UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS

“IMPLEMENTAÇÃO E ESTUDO DA
ARQUITETURA A FLUXO DE DADOS
WOLF”

Marcos Antônio Cavenaghi

Tese Apresentada ao Instituto de Física de São Carlos, da Universidade de São Paulo, para obtenção do Título de **Doutor** em Ciências: Física Aplicada - opção: Física Computacional.

Orientador: Prof. Dr. Álvaro Garcia Neto

DEPARTAMENTO DE FÍSICA E INFORMÁTICA
SÃO CARLOS - 1997

IFSC-USP SERVIÇO DE BIBLIOTECA E
INFORMAÇÃO

Cavenaghi, Marcos Antônio
Implementação e Estudo da Arquitetura a Fluxo de Dados Wolf / Marcos Antônio
Cavenaghi - São Carlos, 1997.
193 p.

Tese (Doutorado) - Instituto de Física de São Carlos, 1997.
Orientador: Prof. Dr. Álvaro Garcia Neto

1. Arquiteturas Paralelas 2. Arquiteturas a Fluxo de Dados 3. Simulação

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE **MARCOS
ANTONIO CAVENAGHI** APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS,
DA UNIVERSIDADE DE SÃO PAULO, EM 25 DE NOVEMBRO DE 1997.

COMISSÃO JULGADORA:

Prof. Dr. Alvaro Garcia Neto/IFSC-USP

Prof. Dr. Marcos José Santana/ICMSC-USP

Prof. Dr. Luis Carlos Trevelin/UFSCar

Profa. Dra. Liria Matsumoto Sato/EP-USP

Prof. Dr. Onofre Trindade Júnior/ICMSC-USP

“Aos meus pais Arlindo e Clarice, à minha irmã Denise e à minha sobrinha Paula pelo incentivo, confiança e amor. E a todas as pessoas que, através de seus exemplos, mostraram-me um pouco do profundo significado do que é a vida e qual seu objetivo.”

IFSC-USP SERVIÇO DE BIBLIOTECA ■
INFORMAÇÃO

“No princípio era o Verbo, e o Verbo estava com Deus, e o Verbo era Deus. Ele estava no princípio com Deus. Todas as coisas foram feitas por ele, e sem ele nada do que foi feito se fez. Nele estava a vida, e a vida era a luz dos homens. E a luz resplandece nas trevas, e as trevas não a compreenderam. Houve um homem enviado de Deus, cujo nome era João. Este veio para testemunho, para que testificasse da luz, para que todos cressem por ele. Não era ele a luz, mas para que testificasse da luz. Ali estava a luz verdadeira, que alumia a todo o homem que vem ao mundo. Estava no mundo, e o mundo foi feito por ele, e o mundo não o conheceu. Veio para o que era seu, e os seus não o receberam. Mas, a todos quantos o receberam, deu-lhes o poder de serem feitos filhos de Deus, aos que creem no seu nome, os quais não nasceram do sangue, nem da vontade da carne, nem da vontade do varão, mas de Deus. E o Verbo se fez carne e habitou entre nós, e vimos a sua glória como a glória do unigênito do Pai, cheio de graça e de verdade.”

João 1, 1-14

"Oh que saudade do luar da minha terra
Lá na serra branqueando
Folhas secas pelo chão;
Este luar cá da cidade tão escuro
Não tem aquela saudade
Do luar do meu sertão."

Catulo da Paixão Cearense/João Pernambuco

Agradecimentos

Ao Prof. Dr. Álvaro Garcia Neto, cuja orientação e amizade muito contribuíram para a elaboração desse trabalho e para minha formação profissional.

Aos professores Ruggiero, Jan e Gonzalo cujos ensinamentos ofereceram-me ótimas oportunidades de aprendizado.

Aos integrantes do grupo *Dataflow* do FFI Patrícia Magna, João Ângelo, João Perea, José Ricardo e Fernando pela amizade e discussões sempre muito ricas em informações.

À Cláudia pela amizade e pelo excelente trabalho de secretaria.

À Leonilda Lopes pela amizade e eficiência na hora de cuidar da burocracia gerada pela Unesp.

Aos professores do DCo pelo ótimo ambiente de trabalho.

Ao CNPq e CAPES pelo apoio financeiro.

E a todas as pessoas que direta ou indiretamente contribuíram na realização deste trabalho.

Conteúdo

Introdução	1
I Implementações a Fluxo de Dados	3
1.1 Introdução	3
1.2 A Máquina a Fluxo de Dados de Manchester	3
1.3 A <i>Mit Tagged Token Dataflow Architecture</i>	5
1.3.1 O Conceito de Elemento de <i>I-Structure</i>	8
1.4 A Máquina Monsoon	9
1.4.1 O Conceito de <i>Explicit Token Store (ETS)</i>	11
1.5 A Máquina *T	13
1.5.1 O Conceito de <i>Multithreading</i>	15
1.6 A Máquina EM-4	16
1.6.1 O Conceito de <i>Macro-Dataflow</i>	18
1.7 Considerações Finais e Síntese do Capítulo	19
1.8 Conclusões	22
II A Arquitetura a Fluxo de Dados Wolf	23
2.1 Introdução	23
2.2 Definição da Arquitetura Wolf*	24
2.3 Funcionamento da Arquitetura Wolf*	24
2.4 Unidades da Arquitetura Wolf Implementadas no Simulador*	28
2.4.1 O Módulo de Entrada e Saída	28
2.4.2 A Chave de Coleta	29
2.4.3 A Memória de Dados	32

2.4.4	A Memória de Instruções Interna	34
2.4.5	A Memória de Instruções Externa	36
2.4.6	A Chave de Distribuição	38
2.4.7	A Fila de Fichas	40
2.4.8	A Unidade Funcional	42
2.4.9	A Memória de Estruturas	45
2.5	Linguagens para Programação Paralela	47
2.5.1	Características de uma Linguagem para Programação Paralela . . .	48
2.5.2	<i>Data Parallel C Extensions - DPCE</i>	50
2.5.3	<i>High Performance Fortran - HPF</i>	50
2.5.4	<i>Streams and Iterations in a Single Assignment Language - SISAL</i> .	51
2.6	O Processo de Compilação para o Simulador	52
2.7	Algoritmos Utilizados	54
2.8	Conclusões	56
III Investigando a Arquitetura Wolf		58
3.1	Introdução	58
3.2	Implementação do Simulador*	58
3.2.1	O Mecanismo de Temporização	59
3.3	Exploração de Seqüencialidade*	59
3.3.1	Execução de Código Seqüencial em Arquiteturas a Fluxo de Dados	60
3.3.2	Mecanismo Simples de Detecção e Execução de Código Seqüencial .	63
3.3.2.1	Resultados da Implementação no Simulador	64
3.4	Experimento de Balanceamento das Unidades do Wolf*	66
3.4.1	Crítérios para Escolha de Tempos e Parada	68
3.4.2	Resultados do Experimento	70
3.4.2.1	Escolha de um dos Resultados	73
3.4.2.2	Validação dos Resultados	74
3.5	O Experimento de Multiprogramação*	77
3.5.1	Investigação do Comportamento dos Tempos do Experimento . . .	82
3.6	Conclusões	85

IV Resultados Obtidos com o Saw	87
4.1 Introdução	87
4.2 Resultados*	87
4.2.1 Comparação entre 5 e 2 Anéis no Saw	89
4.2.2 Ackermann	91
4.2.3 Dinâmica Molecular	92
4.2.4 Somatorial	94
4.2.5 FFT	96
4.2.6 Gauss	97
4.2.7 Interpolação	99
4.2.8 Laplace	101
4.2.9 Multiplicação de Matrizes	103
4.2.10 N-Queens	105
4.2.11 Quicksort	108
4.2.12 Simple	110
4.3 Conclusões	112
V Distribuição de Carga e a Arquitetura Wolf II	113
5.1 Introdução	113
5.2 Problemas de Distribuição de Carga*	113
5.3 Hipóteses para a má Distribuição de Carga*	115
5.3.1 Comportamento Anômalo da Fila de Fichas	115
5.3.2 Comportamento Anômalo da Chave de Coleta	118
5.3.3 Comportamento Anômalo da Temporização	119
5.3.4 Variação na Velocidade da Memória de Instruções Interna	120
5.3.4.1 Análise dos Resultados	123
5.4 Justificativa para a Mudança na Arquitetura Wolf*	124
5.5 A Arquitetura Wolf II*	125
5.5.1 Descrição Funcional	126
5.5.2 O <i>Interleaving</i> da Memória de Dados	128
5.5.3 Eliminação da Memória de Instruções Externa	129

5.5.3.1 Proposta de uma Arquitetura Vetorial para o WOLF II . . 130

5.6 Conclusões 132

Conclusões e Trabalhos Futuros * 134

Bibliografia 139

* Marcam as seções que apresentam as contribuições originais deste trabalho.

Lista de Figuras

1.1	Máquina Fluxo de Dados de Manchester.	4
1.2	A Arquitetura da MTTDA.	6
1.3	A Arquitetura da Monsoon.	10
1.4	O Modelo ETS.	12
1.5	A Máquina *T.	14
1.6	A Máquina FM-4.	17
2.1	A Arquitetura Fluxo de Dados Wolf	25
2.2	O Módulo de Entrada e Saída.	29
2.3	A Chave de Coleta.	30
2.4	A Memória de Dados.	33
2.5	A Memória de Instruções Interna.	35
2.6	A Memória de Instruções Externa.	37
2.7	A Chave de Distribuição.	40
2.8	A Fila de Fichas.	41
2.9	A Unidade Funcional.	43
2.10	A Memória de Estruturas.	46
2.11	Processo de Compilação.	53
2.12	Seqüência de Compilação de um Programa SISAL.	53
3.1	Código Seqüencial.	61
3.2	Ocupação da Memória de Dados para 1 <i>Gauss</i>	64
3.3	Ocupação da Memória de Dados sem Seqüencialidade.	65
3.4	Ocupação da Fila de Fichas sem Seqüencialidade.	66
3.5	Ocupação da Memória de Dados para 5 <i>Gauss</i>	83

3.6	Ocupação da Memória de Dados para 1 <i>Gauss</i>	83
3.7	Ocupação da Fila de Fichas para 5 <i>Gauss</i>	84
3.8	Ocupação da Fila de Fichas para 1 <i>Gauss</i>	85
4.1	Ocupação da Memória de Dados para Ackermann.	91
4.2	Ocupação da Memória de Dados para Dinâmica Molecular.	93
4.3	Ocupação da Memória de Dados para Somatorial.	94
4.4	Ocupação da Fila de Fichas para Somatorial.	95
4.5	Ocupação da Memória de Dados para FFT.	96
4.6	Ocupação da Fila de Fichas para FFT.	97
4.7	Ocupação da Memória de Dados para Gauss.	98
4.8	Ocupação da Fila de Fichas para Gauss.	99
4.9	Ocupação da Memória de Dados para Interpolação.	100
4.10	Ocupação da Fila de Fichas para Interpolação.	100
4.11	Ocupação da Memória de Dados para Laplace.	102
4.12	Ocupação da Fila de Fichas para Laplace.	102
4.13	Ocupação da Memória de Dados para Multiplicação de Matrizes.	104
4.14	Ocupação da Fila de Fichas para Multiplicação de Matrizes.	105
4.15	Ocupação da Memória de Dados para N-Queens.	106
4.16	Ocupação da Fila de Fichas para N-Queens.	107
4.17	Ocupação da Memória de Dados para Quicksort.	108
4.18	Ocupação da Fila de Fichas para Quicksort.	109
4.19	Ocupação da Memória de Dados para Simple.	110
4.20	Ocupação da Fila de Fichas para Simple.	111
5.1	Exemplo de utilização dos anéis.	114
5.2	Arquitetura Wolf com a FF descentralizada.	116
5.3	Distribuição de Carga com Descentralização da FF.	117
5.4	Variação do tempo da MI.	121
5.5	A Arquitetura Wolf II.	126
5.6	A Arquitetura da Memória de Dados.	129
5.7	Arquitetura para o Processador e Memória vetoriais.	131

Lista de Tabelas

3.1	Tráfego de Fichas pela Memória de Dados	62
3.2	Porcentagem de Utilização de cada unidade - parte 1.	71
3.3	Porcentagem de Utilização de cada unidade - parte 2.	71
3.4	Porcentagem de Utilização de cada unidade - parte 3.	72
3.5	Porcentagem de Utilização de cada unidade - parte 4.	72
3.6	Porcentagem de Utilização de cada unidade - parte 5.	73
3.7	Tempos Utilizados para as Unidades do Simulador	74
3.8	Validação do Experimento.	75
3.9	Multiprogramação com <i>Gauss</i>	77
3.10	Multiprogramação com <i>Fatorial</i>	79
3.11	Multiprogramação com <i>Ackermann</i>	80
3.12	Multiprogramação com 5 Programas Diferentes.	80
3.13	Multiprogramação com 4 Programas Diferentes.	81
4.1	Tempo de Execução em uma Arquitetura von Neumann e no Wolf	88
5.1	Distribuição de Carga nos Anéis MI-UF.	118
5.2	Comportamento da CC.	118
5.3	Comportamento da Temporização.	119
5.4	Utilização de cada unidade.	123

Resumo

Esse trabalho apresenta a arquitetura a fluxo de dados Wolf. Essa arquitetura foi proposta considerando-se alguns problemas conhecidos em execução de código em arquiteturas a fluxo de dados tais como tratamento de código seqüencial e tratamento de estruturas de dados (vetores e matrizes).

Wolf é baseado no modelo de fluxo de dados dinâmico e explora granulosidade variável, sendo a mais fina “a nível de” instrução. Alguns conceitos explorados por outras arquiteturas a fluxo de dados guiaram o desenvolvimento do Wolf. Entre esses estão o conceito de “macro-dataflow” e “multithreading”.

Para o estudo da arquitetura Wolf foi desenvolvido um simulador dirigido a tempo que implementa suas características. O simulador Saw foi escrito na linguagem orientada a objetos C++ e pode ser compilado em qualquer plataforma que use um compilador padrão ANSI de 32 *bits*. O código do simulador foi exaustivamente testado e os resultados numéricos dos programas executados estão de acordo com resultados apresentados por uma arquitetura von Neumann padrão.

Após o estudo dos resultados obtidos com as simulações, foram identificados alguns problemas com a arquitetura Wolf. Algumas hipóteses foram testadas para solucionar tais problemas. Os resultados desses testes levaram à alteração da arquitetura. Desta alteração, originou-se a arquitetura Wolf II que será apresentada também nesse trabalho. Por se tratar de uma consequência dos estudos realizados com a arquitetura Wolf, não serão feitos experimentos com o Wolf II.

Abstract

This work presents the dataflow architecture Wolf. Wolf has been proposed in the focus of some known problems identified in previous works: the execution of data structures (vectors and matrices) and sequential code, to name a few.

Wolf is based in the dinamic dataflow model and explores variable granularity (the thinnest is at instruction level). Some concepts developped in the designing of other hybrid architectures, guided the Wolf implementation. The macro-dataflow and multithreading were two of them.

Focusing the study of the Wolf architecture, it has been developped a time driven simulator (Saw). The object oriented language C++ was used for this implementation. The code can be compiled on any ANSI standard 32 bits compiler. This code was exhaustively tested and the numeric results obtained with the experiments were equal to the ones obtained with a von Neumann architecture.

This study identified some problems with the Wolf architecture. Some proposals were implemented in the simulator to try to identify the causes for the problems. The results led to an alteration in the Wolf architecture. The new proposed architecture (Wolf II) is described in the last chapter, but it was not submitted to experiments as Wolf was.

Introdução

Wolf [1, 2, 3, 4, 5, 6, 7] é uma arquitetura baseada no modelo de fluxo de dados dinâmico com granularidade variável. Esta arquitetura foi proposta inicialmente em reuniões do grupo de fluxo de dados do Instituto de Física e Química de São Carlos (IFQSC-USP) composto pelos professores Álvaro, Ruggiero e Catto (Unicamp) e alunos de pós-graduação. Estas reuniões ocorreram até 1992. A proposta inicial era propor uma arquitetura que reuni-se as lições aprendidas com a implementação da Máquina a Fluxo de Dados de Manchester (MFDM), visto que os três professores participaram das pesquisas com a MFDM praticamente durante todo o período em que esta ficou operacional.

