da rotina */

 /* Carrega tabela com valores das
    tangentes hiperbólicas */
 ChanIn(Input[0],tab,sizeof(tab));

 /* Carrega valores de normalização */
 ChanIn(Input[0],normal,sizeof(normal));

 /* Carrega pesos */
 ChanIn(Input[0],hidd,sizeof(hidd));
 ChanIn(Input[0],outlay,sizeof(outlay));
```

```
while(fim!=1)
{

 /* Lê evento atual */
 DirectChanIn(Input[1],evento,sizeof(evento));

 /* Normaliza o evento */
 for(k=0;k<=11;k++)
 {
  evento[k] /= normal[k];
 } /* for var k */


/* Processamento Neuronal do evento */

/* Processamento dos Perceptrons ou Neurônios
   da camada de entrada */
/* Não existe processamento aqui, pelo menos por enquanto. */


/* Processamento da Hidden Layer ou
   Neurônios da camada escondida */
  for(i=0;i<=5;i++)
  {
   /*variáveis locais a este for (var i) */
   float soma=0;

   for(j=0;j<=11;j++)
   {
    /* Realiza a soma dos produtos internos da saída
                     da camada de entrada com os pesos */
    soma+=(evento[j]*hidd[i].peso[j]);

   }/* for var j (2) */

   /* Passa pela função de ativação */
   hidd[i].saida=ativa(soma, tab, hidd[i].threshold);

  }/* end for var i (2) - Camada escondida */

  /* Processamento da Output Layer ou
      Neurônios da camada de saída */
  for(i=0;i<=3;i++)
  {
   /*variáveis locais a este for (var i) */
   float soma=0;

   for(j=0;j<=5;j++)
   {
    /* Realiza a soma dos produtos internos
      da saída da camada escondida com os pesos */
```

```
    soma+=(hidd[j].saida*outlay[i].peso[j]);

   }/* for var j (3) */

   /* Passa pela função de ativação 'ativa' */
   outlay[i].saida=ativa(soma, tab,
                   outlay[i].threshold);

  }/* for var i (3) */

 /* Coloca saída num vetor */
 for(i=0;i<=3;i++)
 {
 out_vect[i]=outlay[i].saida;
 }

 /* Externaliza saída */
 DirectChanOut(Output[1],out_vect,sizeof(out_vect));

}/* End while , reseta processo */

}/*End Main */

/*================================================*/
/* Funções chamadas pela rotina main             */
/*================================================*/

float ativa(float valor, float *tb, float threshold)

{
 /* Soma threshold do neurônio */
 valor+=threshold;

 /* Testa se valor é negativo */
 if( valor <= 0 ) /* 0 */
 {
  /* Caso seja, retorna o -1 para valor<-10
     ou indexa a matriz tb segundo uma álgebra para
     identificação da binagem do histograma */
  if ( valor <= -10 ) /* 1 */
  {
   return -1.0;
  }/* then 1 */
  else
  {
   /* Acha a parte inteira da expressão à direita,
      em outras, trunca na terceira casa decimal
      (estou multiplicando por 1000) "valor" */
   int indice = 1000*valor+10000;
   return(*(tb+indice));
  }/* else 1 */
```

```
 }/* then 0 */
 else
 {
  /* Caso não, retorna o 1 para valor>10
     ou indexa a matriz tb segundo uma álgebra para
     identificação da binagem do histograma */
  if ( valor > 10 ) /* 2 */
  {
   return 1.0;
  }/* then 2 */
  else
  {
   /* Acha a parte inteira da expressão à
      direita, em outras, trunca na terceira casa decimal
      (estou multiplicando por -1000) "valor" */
   int indice = 1000*valor+10000;
   return(*(tb+indice));
  }/* else 2 */

 }/* else 0 */
}/* Função ativa */

static Channel **CreateChannels (Channel *Channels[],
                                      int ChannelsSize)
{
    Channel **CopyChannels = NULL;

    if ((CopyChannels = malloc((ChannelsSize + 1)]
                          * sizeof(Channel *))) == NULL)
        abort();
   else
    {
        int ChannelsIndex = 0;

        while (ChannelsIndex++ < ChannelsSize)
            CopyChannels[ChannelsIndex - 1] =
                           Channels[ChannelsIndex - 1];

        CopyChannels[ChannelsSize] = NULL;
    }
    return(CopyChannels);
} /* End CreateChannels */
```