Para investigar a arquitetura Wolf, foi desenvolvido um simulador (Saw) que implementa as características da arquitetura em questão. O simulador foi desenvolvido na linguagem orientada a objetos C++. Utilizou-se apenas funções documentadas pelo padrão ANSI sendo possível compilar o código do simulador em diferentes plataformas. No trabalho de mestrado deste mesmo autor [8], foi desenvolvido um simulador simplificado para a arquitetura Wolf. O leitor não familiarizado com a arquitetura Wolf, poderá ter um melhor aproveitamento se iniciar seus estudos pelo trabalho de mestrado.

Esta tese apresenta os resultados obtidos na investigação da arquitetura a fluxo de dados Wolf através do uso do simulador Saw, e para isto foi dividida em cinco capítulos:

O Capítulo 1 apresenta as principais arquiteturas a fluxo de dados desenvolvidas nos principais grupos de pesquisa (*Manchester*, *MIT* e *ETL/MITI*). A discussão sobre estas arquiteturas visa mostrar suas principais características e inovações. No caso das arquiteturas estudadas no *MIT*, serão apresentados os problemas encontrados e as soluções propostas com a implementação de novas arquiteturas. Note-se que todas as arquiteturas apresentadas foram propostas como “arquiteturas de uso geral”, não sendo projetadas para resolver problemas específicos.

O Capítulo 2 apresenta a arquitetura a fluxo de dados Wolf. Aspectos da implementação de cada uma de suas unidades também são discutidas em detalhe. Durante o desenvolvimento do simulador foram propostas várias características à cada unidade da arquitetura. Estas características foram implementadas no simulador e o detalhamento delas encontram-se neste capítulo. São apresentadas também algumas linguagens de programação paralela usadas por várias propostas de arquiteturas paralelas. O capítulo encerra-se com a apresentação do processo de compilação usado para gerar código para o simulador.

O Capítulo 3 apresenta as características implementadas pelo simulador com base nas considerações de cada unidade discutida no capítulo anterior. Mostra também a investigação do simulador desenvolvido, isto é, descreve o ajuste do simulador para obtenção de resultados. Este ajuste consistiu em alguns experimentos que são descritos no decorrer do capítulo. Todos os ajustes feitos no simulador são importantes, visto que esses ajustes permitem a adequação de suas unidades ao comportamento desejado.

O Capítulo 4 discute os resultados obtidos com o simulador. Estes resultados foram apresentados em forma de curvas de ocupação de duas unidades simuladas. O estudo da ocupação destas unidades permite concluir como a arquitetura comporta-se durante o processo de execução de um programa. Algumas características gerais do algoritmo utilizado também podem ser obtidas com base nesses dados. Para situar a arquitetura em termos de desempenho, foi utilizado um experimento que comparou o tempo de execução de um programa em uma arquitetura von Neumann e no simulador.

O Capítulo 5 apresenta o problema de distribuição de carga nos anéis memória-processador verificado com o uso do simulador. São propostas algumas hipóteses para explicar a distribuição ineficiente. Cada hipótese é discutida e testada através da experimentação com o simulador. As conclusões de tais experimentações levaram a uma remodelagem da arquitetura. Com esta remodelagem espera-se que a nova arquitetura seja mais eficiente que a arquitetura Wolf.

Capítulo I

Implementações a Fluxo de Dados

1.1 Introdução

Este capítulo apresenta as mais importantes implementações baseadas no paradigma de fluxo de dados. Para isto aborda os principais conceitos inovadores envolvidos em cada uma destas implementações. A arquitetura Wolf, tema central desta tese, é baseada nas várias contribuições dos diversos grupos de pesquisa em arquiteturas a fluxo de dados apresentadas neste capítulo.

1.2 A Máquina a Fluxo de Dados de Manchester

A Máquina a Fluxo de Dados de Manchester (MFDM) [9] é baseada no paradigma de fluxo de dados dinâmico. Neste modelo, a computação é controlada por dados encapsulados em fichas que circulam em um anel de processamento. A Figura 1.1 apresenta a MFDM cuja estrutura básica é composta por um anel com quatro módulos conectados com uma chave de entrada e saída (*I/O Switch*) responsável pela entrada e saída das fichas do anel. A *Token Queue* é uma fila com a estratégia FIFO (*First In First Out*) e

sua função é controlar o tráfego de fichas para homogeneizar a carga do anel para evitar impasses do tipo (*deadlock*).

Figura 1.1: Máquina Fluxo de Dados de Manchester.

A *Matching Store* (também referenciada como *Matching Unit*) tem a tarefa de emparelhar as fichas que possuem um destino comum e a mesma *cor*¹ implementando a regra de disparo. As fichas destinadas às instruções de uma entrada e os pares de fichas, buscam a instrução correspondente na *Node Store* onde estão armazenadas as instruções do programa que está sendo executado. As fichas contendo as instruções e os dados são então mandadas para as *Functional Units* (FU) onde as instruções contidas nas fichas são executadas.

Algumas lições foram aprendidas durante os dez anos de estudo e a implementação da MFDM a saber:

- A forma circular do pipeline aumenta o tempo de latência e causa flutuações no tráfego das fichas.

¹No modelo a fluxo de dados dinâmico usado pela MFDM, a cor de um dado designa a instancição da chamada a uma função. O termo *cor* será usado neste trabalho designando exatamente isto.

- O número de estágios do pipeline da MFDM é maior que o necessário e atrapalha o desempenho.
- A implementação assíncrona dos estágios causa atrasos no *pipeline* e dificulta a depuração do protótipo.
- O comprimento fixo da palavra (32 bits) não é adequado, sendo muito pequeno para aplicações numéricas e excessivamente grande para aplicações simbólicas.
- A granulosidade das instruções da MFDM aparenta ser mais fina que o desejável.
- A granulosidade fixa da MFDM impossibilita a pesquisa sobre a granulosidade ótima em processadores a fluxo de dados.
- Há redundância em diversos campos das fichas, aumentando de forma desnecessária o espaço de emparelhamento.
- O emparelhamento associativo degrada o desempenho da máquina, aumentando consideravelmente a latência para execução de uma instrução.

A linguagem de programação usada é *SISAL*, uma linguagem funcional desenvolvida pelo *Lawrence Livermore National Laboratory* na década de 1980. Um protótipo de um anel foi construído e esteve operacional durante vários anos, tendo sido desativado em 1991.

1.3 A *Mit Tagged Token Dataflow Architecture*

A *Mit Tagged Token Dataflow Architecture (MTTDA)* [10] (Figura 1.2) consiste de um número de elementos de processamento idênticos e elementos de *I-Structure* (definido

na seção seguinte) também idênticos interconectados por uma rede de pacotes n-cúbica. Um único elemento de processamento e um único elemento de *I-Structure* podem ser considerados como uma máquina a fluxo de dados completa.

Figura 1.2: A Arquitetura da MTTDA.

Um elemento de processamento (PE) tem dois componentes: uma seção de controle e uma seção de computação. A seção de controle recebe fichas especiais que podem manipular qualquer parte do estado de um PE. A seção de computação é o *pipeline* de execução da máquina. Uma ficha de entrada que chega é roteada para um dos dois componentes baseado em seu contexto (*label*). Uma ficha na saída é roteada de volta ao início do PE ou para um elemento de *I-Structure*.

Um acesso específico ao bloco de código é determinado por um contexto que identifica dois registradores: o registrador de bloco de código (*cbr*) que aponta para o endereço base na memória de programa, e o registrador de base de dados (*dbi*) que aponta para o endereço base na Memória Constante (MM).

Quando um bloco de código é ativado, quem chama primeiramente obtém recursos de

um gerenciador de recursos, que aloca um par *cbr/dbr* e espaço na Memória Constante. Depois inicializa o *cbr/dbr* de forma que aponte para o endereço base de instrução e constante; finalmente retorna o número *cbr/dbr* como um contexto. Para executar o bloco de código, fichas são enviadas para a unidade de *matching* do PE.

A *Waiting Matching* é uma memória que contém um conjunto de fichas que estão esperando pela parceira. Se a ficha que entra é destinada para uma instrução monádica², ela passa direto e vai para a unidade de *fetch* de instrução. Se a ficha que entra é destinada para uma instrução diádica³, o conteúdo da memória é examinado para ver se a parceira já chegou. Se a parceira já chegou ocorre o *matching* e o par de fichas é enviado para a unidade de *fetch* de instrução. Se a memória não contém o par, a ficha é deixada na memória.

O *tag* nas fichas que entram na unidade de *fetch* de instrução identifica uma instrução a ser lida da memória de instruções. O *tag* também especifica um *dbi*, que usando um endereço de base constante e um *offset*, busca qualquer constante da Memória de Constantes. Todos os dados necessários para execução de uma instrução são agora passados para a próxima unidade.

A unidade lógica aritmética (ALU) e a unidade de cálculo de *tag* (*Form Token*) são duas unidades que operam em paralelo. A ULA usa os operandos, literais, constantes e *opcodes* e produz um resultado. A unidade de cálculo de *tag* usa o *cbr* e *dbi* do contexto e os *offsets* do destino e calcula os novos *tags* para a saída. O resultado do cálculo junto com o novo *tag* são enviados para a unidade montadora de fichas. Nesta unidade os resultados de operações, os destinos e os novos *tags* são combinados numa ficha resultante que retorna ao início do *pipeline*.

²Uma instrução monádica necessita de apenas um operando em sua entrada

³Uma instrução diádica necessita de dois operandos em suas entradas

A linguagem de programação da MTTDA é *Id*, uma linguagem específica para arquiteturas a fluxo de dados desenvolvida pelo grupo do MIT na década de 1980 [11]. A arquitetura MTTDA foi implementada em um simulador em 1987 e largamente explorada. As lições aprendidas com sua implementação e estudo deram origem a uma nova arquitetura (*Monsoon* - descrita na seção seguinte), incorporando conceitos novos ao modelo de fluxo de dados. Todos os resultados obtidos com esta arquitetura foram através do simulador. A Implementação do *hardware* nunca foi descrita na literatura. O artigo que introduz esta arquitetura foi publicado no início de 1990. Não foram publicados resultados que mostrassem os pontos problemáticos da arquitetura.

1.3.1 O Conceito de Elemento de *I-Structure*

Um elemento de *I-Structure* (IS) [12] é um módulo de memória com um controlador que gerencia pedidos de leitura e escrita de estruturas de dados como vetores e matrizes. Na área de armazenamento de dados, cada posição tem *bits* de presença extras que especificam estados: **Presente**, **Ausente** ou **Esperando**. Quando uma IS é alocada, todas as suas posições são colocadas no estado **Ausente**.

Quando uma ficha de leitura chega a um elemento de IS, ela contém o endereço da posição a ser lida e o *tag* da instrução que está esperando o valor. Se esta posição é **Presente**, o dado é lido e enviado para a respectiva instrução, não alterando o estado. Se o estado é **Ausente** ou **Esperando**, a leitura é deferida, isto é o dado é “enfileirado” naquela posição. A fila formada através deste processo é uma lista encadeada de *tags*.

Quando uma ficha de escrita chega, esta contém o endereço da posição a ser escrita e o respectivo dado. Se o estado da posição é **Ausente**, o dado é escrito e o estado

mudado para **Presente**. Se o estado é **Esperando**, o dado é escrito e o estado mudado para **Presente**, sendo tal dado enviado também para todos os endereços “enfileirados” na dada posição. Se o estado da posição já for **Presente**, um erro é gerado.

1.4 A Máquina Monsoon

A máquina Monsoon [13] (Figura 1.3) é um processador de uso geral que incorpora o mecanismo de *Explicit Token Store* (ETS) (definido na seção seguinte), eliminando o *matching* associativo tão dispendioso para o modelo a fluxo de dados. Originada da evolução da MTTDA, é composta por elementos processadores em *pipeline* conectados por uma rede multi-estágio em módulos de memória que possuem o mecanismo de armazenamento de IS. As mensagens entre processadores são passadas através de fichas.

O mapeamento de *frames* de ativação (área para armazenamento de um bloco de código) é feito da seguinte forma: um *frame* necessário à execução de um procedimento é dinamicamente alocado em um elemento de processamento e executado totalmente neste elemento. Iterações de um *loop* são distribuídas para múltiplos elementos de processamento, atribuindo diferentes *frames* de ativação para cada iteração concorrente. Esta estratégia melhora a localidade sem perder a granulosidade fina. Não é usado algoritmo de balanceamento de carga, isto é processos alocados em um elemento de processamento são terminados naquele elemento não sendo passados para outro. A máquina possui duas filas de fichas distribuídas entre os elementos de processamento (*System Queue* e *User Queue*).

Cada elemento de processamento é um processador com um *pipeline* de oito estágios. Em cada ciclo do processador, uma ficha é posta no início do *pipeline* e no final de

Figura 1.3: A Arquitetura da Monsoon.

oito ciclos, seus resultados (nenhuma, uma ou duas fichas) são postos na saída do mesmo. Dentro do *pipeline* ocorre o *fetch* de uma instrução na memória de instruções. Um endereço efetivo dentro da área de armazenamento de *frames* é calculado e uma leitura/escrita é feita nos *bits* de presença da posição calculada. Em seguida ocorre uma leitura ou escrita para o dado armazenado na posição.

O dado da ficha de entrada e qualquer operando lido do *frame* é processado por uma ULA de ponto flutuante de três estágios. Paralelamente a este processamento, dois *tags* novos são calculados. As fichas produzidas podem voltar ao início do *pipeline*, ficarem armazenadas em uma das filas de fichas do elemento processador, ou serem enviadas para a rede de intercomunicação que se encarregará de distribuí-las para o elemento de processamento correto.

A linguagem de programação é *Id* (a mesma linguagem usada para programar a MTTDA). Resultados publicados mostram[14] que foram obtidos *speedups* maiores que 7 para um conjunto de 8 processadores. Proposta através de um trabalho de doutoramento

no MIT em 1988 [13], a máquina Monsoon contribuiu para o aprimoramento dos conceitos de fluxo de dados, proporcionando o aparecimento de novas idéias. Um conceito fundamental nos dias de hoje para arquiteturas a fluxo de dados e cujo embrião surgiu dos estudos da Monsoon é o *multithreading*. Neste sentido, pode-se dizer que a máquina Monsoon originou algumas das idéias fundamentais para implementação de uma nova arquitetura a fluxo de dados chamada *T.

1.4.1 O Conceito de *Explicit Token Store (ETS)*

O Explicit Token Store (ETS) [15] (Figura 1.4) é um modelo simplificado de execução de fluxo de dados dinâmico centrado na noção de *frame* de ativação. Este modelo permite armazenamento explícito de operandos e *bits* de presença baseado na associação do *tag* (contexto e instrução destino) com posições físicas nos quadros de memórias locais aos elementos de processamento.

Os *bits* de presença são usados para guiar o processamento de instruções. Este sistema elimina o *matching* associativo: a alocação de armazenamento implícito presente em modelos de fluxo de dados com *tags*. O compilador determina o uso de *slots* dentro de um *frame* e instala operações de alocação e desalocação em *frames* nos grafos.

Um programa é composto de blocos de código (funções, *loops*, etc). O escalonamento de um destes blocos para execução, dispara a alocação dinâmica de um *frame* de ativação, provendo armazenamento local para tal ativação. Os arcos no bloco de código (variáveis locais para a função) são estaticamente mapeados nos *slots* dos *frames* colorindo o grafo. Cada instrução especifica a posição de seus operandos como um simples cálculo de endereço.

Figura 1.4: O Modelo ETS.

Neste modelo, uma ficha possui um dado, um ponteiro para a instrução a ser executada (IP) e um ponteiro para um *frame* de ativação (FP). Os dois últimos formam o *tag*. A instrução buscada da posição IP especifica o *opcode*, o *offset* no *frame* de ativação onde ocorrerá o *matching* e uma ou mais instruções de destino, que receberão o resultado da operação.

A característica principal dos *frames* de ativação é que cada *slot* de *frame* possui *bits* de presença associados que especificam a disposição de cada *slot*. A regra de disparo é realizada por uma transição de estado nestes *bits* de presença. Se quando chegar uma ficha o *slot* estiver vazio, o valor contido na ficha é depositado no *slot* e nenhum outro processamento da instrução é feito. Se ao chegar uma ficha o *slot* estiver cheio, o valor é extraído do *slot* e a instrução correspondente é executada produzindo uma ou mais fichas como resultado.

A execução dos programas geralmente produz uma árvore de *frames* de ativação, pois um procedimento pode gerar chamadas paralelas, onde o processo pai e o processo filho

executam concorrentemente. O processo filho, por sua vez pode gerar chamadas paralelas e assim por diante, produzindo a árvore de *frames* de ativação.

O sistema de armazenamento em uma máquina ETS é constituído por um vetor de posições. Cada posição contém um tamanho fixo e *bits* de presença para representar o estado da posição. Cada ficha possui um ponteiro para a instrução, um ponteiro para um *frame* de ativação e um único dado. Duas chamadas concorrentes para um mesmo procedimento, compartilharão um mesmo código, mas usarão *frames* de ativação distintos para armazenagem.

1.5 A Máquina *T

A arquitetura da *T (StarT) originou-se basicamente de estudos realizados com a máquina Monsoon e foi introduzida através de um artigo de 1992 [16]. A Figura 1.5 mostra a arquitetura da máquina *T. A arquitetura é composta por três processadores assíncronos. Cada um destes processadores pode ser visto como um processador RISC convencional com apenas algumas extensões.

O *Data Processor* (DP) executa *threads* (definido na seção seguinte) obtidos através da *Continuation Queue* (CQ) e é otimizado para execuções sequenciais. O contador de programa é chamado de *Data Processor Instruction Pointer* ou **dIP**. O registrador **dFP** é por convenção, o registrador que armazena o *frame pointer* do *thread* em execução. Quando um *thread* termina de ser executado e não há nenhum outro *thread* na **CQ**, o **DP** espera até que um fique disponível.

O *Synchronization Coprocessor* (SC) é responsável pelo recebimento de respostas a pedidos de leitura remotos. Possui um contador de programas (**sIP**) e registradores de

Figura 1.5: A Máquina *T.

uso geral. O **SC** é ativado através da chegada de uma mensagem na *message queue*. Quando uma mensagem chega, todos os registradores do processador são carregados com os respectivos valores e então é iniciada a execução da instrução a partir de **sIP**. Uma operação de leitura de memória remota é concluída armazenando o valor da mensagem que está em **sV1**, na posição de destino. Após esta operação, uma *continuation* (definida na seção seguinte) referente ao *thread* suspenso é colocada na **CQ**.

O *Remote Memory Request Coprocessor* (**RMem**) é responsável pelo atendimento à pedidos de leitura à memória local iniciados por nós remotos. O **RMem** retira da *message queue* um pedido de leitura local, inicia um ciclo de leitura à memória local e envia o dado lido em forma de uma mensagem do tipo $\langle IP, Endere\c{c}o, Valor1, Valor2, \dots \rangle$ de volta à rede.

A máquina *T é programada usando-se a linguagem *Id*. Um protótipo em *hardware* foi implementado. Os estudos realizados com esta arquitetura mostraram que existe uma

latência grande na suspensão da execução de *threads* devido á inevitável dependência de dados existente entre eles. Não foram apresentados resultados obtidos com a máquina, apenas artigo com conclusões. Para diminuir esta latência, foi proposta uma nova arquitetura chamada *T-NG [17, 18]. Esta arquitetura usa mecanismos de coerência de *cache* para a memória global compartilhada. A *T-NG não será abordada por este trabalho, já que não possui nenhum mecanismo especial cujo conceito será usado neste trabalho.

1.5.1 O Conceito de *Multithreading*

Um *thread* [19] é uma seqüência de instruções estaticamente ordenadas pelo compilador. Quando um *thread* é escalonado para execução, este não é interrompido até que todos os seus nós sejam executados. Um *thread* define uma unidade básica de trabalho consistente com o modelo de fluxo de dados. O conceito de *thread* combina chaveamento de contextos “a nível de” instruções com escalonamento seqüencial.

Como um *thread* define a unidade básica de escalonamento de um programa, a granulosidade da computação é definida pelo tamanho do *thread* utilizado. Cada *thread* possui um custo de processamento associado, afetando diretamente a quantidade de *overhead* necessária à sincronização. Porém, o alvo principal no particionamento de um programa em *thread*, é a maximização do paralelismo e a minimização do *overhead*.

Existem várias maneiras de se obter um *thread* quando do particionamento de um programa. Pode-se obter *threads* subdividindo um *loop* paralelo em um número de processos seqüenciais. Como consequência, a granulosidade de cada *thread* tende a ser grossa, limitando a quantidade de paralelismo que pode ser explorada. Por outro lado, as linguagens que não são funcionais, tornam o particionamento não trivial devido às dependências de

dados que são resolvidas apenas dinamicamente.

Iannucci [20] propõe um método de particionamento para maximizar o paralelismo explorável, isto é, instruções que são agrupadas em *threads* devem fazer parte de um código onde o paralelismo explorável é baixo ou de preferência nenhum. Qualquer arco que ultrapasse os limites de um *thread* implica em sincronização dinâmica introduzindo *overhead* e aumentando os ciclos de processamento.

Todo o particionamento do programa em *threads* é executado em tempo de compilação, tornando difícil prever resultados de dependências dinâmicas. Desta forma, as técnicas para particionamento de programas em *threads* não abrangem todos os casos. Isto provoca um aumento no tempo de execução de um *thread*, aumentando o *overhead* introduzido devido às dependências de dados causadas por arcos dinâmicos.

Uma tentativa de minimizar este *overhead* é multiplexar o escalonamento entre vários *threads*. Este é o conceito de *multithreading*. Neste sistema, quando um *thread* necessitar de um dado armazenado remotamente, o processador suspende a execução do *thread* em questão e inicia a execução de um outro *thread*. O custo do chaveamento entre os *threads* é inferior à latência de uma leitura a um dado armazenado remotamente.

Uma consequência deste mecanismo é o surgimento do conceito de *continuation*. Um *continuation* é um identificador que as mensagens de leitura e resposta de leitura carregam para que o respectivo *thread* possa ser reiniciado adequadamente.

1.6 A Máquina EM-4

O EM-4 [21] é uma máquina com granulosidade variável empregando o conceito de arcos fortemente conectados (definido na seção seguinte) e formada por vários elementos de

proceamento denominados **EMC-R** que adotam a filosofia RISC. A Figura 1.6 mostra a arquitetura da máquina EM-4.

Figura 1.6: A Máquina EM-4.

A estrutura lógica da interconexão entre elementos de proceamento é uma rede ômega de multi-estágios. Esta rede ômega pode ser dividida em grupos que consistem de um mesmo número de elementos de proceamento e de estágios (módulos horizontais). A execução de uma instância de uma função é mandada para um grupo quando a função é escalonada para execução. Cada EMC-R consiste de:

1. **Unidade de Chave (SU):** Conecta o processador à rede. A SU implementa balanceamento de carga “a nível de” função. Este balanceamento é implementado através de pacotes especiais que percorrem a rede monitorando a carga de cada elemento de proceamento.
2. **Unidade de *Buffer* de Entrada (IBU):** A IBU é uma FIFO implementada com uma memória. Quando o *buffer* está cheio, uma parte de memória fora do *chip* é

usada como *buffer* secundário.

3. **Unidade de *Fetch* e Emparelhamento (FMU):** A FMU realiza a operação de emparelhamento e busca de instruções.
4. **Unidade de Execução (EXU):** Responsável pela execução de instruções sobre os dados, ambos trazidos pela FMU.
5. **Unidade de Controle de Memória (MCU):** A MCU arbitra sobre as requisições de acesso à memória pela IBU, FMU e EXU, enviando então dados entre a área de memória fora do *chip* e o EMC-R.
6. **Controlador de Manutenção (MAINT)** O MAINT opera principalmente para iniciar o sistema, monitorar a execução de programas e *debug* do *hardware* e *software*.

Um protótipo com 80 elementos de processamento está operacional desde 1990 no *Electrotechnical Laboratory* (ETL/MITI) no Japão. Não foram publicados resultados, apenas artigos com análises e conclusões. A linguagem usada para programação da máquina EM-4 é *DFCH* [22], uma versão de C com características de atribuição única e sem o conceito de ponteiros. As pesquisas realizadas com o EM-4 originaram ainda outras arquiteturas (EM-X e EM-5) [23]. Estas arquiteturas não serão abordadas por este trabalho, já que não possuem nenhum mecanismo especial cujo conceito será usado neste trabalho.

1.6.1 O Conceito de *Macro-Dataflow*

O modelo *macro-dataflow* é baseado no conceito de arcos fortemente conectados. Esta seção apresenta uma discussão sobre a implicação deste conceito no modelo de fluxo de dados, não nos interessando qualquer definição ou prova matemática sobre o assunto.

Detalhes sobre a definição e prova matemática podem ser encontrados em [24]. Um subgrafo fluxo de dados cujos nós são conectados por arcos fortemente conectados é chamado de bloco fortemente conectado.

Existe uma dependência de dados marcante entre cada nó de um bloco fortemente conectado, permitindo assim que o bloco seja tratado como um *macro-nó* onde cada nó (instrução) deste macro-nó é executado um após o outro ininterruptamente no elemento de processamento que implementa o modelo.

Desta forma, pode-se dizer que o modelo *macro-dataflow* é um modelo que permite granulosidade variável, sendo a mais fina “a nível de” instrução. Cada bloco fortemente conectado é habilitado para execução através da chegada de operandos através de arcos normais. Um arco normal representa a dependência de dados existente entre dois blocos fortemente conectados.

O elemento de processamento é composto pela integração de um *pipeline* circular baseado no modelo de fluxo de dados e um *advanced control pipeline* que implementa um fluxo de controle para execução de cada macro-nó. Dados locais a um macro-nó são armazenados em registradores locais, diminuindo o *overhead* de comunicação entre cada elemento. Como só existe uma ativação de um macro-nó sendo executada por vez, o emparelhamento de operandos é simplificado pela diminuição da chave de emparelhamento.

1.7 Considerações Finais e Síntese do Capítulo

Um dos problemas ainda encontrados em toda máquina a fluxo de dados, é a linguagem de programação usada. Existem várias propostas de linguagens para programação paralela. Estas propostas podem ser divididas em duas categorias: na primeira, o pro-

gramador deve conhecer características arquiteturais da máquina que está trabalhando para poder explorar adequadamente todos os seus recursos de paralelismo. Na segunda, o programador deixa ao compilador a tarefa de paralelizar seu código.

Problemas comuns existem nas duas categorias de linguagens. O principal está no fato destas linguagens possuírem características especiais que as diferem das linguagens imperativas usadas em máquinas *von Neumann*. Estas características tornam difícil a tarefa de migrar pacotes de software já prontos para estas linguagens, além de dificultar a popularização pela “dificuldade” na forma de programação. Uma possibilidade, apesar de ser apenas especulativa e não ter sido estudada por este trabalho, seria o uso de pré-compiladores para mapear pacotes em linguagem convencional para linguagem funcional e então, usar compiladores para mapear de linguagem funcional para uma linguagem com características de linguagem paralela.

Diversas propostas e implementações a fluxo de dados foram realizadas desde meados da década de 1970. A arquitetura da MFDm foi uma das primeiras a ser estudadas. As lições aprendidas com a implementação desta arquitetura e da arquitetura da MTTDA, nortearam o desenvolvimento da arquitetura da Monsoon no MIT. A Monsoon eliminou o emparelhamento associativo criando o conceito de *Explicit Token Store* ou “armazenamento explícito de fichas” (armazenamento virtual).

Para implementar as inovações propostas pelo modelo ETS, foi criado o conceito de *thread*. Conceitualmente, um *thread* é um bloco de código cuja dependência de dados entre as instruções que o compõe é muito estreita. O escalonamento de um *thread* para execução é feito assim que a execução do *thread* anterior termine no processador. Este esquema dificulta a quebra de um programa em *threads*, devido principalmente às dependências de dados dinâmicas.

Devido à dificuldade de prever arcos dinâmicos e ao mapeamento estático de *threads* nos elementos de processamento, surgiu a idéia de suspender um *thread* quando este necessita de dados não armazenados localmente. Quando isto ocorre, outro *thread* é escalonado para execução evitando que o elemento de processamento fique ocioso enquanto espera pela chegada do dado. Este é o conceito de *multithread* e a arquitetura *T implementa este conceito.

No modelo de *multithread*, o particionamento do grafo em *threads* é feito pelo compilador. O critério utilizado para este particionamento torna crítica a execução de um programa, visto que se o particionamento não for adequado, a latência gerada pela dependência de dados entre cada *thread* pode degradar a execução do programa. Uma maneira de minimizar esta dependência é utilizar o conceito de “bloco fortemente conectado”. Desta forma um grafo é particionado em blocos fortemente conectados, minimizando a dependência entre cada bloco. Este conceito foi desenvolvido no Japão e é conhecido como *macro-dataflow*.

Os diversos grupos de pesquisa apresentaram poucos resultados práticos sobre as arquiteturas a fluxo de dados estudadas. Geralmente os artigos sobre uma determinada arquitetura apenas apresentam esta arquitetura e discorrem sobre suas principais inovações e propostas para resolver problemas até então conhecidos. Artigos que apresentam resultados de *speedup* e ocupação de recursos do sistema estudado são muito raros. A preocupação comum das principais implementações a fluxo de dados foi ser capaz de resolver problemas genéricos. Nenhuma das arquiteturas apresentadas neste capítulo foi desenvolvida com a finalidade de ser dedicada à resolução de um determinado problema ou conjunto de problemas.

1.8 Conclusões

A exploração comercial em larga escala de máquinas baseadas em arquiteturas a fluxo de dados ainda não é uma realidade. À medida que a demanda por máquinas mais rápidas e com uma capacidade maior de memória surgir, arquiteturas a fluxo de dados serão novamente alvo de intensas pesquisas. Uma análise comparativa dos recursos utilizados por *softwares* comerciais nos últimos dez anos mostra que a demanda por memória, espaço para armazenamento e velocidade de processamento cresceu bruscamente. Num futuro próximo (final da década de 1990 e início da década de 2000), o uso computacional envolvendo multimídia em redes de alta velocidade continuará a exigir processadores cada vez mais velozes.

Os protocolos utilizados hoje para tratamento deste tipo de dado geram muitos pacotes que, além da alta velocidade para transmissão, necessitam também de um tratamento rápido. Tendo em vista a natureza do modelo de fluxo de dados e as pesquisas em redes de alta velocidade, máquinas baseadas neste modelo podem ser boas soluções para os tipos de aplicações que envolvam multimídia, tanto na parte de tratamento dos pacotes quanto na transmissão dos mesmos.

Capítulo II

A Arquitetura a Fluxo de Dados Wolf

2.1 Introdução

Este capítulo apresenta a arquitetura a fluxo de dados Wolf e discute aspectos gerais da implementação de cada unidade (o termo **unidade** será usado para designar qualquer parte da arquitetura, não necessariamente referindo-se à Unidade Funcional, que representa apenas uma das várias unidades que compõem a arquitetura Wolf) que a compõe. As unidades descritas neste capítulo foram implementadas usando a linguagem orientada a objetos C++ e o conjunto destas unidades denominaremos de Simulador da Arquitetura Wolf (Saw).

Apresenta também três linguagens de alto nível elaboradas com o intuito de facilitar a programação de computadores paralelos: *Data Parallel C Extensions* (DPCE), *High Performance FORTRAN* (HPF) e *Streams and Iterations in a Single Assignment Language* (SISAL). As duas primeiras linguagens (DPCE e HPF) são apresentadas apenas a título ilustrativo, já que não é objetivo deste trabalho fazer um tratado sobre linguagens de programação paralela. A linguagem SISAL é apresentada com um pouco mais de detalhes já que esta linguagem será utilizada como linguagem de alto nível para os programas a

serem compilados para os testes que serão realizados.

Finalmente, o processo de compilação usado para gerar os programas utilizados pelo simulador para os testes são apresentados. Neste processo, um programa SISAL é compilado num grafo a fluxo de dados. Este grafo é carregado diretamente no simulador para investigação do comportamento da arquitetura.

2.2 Definição da Arquitetura Wolf

A arquitetura a fluxo de dados Wolf é baseada no modelo de fluxo de dados dinâmico com granulosidade variável [25]. A Figura 2.1 apresenta sua estrutura. A arquitetura Wolf visa resolver alguns problemas encontrados em arquiteturas baseadas no paradigma de fluxo de dados tais como:

- Ineficiência no tratamento de código intrinsecamente seqüencial
- Tratamento de estruturas de dados tais como vetores e matrizes

Estes e outros problemas foram identificados nas pesquisas anteriores de fluxo de dados realizada nos diversos grupos de pesquisa, tal como discutido anteriormente neste trabalho.

2.3 Funcionamento da Arquitetura Wolf

O funcionamento de uma arquitetura a fluxo de dados é controlado por fichas. Uma ficha representa um dado no arco do grafo de dependência sob interpretação. Fichas são inseridas na máquina pela **entrada** na Chave de Coleta. Da Chave de Coleta, as fichas

passam para a Memória de Dados se esta estiver disponível, ou são armazenadas na Fila de Fichas para distribuição posterior.

Figura 2.1: A Arquitetura Fluxo de Dados Wolf

A Memória de Dados gera um pacote que é um par de fichas de mesma cor dirigidas para um mesmo nó. Tal pacote é então enviado para a Memória de Instruções Externa. Nesta unidade está armazenada uma cópia do programa que está sendo executado. A finalidade desta unidade é fornecer *opcodes* para dados destinados à operações de tratamento de estruturas de dados.

Quando um dado destinado a uma operação de tratamento de vetores, criação de um novo processo ou instrução *proliferate*¹ chega à esta unidade, a Memória de Instruções

¹Uma instrução *proliferate* gera um *stream* de fichas dependendo do valor de um de seus operandos. É usada para tratamento de vetores. No conjunto de instruções usados pelo simulador, existem três instruções deste tipo (NTUP, LPRL, LPOF). Estas instruções são importantes porque podem ficar ativas

Externa constrói uma ficha contendo o par de fichas, o *opcode* e o(s) endereço(s) de destino do(s) dado(s) que será(ão) produzido(s) e envia esta ficha para a Chave de Distribuição. Caso o dado que chega não for destinado a uma operação de tratamento de vetores, criação de um novo processo ou *proliferate*, esta unidade envia o pacote inalterado para a Chave de Distribuição.

O pacote que chega na Chave de Distribuição pode ter três destinos diferentes: para a Memória de Estruturas, para a Unidade de Controle de Paralelismo ou para a Memória de Instruções Interna. A Chave de Distribuição envia o pacote para a unidade respectiva. No caso de pacotes destinados à Memória de Instruções Interna, estes são enviados para a primeira unidade livre².

Na Memória de Instruções Interna está armazenada uma cópia do programa que está sendo executado. Esta unidade cria a partir do pacote que chega, um pacote executável com todas as informações necessárias para a execução da instrução enviando-o à sua Unidade Funcional associada. Os resultados produzidos pela unidade funcional podem ser do tipo **livre** ou **encadeado**. Resultados livres são passados para a Chave de Coleta, recirculando no *pipeline*. Resultados encadeados permanecem nos registradores internos da Unidade Funcional para serem, num estágio posterior, usados como operandos de instruções subseqüentes. O conjunto de unidades formado pela Memória de Instruções Interna e sua Unidade Funcional (também denominado Processador) correspondente é chamado de anel MI-UF.

Os nós armazenados pela Memória de Instruções (quando não especificado, este termo representa a Memória de Instruções Interna e a Memória de Instruções Externa, sem

numa Unidade Funcional por um período de tempo muito maior que as outras instruções, geralmente produzindo muitas fichas. Isto geralmente afeta negativamente o funcionamento e o equilíbrio de qualquer multicomputador.

²Toda unidade é considerada livre se não existir fichas em sua entrada.

distinção entre elas) têm a informação de que um dado nó é parte de uma cadeia. Quando um pacote executável de uma cadeia é produzido, a Memória de Instruções inicia, por conta própria, um ciclo de busca usando o endereço de destino do pacote executável produzido como endereço de busca.

Desta forma, quando a Unidade Funcional termina o consumo do pacote executável anterior, um novo pacote executável está em condição de ser produzido pela Memória de Instruções. Este novo pacote executável usará os resultados intermediários do pacote executável anterior como entrada (resultados que estão armazenados nos registradores da Unidade Funcional correspondente). O Capítulo 3 detalha este mecanismo.

Uma ou mais unidades de Memória de Estruturas [26] estão conectadas entre as chaves de Distribuição e Coleta. A Memória de Estruturas é responsável pelo armazenamento de dados estruturados (vetores e matrizes). Esta unidade aloca espaço sob demanda e possui um mecanismo de contagem de referências que permite a liberação do espaço alocado automaticamente. As operações de leitura e escrita podem ser executadas em qualquer ordem, pois a Memória de Estruturas consegue armazenar internamente os pedidos de leitura para os dados ainda não produzidos (*I-Structure* [12]).

A Unidade de Controle de Paralelismo [27] controla a quantidade de processos ativos na máquina. É esta unidade que gerencia a utilização de “nomes de ativação” implementando o *recycling*. Esta unidade foi parcialmente implementada no simulador mas não foi utilizada para obtenção dos resultados desta tese, visto que não foram implementadas as rotinas que controlam a terminação de processos.

2.4 Unidades da Arquitetura Wolf Implementadas no Simulador

O Simulador da Arquitetura Wolf (Saw) foi construído para permitir a investigação do comportamento da arquitetura Wolf. Esta seção apresenta uma descrição, dividida em módulos funcionais, de cada unidade implementada no Saw. A especificação do funcionamento destas unidades e sua implementação no simulador foram estenciais para a validação e para o amadurecimento da arquitetura Wolf. Esta especificação funcional é uma das contribuições originais desta tese.

Cada um dos módulos descritos nesta seção é uma rotina em C++ implementada no simulador Saw.

2.4.1 O Módulo de Entrada e Saída

O Módulo de Entrada e Saída (MES) é responsável pela carga e decodificação do programa que se deseja executar, além de servir como interface para saída de dados. Inicialmente o programa a ser executado está armazenado em três arquivos no computador hospedeiro (*host*) na linguagem usada pela Máquina a Fluxo de Dados de Manchester (MFDM) [9].

Estes arquivos são enviados pelo *host* para o módulo quando deseja-se iniciar a execução do programa. Além da carga do programa, este módulo é responsável pela saída dos resultados produzidos. Cada dado produzido é desta forma enviado para o computador hospedeiro onde pode ser utilizado pelo usuário.

O Módulo de Entrada e Saída possui duas sub-unidades: Unidade de Decodificação

Figura 2.2: O Módulo de Entrada e Saída.

de Programa (UDP) e Unidade de Saída (US) como mostra a Figura 2.2.

UDP: A UDP recebe a ficha de programa enviada pelo *host* no formato usado pela MFDM e decodifica-a para o formato usado pelo Wolf. Após esta decodificação, a ficha resultante é enviada para a Chave de Coleta (CC), que será descrita na seção 2.4.2.

US: A US recebe através da CC as fichas referentes aos resultados produzidos pela execução do programa enviando-as para o *host*. No *host* estas fichas são armazenadas em arquivos.

2.4.2 A Chave de Coleta

A Chave de Coleta (CC) recebe fichas oriundas do MES (seção 2.4.1), das Unidades Funcionais (seção 2.4.8), Fila de Fichas (seção 2.4.7), da Memória de Estruturas (seção 2.4.9) e da Unidade de Controle de Paralelismo (seção 2.3) e pode enviá-las para o

MES, Fila de Fichas ou Memória de Dados (seção 2.4.3) conforme o caso.

A CC usa o seguinte critério para enviar fichas às unidades conectadas às suas saídas: quando uma ficha chega em uma das entradas, a CC armazena esta ficha em um *buffer* interno. Cada entrada possui uma única posição neste *buffer*. Isto é feito para evitar que tal *buffer* se comporte como uma Fila de Fichas. A CC ordena as fichas no *buffer* de forma que as fichas produzidas primeiro têm prioridade para serem enviadas à saída.

Figura 2.3: A Chave de Coleta.

A Figura 2.3 mostra a comunicação entre as unidades que compõe a CC. A funcionalidade de cada unidade é:

Unidade de Decodificação de Fichas - UDecF: Esta unidade recebe as fichas das unidades conectadas às entradas da CC. Cada ficha recebida é dividida em duas partes. A primeira parte consiste do campo que designa o momento em que a ficha foi criada e a identificação de qual das entradas a ficha é oriunda. Esta parte é enviada para a Unidade de Controle e Ordenação. A segunda parte é composta pelo restante das informações da ficha e é enviada para a Unidade de Armazenamento.

Unidade de Controle e Ordenação - UCO: Esta unidade é responsável pelo controle e ordenação das fichas armazenadas na Unidade de Armazenamento (UA). A UCO possui uma tabela de endereços da Unidade de Armazenamento que indica todas as posições ocupadas pelas fichas incluindo a informação referente ao momento de criação da ficha e de qual das entradas a ficha se origina. Desta forma, quando uma nova ficha é enviada para a UA, a UCO envia uma mensagem para esta unidade contendo o endereço de armazenamento da nova ficha. A UCO pode também receber informações da Unidade de Distribuição de Fichas referentes a pedidos de novas fichas para as saídas. Se um pedido deste tipo chega até à UCO, esta envia à UA uma mensagem para que esta libere uma ficha. Esta mensagem é composta do endereço da primeira ficha na lista. Após o envio desta mensagem, a UCO reorganiza a tabela de forma que a posição de cada ficha seja atualizada. Após a atualização da tabela, a UCO envia à UDecF uma mensagem contendo a informação de qual das entradas a ficha se origina e ao chegar à UDecF esta libera a entrada correspondente.

Unidade de Armazenamento - UA: Esta unidade é responsável pelo armazenamento das fichas que chegam às entradas da CC. Para cada entrada, a UA armazena a ficha correspondente. A ficha liberada pela UA vai para a Unidade de Distribuição de Fichas.

Unidade de Distribuição de Fichas - UDisF: A UDisF identifica para qual das saídas da CC a ficha se destina: MES ou Memória de Dados (descrita na seção 2.4.3). Antes de enviar a ficha para a saída correspondente, a UDisF verifica se esta saída está desocupada. Se a saída estiver desocupada, a UDisF envia a ficha à saída

correspondente. Se a saída estiver ocupada podem ocorrer duas possibilidades dependendo do destino da ficha. Se a ficha se destinar ao MES, a UDisF aguarda que esta unidade fique disponível. Isto é feito devido ao pequeno número de fichas que são destinadas a esta unidade. Se a ficha se destinar à Memória de Dados, a UDisF a envia para a Fila de Fichas (descrita na seção 2.4.7). Quando uma ficha é enviada com sucesso, a UDisF envia à UCO uma mensagem para que esta providencie uma nova ficha.

2.4.3 A Memória de Dados

A Memória de Dados (MemDad) é responsável pelo emparelhamento e armazenamento de fichas destinadas a operações diádicas. As fichas referentes a operações monádicas passam com um tratamento mínimo pelas unidades da MemDad. A MemDad possui uma memória RAM para armazenar fichas que estão esperando a parceira.

O número de fichas armazenadas nesta unidade fornece uma medida para a dependência de dados do algoritmo usado. Por exemplo, se em um determinado momento durante a execução de um programa o número de fichas nesta unidade começar a crescer, pode-se afirmar que o ramo do grafo que está sendo executado está gerando fichas para operações diádicas cujo segundo operando não está ainda disponível. Quando ocorre este tipo de situação, costuma-se dizer que as fichas produzidas são de “vida longa”. Se a ocupação da unidade começa a diminuir, existem dependências de dados sendo resolvidas e os resultados produzidos não estão gerando nova dependência, mas sim ajudando na diminuição desta.

Cada ficha possui três informações que a MemDad usa para endereçar a memória: o

destino da instrução para a qual o dado se destina, o nome de ativação do dado e o índice. Com estas três informações, a MemDad tem acesso a qualquer posição da memória. A Figura 2.4 apresenta as unidades que compõem a MemDad.

Figura 2.4: A Memória de Dados.

Unidade de Emparelhamento - UE: A UE recebe as fichas da CC e verifica se a ficha se destina a uma instrução monádica (um operando) ou diádica (dois operandos). Caso a ficha se destine a uma instrução monádica, a UE emparelha esta ficha com uma ficha do tipo *Null* e envia o pacote para a Chave de Distribuição (descrita na seção 2.4.6). O emparelhamento com uma ficha do tipo *Null* é necessário, pois podem haver constantes armazenadas na Memória de Instruções (descrita na seção 2.4.4) e esta precisa saber se tal constante é o operando do lado esquerdo ou direito da instrução correspondente. Se a ficha se destinar a uma instrução diádica, a UE envia à Memória o endereço de armazenamento da ficha e aguarda uma resposta. Se a resposta for uma ficha tipo *Null*, a UE envia a ficha recebida para que seja armazenada na Memória. Se a resposta não for uma ficha do tipo *Null*, será então a

ficha armazenada naquele endereço. Esta é a parceira da ficha que está sendo tratada pela UE. A UE constrói então um pacote contendo as informações referentes às duas fichas e envia para a Chave de Distribuição.

Memória de Armazenamento de Dados - MAD: A MAD é responsável pelo armazenamento de fichas que estão aguardando a parceira. Quando a UE envia um endereço para a MAD, esta verifica se existe alguma ficha armazenada neste endereço. Caso exista uma ficha armazenada no endereço, a MAD extrai tal ficha e a envia para a UE. Se não existir ficha armazenada no endereço, a MAD envia uma ficha tipo *Null* para a UE. A UE responde com a ficha a ser armazenada. Quando a ficha chega à MAD, esta armazena-a no endereço correspondente. Esta unidade faz também o monitoramento de fichas destinadas à operações diádicas. Os resultados deste monitoramento são usados para obter as curvas de “ocupação da Memória de Dados” apresentadas no Capítulo 4.

2.4.4 A Memória de Instruções Interna

A Memória de Instruções Interna (MI) é responsável pelo armazenamento do programa a ser executado. A MI também possui um mecanismo para controlar a execução de trechos de código seqüencial. Nesta unidade pode-se medir a quantidade de instruções executadas como parte de uma cadeia. Esta medida fornece o tamanho dos “macro-nós” executados, dando uma boa medida do grau de seqüencialidade imposto pelo programa. A Figura 2.5 apresenta as unidades que compõem a MI.

Unidade de Decodificação de Fichas - UDecF: A UDecF divide a ficha de entrada enviando o endereço da instrução correspondente para a Memória de Armazena-

Figura 2.5: A Memória de Instruções Interna.

mento de Instruções e o restante da ficha para a Unidade de Codificação de Fichas.

Unidade de Controle e Sequencialização - UCS: A UCS recebe da Memória de Armazenamento de Instruções os endereços das próximas instruções a serem executadas, ou seja as instruções para as quais os resultados produzidos pela execução da instrução que está sendo tratada serão enviados. Com estes endereços a UCS tem condições de identificar se alguma das instruções imediatamente posteriores poderão ser executadas sequencialmente. Em caso afirmativo, a UCS envia uma mensagem para a UDecF para que esta não receba mais fichas da Chave de Distribuição e retorna uma mensagem para a Memória de Armazenamento de Instruções. Esta unidade recebe a mensagem com novos endereços e repete o procedimento descrito acima até que a UCS identifique um caso onde as instruções imediatamente posteriores não possam ser executadas sequencialmente. Quando esta situação ocorre, a UCS envia uma mensagem para a UDecF para que volte a receber fichas da Chave de Distribuição. Podem ocorrer casos onde apenas um dos dois endereços poderá

continuar sendo executado seqüencialmente. Nestes casos a UCS envia como parte da mensagem para a Memória de Armazenamento de Instruções a identificação de qual dos ramos do grafo deverá ser executado seqüencialmente.

Memória de Armazenamento de Instruções - MAI: A MAI armazena o programa a ser executado. Quando a MAI recebe da UDecF o endereço da instrução, ela executa uma leitura naquele endereço enviando seu conteúdo para a Unidade de Codificação de Fichas e apenas os endereços de destino dos resultados que serão produzidos com a execução da instrução que está sendo tratada para a UCS. Ao receber uma mensagem da UCS, a MAI repete a operação descrita neste item.

Unidade de Codificação de Fichas - UCodF: A UCodF recebe da UDecF os dados correspondentes à instrução a ser executada. Recebe também da MAI a instrução correspondente. A UCodF constrói e envia então um pacote contendo estas informações para a Unidade Funcional (descrita na seção 2.4.8). A UCodF possui um mecanismo de sincronização que faz o controle da chegada das duas partes do pacote que é enviado para a Unidade Funcional.

2.4.5 A Memória de Instruções Externa

A Memória de Instruções Externa (MI0) é responsável por verificar se os dados que chegam são destinados a uma instrução de tratamento de vetores/escalares ou controle de atividade da máquina. Instruções de tratamento escalar são instruções que são executadas em uma Unidade Funcional enquanto que instruções de tratamento vetorial são executadas na Memória de Estruturas. As instruções de tratamento vetorial ou de controle de atividade são passadas para a Chave de Distribuição já com a instrução correspondente.

Os vários módulos descritos para esta unidade são semelhantes aos módulos descritos para a Memória de Instruções Interna. A Figura 2.6 apresenta o diagrama de blocos da MIO.

Figura 2.6: A Memória de Instruções Externa.

Unidade de Decodificação de Fichas - UDecF: A UDecF divide a ficha de entrada enviando o endereço da instrução correspondente para a Memória de Armazenamento de Instruções e o restante da ficha para a Unidade de Codificação de Fichas.

Unidade de Tratamento de Instruções Vetoriais - UTIV: A UTIV recebe da Memória de Armazenamento de Instruções a instrução (*opcode*) referente à ficha de entrada. Quando a UTIV recebe tal instrução, esta identifica se é uma instrução de tratamento vetorial/escalar ou de controle de atividade. Se for uma instrução de tratamento escalar, a UTIV retorna uma mensagem à Memória de Armazenamento de Instruções e à Unidade de Codificação de Fichas para que liberem a ficha de entrada sem nenhuma modificação para a saída. Caso seja uma instrução de tratamento vetorial ou de controle de atividade, a UTIV envia uma mensagem para a

Memória de Armazenamento de Instruções para que esta libere o conteúdo completo do endereço para a Unidade de Codificação de Fichas.

Memória de Armazenamento de Instruções - MAI: A MAI armazena o programa a ser executado (esta unidade é semelhante à descrita na Memória de Instruções Interna). Quando a MAI recebe da UDecF o endereço da instrução, ela executa uma leitura naquele endereço enviando a instrução propriamente dita para a UTIV e uma cópia do endereço para a Unidade de Codificação de Fichas. Se a mensagem de retorno da UTIV for referente a uma instrução de tratamento escalar, a MAI fica esperando por outro endereço da UDecF. Caso contrário, a MAI envia todo o conteúdo do endereço para a Unidade de Codificação de Fichas.

Unidade de Codificação de Fichas - UCodF: A UCodF recebe da UDecF uma ficha destinada a uma instrução. Recebe também da UTIV uma mensagem para que tal ficha seja liberada caso não seja destinada a uma instrução de tratamento vetorial ou de controle de atividade. Se a ficha for destinada a uma instrução de tratamento vetorial ou de controle de atividade, a UCodF recebe da MAI um pacote contendo todas as informações armazenadas no endereço passado pela UDecF. Com a ficha oriunda da UDecF e o pacote de informações enviados pela MAI, a UCodF prepara uma nova ficha pronta para ser executada e a envia para a Chave de Distribuição.

2.4.6 A Chave de Distribuição

A Chave de Distribuição (CD) é responsável pelo recebimento das fichas vindas da Memória de Instruções Externa e pelo envio de fichas à Memória de Estruturas, à Unidade de Controle de Paralelismo e para os anéis memória-processador (MI-UF). Após identificar

que a ficha está endereçada para um anel MI-UF, a CD procura sequencialmente por ordem crescente de anéis (do primeiro até o último) até encontrar um anel livre. Encontrando um anel livre a CD envia a ficha para este anel. Este mecanismo foi adotado devido a três fatores:

1. **Facilidade de Implementação:** Por não envolver o uso de tabelas e funções randômicas para armazenagem e escolha do anel livre, já que o uso destas tabelas e funções envolveriam recursos adicionais em termos de simulação, o mecanismo é simples de ser implementado.
2. **Desempenho não se Altera:** Em termos de simulação, não há perdas no desempenho do simulador. Isto não ocorreria se estivésemos implementando o mecanismo em *hardware*, já que neste último o desempenho seria alterado devido à necessidade de implementar *hardware* extra para conter a tabela e resolver as funções randômicas.
3. **Curva de Distribuição:** O uso do mecanismo nos permite ter uma idéia realista da distribuição de carga pelos anéis, tornando possível uma avaliação da quantidade de anéis efetivamente usados durante a execução de um programa.

A Figura 2.7 mostra as duas unidades que formam a CD.

Unidade de Identificação de Fichas - UIF: A UIF recebe as fichas da Memória de Instruções Externa. Estas fichas possuem um identificador que indica para qual das saídas desta unidade ela deve ser enviada. A UIF envia então a ficha para a saída correspondente.

Arbitrador: O Arbitrador recebe as fichas que se destinam aos anéis memória-processador e as envia para o primeiro anel livre. Para identificar o anel livre, o Arbitrador

Figura 2.7: A Chave de Distribuição.

possui um mecanismo que verifica a presença de fichas em cada uma de suas saídas. Esta busca é feita de forma seqüencial ocorrendo desta forma uma super-utilização dos anéis mais superiores (esta super-utilização será discutida no Capítulo 5).

2.4.7 A Fila de Fichas

A Fila de Fichas (FF) é responsável pelo armazenamento de fichas. Quando a Chave de Coleta não pode enviar fichas para a Memória de Dados por esta estar ocupada, envia então para esta unidade. A FF foi criada para evitar situações de *deadlock*³. A FF possui um mecanismo interno que dá prioridade às fichas que chegam primeiro à unidade, ou seja quando fichas estão armazenadas na FF, a primeira ficha que é mandada para a Chave de Coleta é a ficha que possui seu campo *tempo* menor.

A ocupação desta unidade é uma medida da latência provocada pela Memória de Dados

³Um *deadlock* ocorre quando todas as unidades do *pipeline* possuem dados prontos para serem enviados ao próximo estágio mas não podem enviá-los para não comprometer a integridade dos dados do estágio seguinte.

devido à sua incapacidade de tratar fichas na mesma taxa em que estas são produzidas. Quando a ocupação da Fila de Fichas cresce, significa que a produção de fichas está numa taxa maior que a Memória de Dados consegue tratar. A quantidade de fichas nesta unidade não deve ser muito grande, pois significaria que a ineficiência da MD está afetando o desempenho global. Por outro lado, se a ocupação for zero, pode significar que existam unidades ociosas (*starvation*). O tamanho ótimo não deve ser dependente do programa, mas para implementar isto deve-se usar os conceitos de *throttle*. A Figura 2.8 mostra as unidades internas à FF.

Figura 2.8: A Fila de Fichas.

Unidade de Decodificação de Fichas - UDecF: A UDecF divide a ficha de entrada em duas partes. A primeira parte consiste do campo que diz em qual momento a ficha foi criada e é enviada para a Unidade de Gerenciamento da Fila. A segunda parte é composta pelo restante da ficha e é enviada à Unidade de Armazenamento.

Unidade de Gerenciamento da Fila - UGF: A UGF controla o armazenamento de fichas na Unidade de Armazenamento através de uma tabela com os endereços

de todas as fichas armazenadas. Esta tabela implementa prioridade para sair, da ficha que existe já há mais tempo na unidade. Quando a UGF recebe da UDecF a informação referente ao momento da chegada da ficha que está sendo enviada à Unidade de Armazenamento, a UGF inicia um mecanismo de comparação entre a informação que chegou e as informações armazenadas em sua tabela. Ao encontrar uma posição adequada para a informação que chega, a UGF envia um endereço para a Unidade de Armazenamento para que esta possa armazenar a ficha. A cada modificação na tabela, a UGF envia a quantidade de fichas armazenadas para a Unidade de Controle de Paralelismo.

Unidade de Armazenamento - UA: A UA armazena fichas em endereços enviados pela UGF. Quando a ficha na saída para a Chave de Coleta é consumida, a UA envia uma mensagem para a UGF para que esta envie um endereço para liberação de uma nova ficha. Quando o endereço chega à UA, esta extrai a ficha daquele endereço e a envia para a saída da Chave de Coleta. quando a ficha é extraída, a UGF elimina seu endereço da tabela.

2.4.8 A Unidade Funcional

A Unidade Funcional (UF) é responsável pela execução da instrução. Esta unidade possui um contador que é incrementado a cada vez que uma instrução é executada. Este contador é utilizado para medir a utilização dos anéis MI-UF nas considerações sobre balanceamento apresentadas no Capítulo 5. O diagrama de blocos da UF é mostrado na Figura 2.9.

Unidade de Decodificação de Fichas - UDecF: A UDecF divide a ficha de entrada

registradores devem ser enviados para a Unidade Lógica Aritmética.

Banco de Registradores - BR: O BR armazena os operandos das fichas de entrada e os resultados intermediários durante a execução de código seqüencial. Quando chega uma instrução na UDecF referente a um trecho de código seqüencial, os seus dados são do tipo *Null*. Toda vez que o BR recebe um dado do tipo *Null*, este envia uma cópia dos registradores específicos para a Unidade Lógica Aritmética. Quando o BR recebe dados da Unidade Lógica Aritmética, estes são armazenados.

Unidade de Controle Seqüencial - UCS: A UCS recebe da UDecF os endereços dos resultados que serão produzidos pela instrução que está sendo tratada e através destes endereços identifica se há a possibilidade de executar uma das instruções imediatamente posteriores em seqüencial. Se for possível executar alguma instrução em seqüencial, a UCS envia uma mensagem para a Unidade de Codificação de Fichas para que esta não libere novas fichas para a Chave de Coleta (descrita na seção 2.4.2). As mensagens enviadas para a Unidade de Codificação de Fichas informam-na sobre como deverá ser feita a construção da ficha que sairá para a Chave de Coleta. A UDecF pode receber também da UII uma mensagem que identifica uma instrução de tratamento vetorial. Neste caso a UCS interrompe a execução do trecho seqüencial enviando uma mensagem para a Unidade de Codificação de Fichas.

Unidade Lógica Aritmética - ULA: A ULA recebe do BR os operandos necessários à execução de uma instrução e recebe da UII a instrução decodificada para que possa ser executada. A ULA aplica a instrução sobre os operandos e envia os resultados de volta ao BR e também para a Unidade de Codificação de Fichas.

Unidade de Codificação de Fichas - UCodF: A UCodF recebe mensagens da UCS

e pode controlar a saída de fichas para a Chave de Coleta. Esta unidade constrói a ficha de saída com base em informações locais. Estas informações são provenientes da cópia da ficha de entrada enviada pela UDecF e da cópia de resultados enviados pela ULA. Uma mensagem enviada pela UCS para a UCodF é a informação sobre qual campo da ficha deve ser atualizado pelo dado vindo da ULA. Isto se faz necessário já que a ULA pode realizar operações que mudam alguns campos especiais da ficha (índice e nome de ativação). Quando a UCS envia uma mensagem para a UCodF referente ao término de trecho seqüencial, a UCodF constrói uma ficha com as informações armazenadas localmente e envia tal ficha para a Chave de Coleta.

2.4.9 A Memória de Estruturas

O esquema da Memória de Estruturas (ME) é mostrado na Figura 2.10. Esta memória divide-se em quatro sub-unidades: Fila de Acessos Deferidos (FAD), Memória (Mem) propriamente dita, Unidade de Desalocação (UnDes) e Alocador (Al).

A ME pode ser implementada através do uso de um bloco de memória RAM convencional para armazenar os vetores e matrizes. Uma palavra na ME contém o tipo do dado com dois *bits* adicionais chamados de *bit* de presença e *bit* de acesso deferido. O uso destes dois *bits* permite que leituras e escritas sejam feitas fora de ordem (leituras antes de escritas).

O *bit* de presença indica a presença de um dado e a leitura poderá ser feita normalmente. Se a leitura ocorrer primeiro, esta será adiada e o pedido de leitura fica armazenado na FAD. O *bit* de acesso deferido é colocado em um para indicar esta situação. Uma escrita

subseqüente no endereço, liberará o pedido da FAD.

A coleta de lixo (*garbage collection*) é implementada pela UnDes (descrita a seguir). Para isto é usado um contador de referências. Quando o contador de referências chega a zero, a coleta de lixo é iniciada e o espaço alocado é finalmente liberado para futuras alocações. Segue uma breve descrição de cada sub-unidade da ME:

Figura 2.10: A Memória de Estruturas.

Alocador - Al: O Al aloca espaço na Mem para armazenagem de estruturas de dados tais como vetores e matrizes.

Unidade de Desalocação - UnDes: A UnDes dispara a coleta de lixo quando o contador de referências chega a zero.

Memória - Mem: A Mem é a unidade que propriamente armazena as estruturas de dados.

Fila de Acessos Deferidos - FAD: A FAD armazena os pedidos de leitura em endereços e os pedidos de decremento do contador de referências.

A Memória de Estruturas tem muitos detalhes complexos, omitidos porque não são diretamente relacionados com este trabalho. O trabalho de Sargeant [26] descreve com detalhes a ME.

2.5 Linguagens para Programação Paralela

Esta seção apresenta três linguagens de alto nível para programação de computadores paralelos a saber: *Data Parallel C Extensions* (DPCE) [28], *High Performance Fortran* (HPF) [29] e *Streams and Iteractions in a Single Assignment Language* (SISAL) [30]. Programação paralela é um dos problemas não resolvidos em arquiteturas paralelas. Uma das ambições das pesquisas em arquiteturas a fluxo de dados é a solução deste problema, fazendo a exploração de paralelismo sem o auxílio do programador.

As linguagens DPCE e HPF foram escolhidas por se tratarem de extensões às linguagens C e FORTRAN respectivamente, que são largamente utilizadas no meio acadêmico [31, 32]. A linguagem SISAL foi escolhida por se tratar da linguagem utilizada na MFDM. Como este trabalho é evolucionário, a linguagem SISAL foi usada para desenvolver os programas utilizados neste trabalho (Apêndice A). Várias outras linguagens são utilizadas para programação de máquinas a fluxo de dados. Duas das principais são: Id [11] usado no MIT (Monsoon) e uma versão de C sem ponteiros e com atribuição única [22] (DFCII) usado no Japão (EM-4).

Não é objetivo deste trabalho discutir em detalhes os mecanismos de paralelização implementados por estas três linguagens, apenas far-se-á uma breve introdução a elas. A *World Wide Web* (WWW) trás em vários *sites* maiores informações sobre estas linguagens. Nenhum *site* específico será citado, devendo ser procurados através de um dos vários

searchers disponíveis.

2.5.1 Características de uma Linguagem para Programação Paralela

A programação paralela em linguagens imperativas tais como FORTRAN e C, não é uma tarefa fácil. Estas linguagens, por terem sido desenvolvidas para computadores seqüenciais, não implementam paralelismo com naturalidade. Neste caso, para escrever o algoritmo, o programador deve codificar as operações de sincronização e comunicação. Esta complexidade extra aumenta o custo de desenvolvimento de programas e também dificulta a depuração dos mesmos. A linguagem SISAL possui características indispensáveis à uma linguagem para programação paralela:

1. **Ausência de Efeitos Colaterais** A linguagem não deve apresentar efeitos colaterais para que a dependência dos dados não se modifique durante a execução do programa. Uma forma comum de efeito colateral é a modificação de variáveis do programa principal através da passagem de parâmetro para procedimentos por referência e não por valor. Para evitar efeitos colaterais, a linguagem não deve implementar variáveis globais e o compilador deve ser capaz de analisar as variáveis envolvidas tendo em vista a restrição dos seus alcances. Além destas exigências, as máquinas fluxo de dados exigem ainda que nem mesmo um procedimento possa modificar seus argumentos. Com estas restrições podem ainda surgir efeitos colaterais envolvendo a manipulação de estruturas de dados, por exemplo como matrizes. Estes efeitos surgem da maneira como estas estruturas são manipuladas pelas linguagens convencionais, e uma tentativa de solucionar o problema seria a manipulação de tais

estruturas de dados como se fossem escalares.

2. **Localidade de Efeito** A *localidade de efeitos* também é uma característica de uma linguagem para fluxo de dados. Localidade de efeitos significa que uma instrução não tem uma dependência de dados muito longa, isto é, as variáveis usadas para os cálculos dentro da instrução são temporárias o que torna possível sua reutilização posterior. Esta reutilização pode causar uma degradação na execução concorrente de dois fragmentos de código que usem o mesmo conjunto de variáveis, pois existe uma falsa dependência entre os dados armazenados nestas variáveis e neste caso, a única solução seria ter um compilador que pudesse identificar esta falsa dependência e removê-la através da utilização de dois conjuntos diferentes de variáveis.
3. **Escalonamento Vinculado** O escalonamento de uma instrução é vinculado à dependência dos dados. Esta característica garante que uma instrução só será executada quando todos os seus operandos (dados) estiverem disponíveis. Desta forma todas as informações necessárias para a execução de uma instrução estão contidas no grafo de fluxo de dados do programa e já que este grafo é o código de máquina para um computador fluxo de dados, esta propriedade torna a linguagem apropriada para tais computadores.
4. **Atribuição Única** Uma outra propriedade para linguagens de fluxo de dados é a convenção da atribuição única. Esta propriedade reza que uma variável pode aparecer no lado esquerdo de uma atribuição apenas uma vez dentro da área de programa em que ela está ativa. Desta forma pode-se evitar efeitos colaterais derivados da reutilização de variáveis. Esta propriedade juntamente com esta necessidade de evitar efeitos colaterais requer que a linguagem apresente uma sintaxe característica.

2.5.2 *Data Parallel C Extensions - DPCE*

Esta subseção apresenta conceitos básicos sobre uma extensão da linguagem de programação C. Maiores informações poderão ser encontradas em [28]. DPCE é um conjunto de extensões à linguagem C que implementa aplicações em programação paralela. O objetivo do DPCE é prover uma forma de escrever programas que executam eficientemente em sistemas paralelos.

Neste modelo de programação, o programador especifica a distribuição de seus dados em alto nível e o programa é escrito como se os dados fossem endereçáveis globalmente por todos os processadores com apenas um *thread* de execução lógico possível.

Este modo de programação é mais simples do que se o programador tivesse que tratar dados em diversos processadores e com vários *threads* de execução. O compilador cria e trata todos os procedimentos paralelos e a transferência de dados entre os processadores, baseado na distribuição original feita pelo programador.

2.5.3 *High Performance Fortran - HPF*

Esta subseção apresenta conceitos básicos sobre uma extensão da linguagem de programação FORTRAN. Maiores informações poderão ser encontradas em [29]. HPF é uma extensão da linguagem de programação FORTRAN 90 e implementa programação paralela através da especificação da distribuição de estruturas de dados para os processadores de um computador paralelo.

Devido à estreita relação entre HPF e a plataforma sobre a qual ele executa, o programador não familiarizado com a plataforma na qual trabalha, necessita de ajuda es-

pecializada no preparo de seus programas paralelos. Numa tentativa de minimizar este problema, os desenvolvedores da linguagem estão incluindo no pacote um editor para ajudar a distribuição eficiente dos dados, um analisador de desempenho e um sistema de visualização.

O projeto de desenvolvimento desta linguagem possui um objetivo secundário, o de prover uma plataforma para o desenvolvimento de outras ferramentas de *software* para HPF. A versão atual do HPF suporta programação paralela de dados para 50% a 70% das aplicações mas tem suporte limitado para “problemas irregulares”.

2.5.4 *Streams and Iterations in a Single Assignment Language - SISAL*

Esta subseção apresenta conceitos básicos sobre a linguagem de programação SISAL. Maiores informações poderão ser encontradas em [30]. O projeto SISAL tem por objetivo o desenvolvimento de compiladores funcionais de alto desempenho e sistemas para simplificar o processo de escrita de programas científicos em supercomputadores paralelos.

SISAL permite a construção de programas paralelos sem a necessidade do programador possuir grandes conhecimentos sobre processamento paralelo. Explora paralelismo implícito baseado na dependência de dados.

As funções, no sentido exato de suas definições matemáticas, são a base da linguagem SISAL e podem ser definidas recursivamente não importando o nível de recursão usado. A linguagem possui os operadores normalmente encontrados em qualquer outra linguagem de alto nível, além de funções já pré-definidas.

2.6 O Processo de Compilação para o Simulador

Não é interesse deste trabalho fazer um estudo detalhado de cada formato intermediário gerado pela compilação de um programa em SISAL. O objetivo desta seção é mostrar como um programa SISAL é convertido para o grafo a fluxo de dados usado para obtenção dos resultados deste trabalho. No sistema de compilação de *Manchester/LLNL*, para compilar um programa em SISAL [8], existem algumas etapas pelas quais o código deve passar. Estas etapas estão mostradas na Figura 2.11. Na primeira etapa o programa SISAL é convertido em um formato intermediário chamado IF1 [33]. Este formato é independente da máquina e é uma representação gráfica de SISAL. O compilador que gera o formato IF1 faz a parte mais pesada da compilação.

O formato IDC é gerado a partir do formato IF1 e contém uma mistura de nós primitivos (nós com várias entradas e saídas) e macro-nós. Nesta fase da compilação faz-se necessária a quebra destes macro-nós em nós primitivos. Para isto usa-se um outro compilador que transforma o formato IDC em um formato onde existem apenas nós primitivos. Este formato é chamado de IGF.

O formato IGF é transformado através do uso de um otimizador. O otimizador produz um código eficiente e otimiza o número de saídas de um nó. O *linker* gera o formato final do código. Três arquivos são produzidos: um contendo o programa em si (.COD), outro contendo os dados de entrada (.DAT) e o outro uma tabela de segmentos usada na carga do arquivo de programa (.SEG).

Por exemplo, a seqüência de comandos da Figura 2.12 compila um programa em SISAL chamado EX1.SIS.

Figura 2.11: Processo de Compilação.

```

$ SISAL EX1.SIS
$ IF1 EX1 IF1
  * Loading Program
  * Type Equivalence
  * Structure Check
  * Starting analysis
  * Inlining Analysis
    Representation analysis
    Local analysis
    Loop Analysis
    Ref. counting and strictness analysis
    Number of IF nodes unchanged during analysis
  * Coding Program
    Coding Function MAIN
    Coding Main Function call
  * Finished
$ IDC EX1.IDC
$ OPTIM EX1.IGF
$ LNK/1 EX1.REL
$
  
```

Figura 2.12: Sequência de Compilação de um Programa SISAL.

Se existir um programa principal chamado de PRI.SIS e dois subprogramas SUB1.SIS e SUB2.SIS deve-se compilá-los separadamente até o estágio da otimização, quando então deve-se criar um arquivo contendo o nome destes três arquivos. Este arquivo é chamado arquivo mestre e contém os nomes dos arquivos a serem processados, um por linha e é usado com o linker. O linker gera então os arquivos de código, dados e segmento. E o comando LNK PRI une PRI.REL, SUB1.REL e SUB2.REL para formar PRI.COD, PRI.DAT e PRI.SEG.

2.7 Algoritmos Utilizados

Esta seção faz uma introdução aos algoritmos utilizados nas simulações, apresentando algumas de suas características mais marcantes. O código fonte (escrito na linguagem funcional SISAL [30]) de cada um destes algoritmos pode ser encontrado no Apêndice A.

Não é objetivo desta seção a abordagem de detalhes do código fonte. Apenas as características gerais de cada programa serão apresentadas. A escolha destes programas para testar o simulador não foi arbitrária: todos eles (exceto Dinâmica Molecular) já foram utilizados em outras arquiteturas em outros grupos de pesquisa como programas de teste. Alguns dos programas possuem referência bibliográfica associada. Nestas podem ser encontrados maiores detalhes sobre tais programas. Os programas utilizados não são *toy programs*, podendo ser considerados programas de uso geral. Os valores numéricos utilizados como entrada dos programas encontram-se na seção 4.2.

Ackermann: A função de Ackermann [34] é definida por:

$$Ack(0, n) = n + 1; Ack(m, 0) = Ack(m - 1, 1); Ack(m, n) = Ack(m - 1, Ack(m, n - 1));$$

pela definição da função, observa-se que esta é recursiva e gera uma árvore de processos. O número de chamadas ao algoritmo para o cálculo de $Ack(3, n)$ é:

$$\frac{128*4^n - 120*2^n + 9*n + 37}{3}$$

O resultado pode ser obtido por:

$$Ack(3, n) - 1 = 2^{n+3} - 4$$

Dinâmica Molecular: Este algoritmo é baseado no modelo de sistema conhecido como *ensemble* microcanônico de Lennard-Jones [35]. É um algoritmo altamente paralelizável que usa exaustivamente os recursos de armazenamento e gerenciamento de vetores e matrizes (Memória de Estruturas).

Somatorial: Este algoritmo calcula o somatório de N pela fórmula:

$$Sorna(N, (N + 1)/2 + 1) + Sorna((N + 1)/2, 1)$$

É um algoritmo duplamente recursivo que gera uma árvore binária de processos onde cada folha de um mesmo nível pode ser executada em paralelo.

FFT: Este é um algoritmo altamente paralelo que aplica a transformada rápida de Fourier sobre uma matriz $N \times N$. É um algoritmo numérico [36].

Gauss: Aplica o algoritmo de eliminação de Gauss com pivoteamento parcial para resolver um sistema linear. É um algoritmo numérico [37].

Interpolação: Este algoritmo usa o método de interpolação polinomial para fazer aproximações. É um algoritmo numérico [37].

Laplace: Este algoritmo resolve uma equação simples de Laplace sobre um *grid* quadrado usando o método de relaxação [38].

Multiplicação de Matrizes: Este algoritmo cria e multiplica duas matrizes quadradas [38].

N-Queens: Este algoritmo resolve o problema das N rainhas. É um algoritmo não numérico e altamente recursivo e paralelo que usa o método de *backtracking* [39] para resolver o problema [27].

Quicksort: Este algoritmo de ordenação é usado para ordenar um vetor de números. É um algoritmo recursivo.

Simple: Este é um algoritmo relacionado com hidrodinâmica e que possui um alto grau de paralelismo [27].

2.8 Conclusões

Este capítulo registra que a arquitetura a fluxo de dados Wolf está bem detalhada “a nível de” unidade. A especificação da descrição funcional de cada unidade, apresentada neste capítulo, foi realizada para possibilitar a implementação de um simulador para a arquitetura Wolf.

A simulação não é sistêmica, isto é cada unidade opera ignorante do funcionamento das demais. O conjunto se comporta como um multicomputador. Isto fornece maior confiança no simulador como um todo. Todas as descrições dos itens da seção 2.4, com exceção da Memória de Estruturas, são contribuições originais deste trabalho.

Este capítulo apresentou também o processo usado para compilar os programas usados para obtenção dos resultados com o simulador Saw. Este processo é o mesmo desenvolvido para a Máquina a Fluxo de Dados de Manchester (MFDM). O grafo gerado pela com-

pilação de um programa escrito em SISAL é carregado diretamente no simulador. A escolha do compilador da MFDM para ser o compilador do Wolf não foi aleatória, já que o embrião da arquitetura Wolf sofreu a influência direta dos conceitos desenvolvidos para a MFDM.

Capítulo III

Investigando a Arquitetura Wolf

3.1 Introdução

Este capítulo apresenta a investigação da arquitetura Wolf utilizando o simulador Saw. Para isto, são discutidos alguns problemas tais como exploração de seqüencialidade em arquiteturas a fluxo de dados e como a arquitetura Wolf procura resolver tal problema. Descreve também os experimentos realizados com o simulador Saw para balancear cada uma das unidades que o compõe. Para isto serão apresentados resultados obtidos com simulações. Também serão apresentados resultados de multiprogramação (execução simultânea de vários programas) no simulador.

3.2 Implementação do Simulador

O simulador Saw é um simulador dirigido a tempo desenvolvido usando a linguagem orientada a objetos C++. Este simulador implementa as características da arquitetura Wolf descritas anteriormente. Os resultados numéricos apresentados pelo simulador estão de acordo com os resultados apresentados pelos simulador SIM e MR desenvolvidos para

a Máquina a Fluxo de Dados de *Manchester* (MFDm) [40, 41]. A implementação de todas as unidades seguiu à risca a descrição funcional apresentada no Capítulo 2.

3.2.1 O Mecanismo de Temporização

Existe um relógio global no simulador que controla a utilização de fichas durante a execução de um programa. Tal controle é possível devido à utilização de um campo especial em cada ficha denominado *tempo*. Uma ficha é considerada pronta para ser consumida se e somente se o seu campo *tempo* possuir um valor igual ou menor que o valor do relógio global.

O relógio global é incrementado cada vez que todas as unidades implementadas no simulador são chamadas. O valor final deste relógio é usado como o número total de ciclos gastos na simulação. Cada unidade da arquitetura simulada possui um número de ciclos característico associado a ela (discutido na seção 3.4.2.1). Isto é necessário para simular o tempo gasto para uma unidade executar uma tarefa. O campo *tempo* de uma ficha é atualizado toda vez que a ficha passa de uma unidade a outra no simulador.

Supondo que o atraso de uma unidade é cinco *time steps*¹ e que o valor do relógio global é, em um determinado momento igual a T , e finalmente que o tempo da ficha na entrada de uma unidade é também igual ou menor que T , então esta ficha está pronta para ser executada e o resultado produzido será uma ficha com tempo igual a $T + 5$.

3.3 Exploração de Seqüencialidade

¹*time step* é a menor unidade de tempo gerenciada pelo simulador. Neste trabalho também chamamos esta unidade de *ciclo*.

O avanço das pesquisas em arquiteturas a fluxo de dados permitiu detectar problemas no tratamento de código seqüencial [42, 43, 44]. Esta seção apresenta uma técnica eficiente para tratar o problema de execução de código seqüencial em uma arquitetura a fluxo de dados.

3.3.1 Execução de Código Seqüencial em Arquiteturas a Fluxo de Dados

As implementações a fluxo de dados apresentam um baixo desempenho quando executam programas com baixo paralelismo. O tamanho do *pipeline*, geralmente faz com que o desempenho de um programa pouco paralelizável que seja executado em uma máquina com tal arquitetura, seja pior se comparado à execução deste mesmo programa em uma arquitetura von Neumann convencional [43, 20] devido ao *overhead* gerado. Este tamanho excessivo do pipeline não é consequência de falhas de implementação mas sim de uma exigência do modelo a fluxo de dados visto que todas as principais implementações possuem um pipeline comprido.

A partir do grafo de dependência de dados da Figura 3.1 serão discutidos alguns outros problemas. O grafo mostra um exemplo típico de código seqüencial cuja execução não é eficiente em uma arquitetura a fluxo de dados convencional:

Neste código, um ordenamento seqüencial das instruções é forçado pela dependência dos dados. As arquiteturas von Neumann, devido à sua seqüencialidade implícita, são muito eficientes nestes casos, pois geralmente usam registradores para armazenar resultados intermediários entre instruções diminuindo consideravelmente a latência na execução de uma instrução.

Num processador a fluxo de dados tradicional nenhuma inferência sobre o escalona-

Figura 3.1: Código Seqüencial.

mento da instrução Op2 pode ser efetuada no instante do escalonamento da instrução Op1. Entretanto é fácil ver que a regra de disparo que escalonou Op1 também disparou, devido à dependência dos dados, as instruções Op2, Op3 e Op4. Como processadores a fluxo de dados têm dificuldades em identificar este escalonamento implícito, a ficha com o resultado de Op1 precisa circular por todo o pipeline para disparar Op2, e assim sucessivamente para Op3 e Op4.

Esta ineficiência é ainda maior considerando-se que nenhuma instrução subsequente a Op1 necessita de dados adicionais. A única entrada destas outras instruções é o resultado de sua predecessora. Na terminologia da MFD, estas fichas dirigidas a instruções unárias são denominadas *bypass*. Estas fichas são, não obstante suas características, também enviadas para a Memória de Dados, onde um mecanismo força o emparelhamento com um parceiro fantasma (do tipo Nulo).

Outro problema detectado nesta situação é a criação de fichas de “vida curta”, tais como o resultado intermediário de Op1, que é necessário apenas para disparar Op4. Num

processador a fluxo de dados típico, este resultado intermediário é armazenado na Memória de Dados e quase imediatamente retirado. Esta ação desnecessária associada ao tráfego de fichas do tipo *bypass* e o mecanismo de emparelhamento associativo, fazem com que a Memória de Dados seja uma das unidades mais lentas e críticas do pipeline [45].

O tráfego de fichas do tipo *bypass* na Memória de Dados é dependente do programa, causando um *overhead* desnecessário devido ao emparelhamento com fichas do tipo Nulo.

Tabela 3.1: Tráfego de Fichas pela Memória de Dados

Algoritmo	Total de Fichas	Fichas <i>Bypass</i>	% Fichas <i>Bypass</i>
Ackermann	57.462	25.678	44,7
Dinâmica Molecular	4.733.478	1.182.836	25,0
FFT	353.322	121.780	34,5
Gauss	47.311	9.108	19,2
Interpolação	199.524	59.405	29,8
Laplace	722.136	196.480	27,2
Simple	850.901	240.859	28,3
N-Queens	292.744	127.268	43,5
Quicksort	37.763	15.707	41,6

A Tabela 3.1 mostra a relação entre o tráfego total de fichas e o tráfego de fichas do tipo *bypass* pela Memória de Dados. A coluna % **Fichas *Bypass*** indica a porcentagem de fichas *bypass* em relação ao total de fichas. Para *Ackermann*, 44,7% das fichas que passam pela Memória de Dados é do tipo *bypass*.

Já para *Gauss*, apenas 19,2% é do tipo *bypass*. Com estes resultados, pode-se concluir que se houver um mecanismo que detecte a existência de dependência de dados como discutida nesta seção, tal mecanismo poderá ser utilizado para diminuir o *overhead* gerado pela execução de código implicitamente seqüencial na arquitetura Wolf.

Com o objetivo de minimizar os problemas na execução de código seqüencial descrito

nesta seção, será descrito a seguir um mecanismo que explora a seqüencialidade implícita ao código.

3.3.2 Mecanismo Simples de Detecção e Execução de Código Seqüencial

Quando a ficha executável é tratada na UF, até duas outras fichas podem ser geradas. A ficha produzida pode ser de dois tipos: bf livre ou bf encadeada. Fichas livres são passadas para a Chave de Coleta recirculando no *pipeline*. Fichas encadeadas permanecem nos registradores internos da Unidade Funcional que as gerou para serem, posteriormente, usadas como operandos de instruções subseqüentes. O mecanismo que permite isto é descrito a seguir.

Um dos dados armazenado pela Memória de Instruções para cada nó é a informação de que tal nó é parte de uma cadeia. Quando um pacote executável de uma cadeia é produzido, a Memória de Instruções inicia por conta própria, um ciclo de busca usando o endereço de destino do pacote executável produzido como endereço de busca. Desta forma, quando a Unidade Funcional termina o consumo do pacote executável anterior, um novo pacote executável está em condição de ser produzido pela Memória de Instruções.

Este novo pacote executável usará os resultados intermediários do pacote executável anterior como entrada, produzindo novos resultados. Este mecanismo é simples e de fácil implementação, visto que não exige recursos adicionais do compilador. Este mecanismo diminui o tráfego de fichas pelo *pipeline*, diminuindo assim o *overhead* gerado pela execução de código seqüencial na arquitetura Wolf.

Outras arquiteturas também apresentam suas soluções para o problema de execução de código seqüencial. Para resolver este problema, o grupo de pesquisa do Japão criou o

conceito de *Macro-Dataflow* e o grupo de pesquisa do MIT, o conceito de *thread*. Estes conceitos já foram apresentados no Capítulo 1.

3.3.2.1 Resultados da Implementação no Simulador

O programa de “eliminação de Gauss” será usado para obter os resultados da implementação do mecanismo de detecção de código seqüencial no simulador. A Figura 3.3 apresenta a curva de ocupação da Memória de Dados sem a utilização do mecanismo de execução de código seqüencial. A Figura 3.2 apresenta a curva de ocupação da Memória de Dados com a utilização do mecanismo de execução de código seqüencial.

Figura 3.2: Ocupação da Memória de Dados para 1 *Gauss*.

Ambas as curvas possuem o mesmo aspecto, inclusive no número de fichas armazenadas na unidade em cada instante. A diferença entre elas está no tempo total para executar o programa. Sem a utilização do mecanismo, o tempo total para executar *Gauss* foi de 284.326 *time steps*. Com a utilização do mecanismo, o tempo total para executar *Gauss* foi de 266.220 *time steps*. Com a utilização do mecanismo, o tempo total diminui. Este

comportamento foi identificado para todos os demais algoritmos utilizados neste trabalho.

Figura 3.3: Ocupação da Memória de Dados sem Seqüencialidade.

A Figura 3.4 apresenta a curva de ocupação para a Fila de Fichas sem a utilização do mecanismo. A Figura 3.8 apresenta a ocupação da Fila de Fichas com a utilização do mecanismo. Não há diferenças marcantes no formato destas curvas. Como no caso da Memória de Dados, a curva mostra uma diminuição no número total de *time steps* necessários à execução do programa quando o mecanismo de detecção de código seqüencial é utilizado. Observa-se entretanto que existe um pequeno aumento na utilização da Fila de Fichas para o caso da utilização do mecanismo. Isto ocorre devido à diminuição do *overhead* causado por fichas de vida curta, aumentando a exploração de paralelismo pela arquitetura.

Devido às suas características, o mecanismo de detecção de código seqüencial será utilizado para o restante dos experimentos realizados neste trabalho.

Figura 3.4: Ocupação da Fila de Fichas sem Sequencialidade.

3.4 Experimento de Balanceamento das Unidades do Wolf

Esta seção apresenta os motivos que levaram à realização do balanceamento nas unidades do Wolf. Em seguida, o método utilizado será discutido. Finalmente, serão apresentados os resultados seguidos de sua interpretação e validação.

O balanceamento consistiu em alterar as velocidades de cada unidade no simulador. O motivo que nos levou à realização deste experimento foi a verificação da existência de unidades sobrecarregadas (*hot spots*) durante os primeiros experimentos com o simulador. O ajuste destas unidades tornou-se então necessário.

Para o experimento usa-se como entrada para o simulador o programa de “eliminação de Gauss com pivoteamento parcial”. Tal programa foi escolhido por apresentar características de paralelismo e um tempo total de simulação não muito grande².

²Na pior configuração do simulador (ao final deste experimento), *Gauss* executou em aproximadamente vinte minutos.

Devido à quantidade de experimentos realizados, não poderia-se escolher um programa para entrada do simulador cuja simulação fosse muito onerosa. Será mostrado na seção 3.4.2.2 que esta escolha não interferiu nos resultados finais. O experimento consistiu das seguintes etapas:

1. **Inicialização das Unidades:** Nesta etapa, o tempo que cada unidade gasta para processar uma ficha em sua entrada e soltar uma ficha resultante em sua saída foi inicializado em uma unidade de *time step*.
2. **Execução do Programa Gauss:** Nesta etapa, o simulador foi carregado com o programa de eliminação de Gauss e este foi então executado.
3. **Cálculo de Porcentagens:** Nesta etapa, a fração do tempo total de simulação que as unidades ficaram ativas foi calculada e os resultados foram apresentados em forma de porcentagens nas tabelas 3.2, 3.3, 3.4, 3.5 e 3.6.
4. **Cálculo de Parâmetros para Comparação:** Nesta etapa, os parâmetros para comparação dos resultados foram calculados. Para o cálculo da média aritmética simples e desvio padrão (referindo-se às tabelas da seção 3.4.2: **mdia** e **dpd**) as unidades foram classificadas em duas categorias: Unidade Isolada e Unidade Agrupada. Na categoria de Unidade Isolada, foram consideradas as unidades que possuem apenas um representante na simulação (referindo-se às tabelas da seção 3.4.2: **mi_0**, **cd**, **cc**, **ms**, **ss** e **tq**). Na categoria de Unidade Agrupada, foram consideradas as unidades que possuem mais de um representante na simulação (referindo-se às tabelas da seção 3.4.2: **mi_x** e **p_x**, $x = 0, 1, 2, 3 \text{ e } 4$). O coeficiente de variação (rotulado como **%%** nas tabelas da seção 3.4.2) foi calculado dividindo-se o **dpd** por **mdia** e multiplicando-se o resultado por 100. A média aritmética foi calculada

tomando-se os tempos de todas as unidades isoladas e o tempo do representante mais ativo entre as unidades agrupadas. A seção 3.4.1 apresenta uma discussão sobre os critérios de escolha destes tempos.

5. **Análise dos Resultados:** Nesta etapa, a porcentagem do tempo total de simulação que cada unidade ficou ativa foi considerada. Com base nestas porcentagens, acrescentou-se um atraso de 1 *time step* à unidade mais ativa do *pipeline* para equilibrar o trabalho desta unidade. Nos casos onde havia mais de uma unidade com a mesma porcentagem de atividade, apenas uma foi considerada.

As etapas 2, 3, 4 e 5 foram repetidas até obter-se um valor considerado *bom* para os tempos de cada unidade como discutido a seguir.

A escolha dos tempos na categoria de Unidade Isolada é trivial e o tempo de todas as unidades foi considerado. Os tempos de cada conjunto $\mathbf{mi}_x$ e $\mathbf{p}_x$ ($x = 0, 1, 2, 3 \text{ e } 4$) foi considerado em separado e para os cálculos da etapa 4, foram consideradas apenas as maiores porcentagens. Isto se faz necessário visto que todas as $\mathbf{mi}$'s e $\mathbf{p}$'s são módulos idênticos. Nota-se nas tabelas apresentadas na seção 3.4.2, que existe um desbalanceamento nos módulos $\mathbf{mi}_x - \mathbf{p}_x$ ($x = 0, 1, 2, 3 \text{ e } 4$). Este problema será discutido no Capítulo 5.

3.4.1 Critérios para Escolha de Tempos e Parada

Após a verificação da necessidade das unidades serem balanceadas, dois critérios foram estudados: no primeiro, o tempo de cada unidade seria inicializado em algum valor maior que 1 (pode-se escolher 100) *time step* e à medida que o experimento fosse sendo feito, o tempo da unidade que possuísse a maior porcentagem de utilização seria decrementado tornando-a mais rápida; no segundo, o tempo de cada unidade seria inicializado em 1 *time*

step e à medida que o experimento fosse sendo feito, o tempo da unidade que possuísse a menor porcentagem de utilização seria incrementado tornando-a mais lenta.

O primeiro critério possui um inconveniente intrínseco: o limite superior arbitrário. Usando este critério, não há garantias de encontrar um bom valor para os tempos de cada unidade, visto que existe um limite inferior igual a 1 *time step* para os tempos de todas as unidades. Outro inconveniente deste critério é o aumento excessivo do número de *time steps* necessários para a execução de um programa se comparado ao caso onde as unidades estão todas com *time step* igual a 1. O critério a ser utilizado não deveria ter estas limitações, permitindo uma liberdade maior na escolha do critério de parada.

O segundo critério mostrou-se mais conveniente que o anterior, não possuindo as limitações do primeiro sendo portanto utilizado para os experimentos. Não existe um limite superior para os *time steps* de cada unidade e ainda pode-se ter controle sobre a relação entre o tempo total para execução do programa com os tempos de cada unidade inicializado em 1 *time step* (T_1) e o tempo total de execução em cada parte do experimento (T_t).

O experimento utilizando o segundo critério, introduz atrasos nas unidades. Este atraso faz com que o tempo total em cada parte do experimento (T_t) seja maior que T_1 , isto é, para ter-se unidades balanceadas precisa-se de um tempo maior para que um programa seja executado. Este aumento no tempo final de execução de um programa não deve ser preocupante, já que este experimento aproxima os tempos de cada unidade do simulador para tempos mais reais, visto que em uma máquina comercial existe uma diferença nas velocidades de cada unidade, por exemplo processadores e memórias possuem tempos diferentes.

O experimento descrito nesta seção demanda tempo principalmente para a análise dos

resultados e execução do simulador. À medida que as unidades tornam-se mais lentas com o aumento de seus tempos, o tempo total gasto para a simulação também aumenta. Um dos motivos é a dificuldade de sincronização. A princípio, este experimento poderia ser continuado além da Tabela 3.6, até que o coeficiente de variação chegasse a zero ou próximo a zero.

Nota-se pelos resultados que o coeficiente de variação é muito sensível ao aumento do tempo de cada unidade. O experimento pode ser encerrado, pois a meta foi alcançada: mostrou-se que o *pipeline* pode ser balanceado e que o resultado da temporização de cada unidade é independente do programa executado.

3.4.2 Resultados do Experimento

Os resultados obtidos com o experimento de balanceamento das unidades foram divididos em 5 tabelas para facilitar a apresentação e podem ser observados nas tabelas 3.2, 3.3, 3.4, 3.5 e 3.6. Com exceção de **GT**, **%%**, **dpd** e **mdia**, todos os resultados são apresentados em forma de porcentagens como descritos na seção 3.4.

Na Tabela 3.2, observamos o início do experimento. A primeira coluna mostra os nomes de cada unidade (**mi0**, **p0**, **mi1**, **p1**, **mi2**, **p2**, ... e **tq**), o nome dado para os parâmetros de comparação (**mdia**, **dpd** e **%%**) e o tempo total que o programa levou para ser executado (**GT**). Na primeira linha, observa-se o tempo inferido para uma dada unidade, por exemplo **Txx=1** significa o momento inicial quando todas as unidades possuem tempo de 1 *time step*; **Tss=2** o acréscimo de uma unidade de tempo na *Structure Store* ou Memória de Estruturas; **Tss=3** o acréscimo de mais uma unidade de tempo na Memória de Estruturas e assim sucessivamente.

Tabela 3.2: Porcentagem de Utilização de cada unidade - parte 1.

	Txx=1	Tss=2	Tss=3	Tmi=2	Tss=4	Tss=5	Tmi0=2	Tcd=2	Tss=6	Tp=2	Tmi=3	Tss=7
mi0	17,7	17,4	17,4	32,2	31,4	30,1	31,4	31,4	30,5	27,1	38,7	37,4
p0	31,2	30,9	30,5	29,0	28,1	27,4	27,5	27,3	26,8	47,6	45,5	44,2
mi1	5	4,9	4,6	11,5	11,1	10,9	5,6	5,6	5,5	8,6	14,6	14,5
p1	10,7	10,4	10,1	11,7	11,3	10,7	6,7	6,7	6,7	18,1	19,8	19,4
mi2	1,3	1,3	1,3	2,7	2,6	2,6	1,5	1,6	1,5	1,9	2,8	2,9
p2	3,3	3,2	3,1	3,1	3,1	3,0	2,0	2,1	1,9	5,0	4,9	4,8
mi3	0,4	0,4	0,4	0,8	0,8	0,7	0,5	0,5	0,4	0,5	0,8	0,7
p3	0,9	1,0	1,1	0,9	0,9	0,7	0,7	0,6	0,5	1,3	1,5	1,3
mi4	0,1	0,1	0,1	0,2	0,2	0,2	0,1	0,1	0,0	0,1	0,3	0,2
p4	0,2	0,2	0,3	0,1	0,2	0,2	0,2	0,2	0,1	0,4	0,5	0,4
mi_0	29,1	28,6	28,2	28,1	27,3	26,4	46,4	46,4	45,1	45,4	45,3	43,9
ss	6,3	12,5	18,4	18,4	23,8	28,7	25,3	25,3	29,5	29,6	29,6	33,5
cd	29,1	28,6	28,2	28,1	27,3	26,4	23,2	46,4	45,1	45,4	45,3	43,9
cc	47,9	47,2	46,6	46,4	45,0	43,5	39,2	39,2	38,0	38,6	38,7	37,4
ms	47,8	47,1	46,5	46,3	45,0	43,5	38,2	38,2	37,1	37,3	37,2	36,1
tq	38,3	37,9	37,2	36,5	35,5	34,5	33,1	33,0	32,2	31,4	31,3	30,4
mdia	30,9	31,3	31,6	33,1	32,9	32,6	33,0	35,9	35,5	37,8	38,9	38,3
dpd	14,2	12,6	11,2	9,6	8,2	7,2	7,9	8,0	7,0	7,9	6,2	5,2
%%	46,0	40,2	35,4	29,0	24,9	22,2	23,8	22,4	19,6	20,8	16,0	13,6
GT	98.886	100.523	101.827	102.345	105.316	109.006	123.945	123.915	127.576	126.813	127.083	131.018

Tabela 3.3: Porcentagem de Utilização de cada unidade - parte 2.

	Ttq=2	Tss=8	Tmi=4	Tss=9	Tms=2	Tcc=2	Tss=10	Tmi0=3	Tcd=3	Tss=11	Tp=3	Tmi=5
mi0	36,6	35,3	43,9	42,4	38,4	38,5	37,5	39,0	38,8	38,1	36,3	42,9
p0	43,9	42,3	39,8	38,7	34,4	35,0	33,9	34,9	34,7	33,9	48,1	46,2
mi1	15,5	14,8	22,8	22,0	15,0	15,1	15,1	11,0	10,9	10,6	12,5	17,8
p1	20,3	19,5	21,7	20,8	14,8	14,6	14,5	11,2	11,2	11,3	19,2	21,1
mi2	3,9	3,8	5,3	5,5	3,2	3,5	3,4	2,3	2,4	2,4	2,7	3,8
p2	5,9	5,8	6,1	6,3	3,9	4,1	4,2	2,8	3,0	3,0	5,0	5,3
mi3	1,2	1,2	1,4	1,4	0,9	0,9	0,9	0,6	0,6	0,6	0,7	0,7
p3	1,9	1,8	1,6	1,7	1,1	1,2	1,1	0,9	0,9	0,8	1,5	1,2
mi4	0,3	0,4	0,4	0,4	0,3	0,3	0,2	0,2	0,2	0,1	0,2	0,2
p4	0,5	0,5	0,5	0,3	0,3	0,3	0,3	0,4	0,3	0,1	0,5	0,4
mi_0	45,4	43,9	43,8	42,5	34,2	34,6	33,9	47,3	47,1	46,2	46,6	46,6
ss	34,6	38,2	38,1	41,7	33,5	33,9	36,9	34,3	34,1	36,8	37,2	37,2
cd	45,4	43,9	43,8	42,5	34,2	34,6	33,9	31,5	47,1	46,2	46,6	46,6
cc	39,1	37,8	37,5	36,4	32,9	57,1	55,8	52,6	52,4	51,4	51,8	51,9
ms	37,4	36,1	36,0	35,0	56,3	57,0	55,8	51,9	51,6	50,7	51,1	51,1
tq	50,2	49,0	48,7	47,6	46,5	45,2	44,1	43,1	42,8	42,0	41,7	41,5
mdia	41,6	40,8	41,4	40,9	38,8	42,0	41,5	41,8	43,6	43,2	44,9	45,5
dpd	5,4	4,7	4,3	4,0	8,4	10,0	9,5	8,2	7,2	6,5	5,9	4,9
%%	13,1	11,6	10,3	9,8	21,5	23,7	22,8	19,5	16,4	15,0	13,2	10,8
GT	126.658	131.066	131.438	135.267	168.100	166.093	169.680	182.519	183.371	186.931	185.287	185.164

Tabela 3.4: Porcentagem de Utilização de cada unidade - parte 3.

	Tss=12	Tss=13	Ttq=3	Tmi=6	Tp=4	Tss=14	Tmi=7	Tmi0=4	Tcd=4	Tss=15	Tss=16	Tms=3
mi0	42,0	41,1	41,2	47,0	43,6	43,1	47,7	48,8	49,8	48,5	47,3	45,3
p0	44,9	44,3	44,4	42,9	52,8	52,0	50,4	50,6	50,9	49,9	48,9	47,0
mi1	17,2	17,0	17,4	23,6	24,6	24,1	30,7	25,5	24,7	24,5	23,9	19,3
p1	20,5	20,1	20,8	22,8	31,3	30,9	33,4	28,5	27,7	27,3	26,9	21,3
mi2	3,6	3,5	4,1	5,2	6,5	6,1	7,6	5,0	5,0	4,8	4,9	3,4
p2	5,2	5,0	5,7	5,7	9,6	9,0	9,0	6,4	6,5	6,5	6,1	4,7
mi3	0,9	0,9	1,2	1,5	1,9	1,7	1,5	1,1	1,0	1,0	0,9	0,8
p3	1,6	1,4	1,7	1,8	2,8	2,5	1,7	1,4	1,7	1,4	1,4	1,3
mi4	0,2	0,1	0,2	0,3	0,4	0,4	0,1	0,2	0,3	0,2	0,2	0,2
p4	0,4	0,3	0,3	0,4	0,8	0,6	0,2	0,3	0,5	0,3	0,3	0,2
mi_0	45,5	44,6	45,7	46,1	45,7	44,7	44,6	54,7	54,7	53,6	52,3	46,7
ss	39,6	42,1	43,1	43,5	43,1	45,4	45,3	41,7	41,7	43,8	45,6	40,7
cd	45,5	44,6	45,7	46,1	45,7	44,7	44,6	41,0	54,7	53,6	52,3	46,7
cc	50,7	49,6	51,3	51,7	51,2	49,9	49,8	47,2	47,1	46,1	45,0	41,5
ms	49,9	48,9	50,1	50,5	50,1	49,0	48,9	45,0	45,0	44,1	43,1	57,6
tq	40,6	39,8	52,0	52,2	51,5	50,4	50,5	49,5	49,5	48,6	47,8	46,8
mdia	44,9	44,4	46,7	47,5	48,0	47,4	47,7	47,3	49,2	48,5	47,8	46,5
dpd	4,0	3,5	4,0	3,6	3,8	3,3	2,6	4,6	4,5	3,8	3,3	5,1
%%	9,0	7,9	8,5	7,6	8,0	7,0	5,4	9,8	9,2	7,8	7,0	11,0
GT	189.571	193.467	188.752	187.209	188.803	193.261	193.686	210.252	210.231	214.675	219.785	246.304

Tabela 3.5: Porcentagem de Utilização de cada unidade - parte 4.

	Tss=17	Tcc=3	Tss=18	Tmi=8	Tp=5	Tss=19	Ttq=4	Tmi=9	Tmi0=5	Tcd=5	Tss=20	Tss=21
mi0	44,6	44,0	43,5	46,7	45,9	45,1	46,0	48,9	49,5	49,9	49,1	47,8
p0	46,4	45,0	45,2	42,3	51,6	50,1	51,8	50,2	50,0	50,1	49,4	48,3
mi1	19,2	18,8	18,8	23,4	24,0	23,3	24,2	30,0	25,9	25,0	24,5	24,6
p1	21,2	21,0	20,7	22,9	29,1	28,3	29,7	31,3	27,8	27,4	26,6	26,4
mi2	3,2	3,5	3,5	4,8	5,3	5,2	5,4	6,0	4,6	5,2	4,9	4,6
p2	4,4	5,0	4,9	5,3	7,5	7,7	7,7	7,4	5,7	6,4	6,0	6,0
mi3	0,7	0,9	0,9	1,1	1,1	1,3	1,5	1,3	1,0	1,2	1,0	1,0
p3	1,0	1,5	1,3	1,5	1,8	2,1	2,1	1,7	1,6	1,4	1,5	1,2
mi4	0,1	0,3	0,2	0,3	0,3	0,4	0,4	0,3	0,2	0,2	0,2	0,2
p4	0,2	0,5	0,3	0,4	0,6	0,8	0,5	0,5	0,3	0,3	0,3	0,3
mi_0	45,9	45,8	45,4	45,3	45,5	44,7	46,0	45,7	53,6	53,7	52,6	51,6
ss	42,5	42,3	44,4	44,4	44,6	46,2	47,6	47,3	44,3	44,4	45,8	47,2
cd	45,9	45,8	45,4	45,3	45,5	44,7	46,0	45,7	42,8	53,7	52,6	51,6
cc	40,7	56,9	56,5	56,4	56,6	55,7	57,6	57,1	55,2	55,3	54,2	53,2
ms	56,7	56,5	56,0	55,9	56,1	55,1	56,8	56,4	52,8	53,0	51,9	50,9
tq	46,0	45,2	44,9	44,5	44,7	43,7	52,4	52,4	51,4	51,6	50,6	49,6
mdia	46,1	47,7	47,7	47,6	48,8	48,2	50,5	50,5	50,0	51,5	50,8	50,0
dpd	4,7	5,7	5,3	5,4	5,2	4,9	4,8	4,5	4,4	3,4	2,6	2,1
%%	10,2	11,9	11,2	11,4	10,6	10,1	9,6	8,9	8,8	6,6	5,2	4,3
GT	250.471	251.463	253.678	254.003	252.984	257.483	250.059	251.753	268.544	267.796	273.477	278.896

Tabela 3.6: Porcentagem de Utilização de cada unidade - parte 5.

	Tss=22	Tmi=10	Tp=6	Tmi=11	Tss=23	Ttq=5	Tp=7	Tmi=12	Tmi=13	Tms=4
mi0	47,1	50,8	48,6	51,0	50,4	49,8	48,2	49,9	52,5	52,0
p0	47,4	46,4	53,0	50,3	50,2	49,0	55,5	53,7	52,5	51,4
mi1	23,5	29,0	29,5	33,9	33,9	33,3	33,2	37,1	40,0	35,3
p1	25,2	27,6	33,6	34,8	35,2	34,2	40,2	40,4	40,1	36,1
mi2	4,7	5,1	6,5	8,0	7,8	8,2	9,9	12,4	14,3	10,5
p2	5,9	5,9	9,1	10,0	9,1	10,2	13,3	14,8	15,7	11,9
mi3	1,2	1,1	1,4	1,8	1,5	2,2	2,5	3,0	3,1	2,3
p3	1,8	1,4	1,9	2,4	2,0	2,6	3,7	3,8	3,6	2,6
mi4	0,4	0,2	0,3	0,4	0,3	0,4	0,6	0,4	0,5	0,5
p4	0,5	0,2	0,5	0,6	0,4	0,7	0,9	0,8	0,6	0,5
mi_0	50,7	51,2	51,2	51,3	50,6	50,6	50,9	50,8	50,4	45,9
ss	48,6	49,0	49,0	49,1	50,7	50,7	50,9	50,9	50,5	46,0
cd	50,7	51,2	51,2	51,3	50,6	50,6	50,9	50,8	50,4	45,9
cc	52,3	52,7	52,6	52,8	52,2	52,3	52,6	52,3	51,8	49,2
ms	50,1	50,5	50,6	50,6	49,9	49,9	50,2	50,2	49,8	60,4
tq	48,8	49,4	49,7	49,7	49,0	56,1	56,2	56,0	55,7	55,9
mdia	49,5	50,2	50,7	50,8	50,4	51,1	51,9	51,8	51,7	50,8
dpd	1,8	1,9	1,6	1,1	0,9	2,2	2,7	2,1	1,9	5,3
%%	3,6	3,8	3,1	2,2	1,7	4,3	5,2	4,0	3,7	10,4
GT	283.612	280.935	280.899	280.455	284.326	284.332	282.720	282.871	285.376	313.285

As tabelas 3.3, 3.4, 3.5 e 3.6 apresentam a continuação do experimento de balanceamento das unidades.

3.4.2.1 Escolha de um dos Resultados

A seção 3.4 menciona, na etapa 4, o modo de cálculo dos parâmetros para comparação. As fórmulas matemáticas usadas para estes cálculos são [46]:

1. **Porcentagens de Utilização:** Calculada dividindo-se o tempo que a unidade ficou ativa ($T_{unidade}$) pelo tempo total gasto para executar o programa (GT):

$$unidade = (\frac{T_{unidade}}{GT} * 100)$$

2. **Média Aritmética:** Baseada nas porcentagens de utilização das seguintes unidades: **mi0, p0, mi_0, ss, cd, cc, ms e tq:**

$$mdia = (\frac{mi0 + p0 + mi_0 + ss + cd + cc + ms + tq}{8})$$

3. **Desvio Padrão:** Calculado tomando-se as mesmas porcentagens utilizadas para o cálculo da média aritmética:

$$dpd = \sqrt{(\frac{((mi0-mdia)^2+(p0-mdia)^2+(mi_0-mdia)^2+(ss-mdia)^2+(cd-mdia)^2+(cc-mdia)^2+(ms-mdia)^2+(tq-mdia)^2)}{7})}$$

4. **Coeficiente de Variação:** Baseado no Desvio Padrão e Média Aritmética:

$$\% \% = (\frac{dpd}{mdia} * 100)$$

O resultado escolhido como configuração ideal para os tempos de cada unidade corresponde à coluna 6 da Tabela 3.6. Este resultado foi escolhido por apresentar o menor coeficiente de variação ($\% \% = 1,7$) e o menor desvio padrão ($dpd = 0,9$) de todo o experimento. Portanto, os tempos que serão utilizados no simulador são os apresentados pela Tabela 3.7. Note-se que estes resultados são originais deste trabalho.

Tabela 3.7: Tempos Utilizados para as Unidades do Simulador

Unidade	Tempo em <i>time steps</i>
Chave de Coleta - CC	3
Chave de Distribuição - CD	5
Memória de Dados - MS	3
Memória de Instruções Externa - MI0	5
Memória de Instruções Interna - MI	11
Fila de Fichas - FF	4
Memória de Estruturas - SS	23
Unidade Funcional - P	6

3.4.2.2 Validação dos Resultados

Esta seção apresenta os resultados obtidos com outros dois programas escolhidos entre os vários programas disponíveis: *FFT* e *Laplace*. Será mostrado através destes dois programas, que a escolha inicial do programa *Gauss* não interferiu no resultado final do experimento de balanceamento das unidades. Após esta validação, serão usados os resultados como generalização para todos os programas subseqüentemente estudados.

Tabela 3.8: Validação do Experimento.

	fft_1:	%	fft_var:	%	lapl_1:	%	lapl_var:	%
mi0	166.884	23,0	1.304.886	61,4	286.917	18,8	2.387.792	47,3
p0	269.529	37,2	1.172.952	55,2	430.919	28,3	2.017.188	39,9
mi1	38.904	5,4	926.288	43,6	67.806	4,4	1.503.183	29,8
p1	68.943	9,5	834.966	39,3	127.096	8,3	1.332.324	26,4
mi2	5.978	0,8	97.482	4,6	19.094	1,3	221.925	4,4
p2	11.734	1,6	101.124	4,8	41.725	2,7	252.132	5,0
mi3	1.713	0,2	18.315	0,9	7.714	0,5	87.516	1,7
p3	4.596	0,6	23.574	1,1	18.393	1,2	110.544	2,2
mi4	691	0,1	8.910	0,4	4.633	0,3	47.399	0,9
p4	2.154	0,3	9.126	0,4	12.126	0,8	69.372	1,4
mi_0	245.905	33,9	1.229.535	57,9	463.735	30,4	2.318.685	45,9
ss	45.816	6,3	1.053.768	49,6	113.014	7,4	2.599.322	51,5
cd	245.906	33,9	1.229.535	57,9	463.736	30,4	2.318.685	45,9
cc	354.116	48,9	1.133.112	53,4	733.190	48,1	2.282.805	45,2
ms	353.574	48,8	1.061.868	50,0	722.132	47,4	2.166.405	42,9
tq	296.284	40,9	1.119.248	52,7	602.542	39,5	2.279.012	45,1
mdia		34,1		54,8		31,3		45,5
dpd		14,1		4,1		13,9		3,3
%%		41,2		7,6		44,3		7,3
GT		724.623		2.123.805		1.523.921		5.050.204

A Tabela 3.8 mostra os resultados para *FFT* e *Laplace*. A primeira coluna da tabela é idêntica à primeira coluna da Tabela 3.2. A segunda coluna apresenta o número de ciclos que cada unidade ficou ativa usando o programa de *FFT* (neste caso, o tempo de todas as unidades é igual a 1 *time step* - **fft₁**). A terceira coluna apresenta a porcentagem de atividade de cada unidade, a média (**mdia**), o desvio padrão (**dpd**), o coeficiente de variação (**%%**) e o tempo total de simulação (**GT**) dado em *time steps*.

A quarta coluna trás o número de ciclos que cada unidade ficou ativa usando o pro-

grama de *FFT* (neste caso, o tempo de cada unidade é o apresentado na seção 3.4.2.1 - **fft_{var}**). A quinta coluna trás a porcentagem de atividade de cada unidade, a média (**mdia**), o desvio padrão (**dpd**), o coeficiente de variação (**%%**) e o tempo total de simulação (**GT**) dado em *time steps*.

As colunas 3 e 5 mostram que existe um aumento na uniformidade de utilização de cada unidade. Quando o programa de *FFT* para **fft₁** foi executado, obteve-se $dpd = 14,1$ e $\% \% = 41,2$. Quando os tempos das unidades apresentados na seção 3.4.2.1 foram utilizados, obteve-se $dpd = 4,1$ e $\% \% = 7,6$. Com estes resultados pode-se concluir que para o programa de *FFT* utilizado, os dados obtidos para o **dpd** e **%%** diminuem para valores abaixo de 10% quando aplica-se os tempos apresentados na seção 3.4.2.1 às unidades. Conclui-se portanto que os resultados do experimento descrito na seção 3.4 podem ser utilizados para executar o programa de *FFT*.

As descrições das colunas 6 e 8 são idênticas às descrições das colunas 2 e 4 respectivamente. Um aspecto a ser considerado sobre as colunas 6 e 8 é que estas se referem ao experimento realizado com o programa de *Laplace*. A mesma consideração deve ser aplicada às colunas 7 e 9 que apresentam os resultados obtidos com *Laplace*.

As colunas 7 e 9 mostram que existe um aumento na uniformidade de utilização de cada unidade (como ocorreu com *FFT* e *Gauss*). Quando o programa de *Laplace* para **lapl₁** foi executado, obteve-se $dpd = 13,9$ e $\% \% = 44,3$. Quando os tempos das unidades apresentados na seção 3.4.2.1 foram utilizados, obteve-se $dpd = 3,3$ e $\% \% = 7,3$. Com estes resultados pode-se concluir que para o programa de *Laplace* utilizado, os dados obtidos para o **dpd** e **%%** diminuem para valores abaixo de 10% quando aplicamos os tempos apresentados na seção 3.4.2.1 às unidades. Conclui-se portanto que os resultados do experimento descrito na seção 3.4 podem ser utilizados para executar o programa de

Laplace.

Com os resultados apresentados na Tabela 3.8 far-se-á uma generalização do resultado dos tempos de cada unidade apresentado na seção 3.4.2.1 para todos os programas que serão utilizados para obtenção dos gráficos apresentados no Capítulo 4.

3.5 O Experimento de Multiprogramação

O experimento de multiprogramação consistiu em modificar o simulador de forma que pudessem ser executados vários programas simultaneamente. As características do modelo a fluxo de dados permite tratar programas diferentes como se fossem instâncias diferentes de um processo. Não existe portanto a necessidade de se usar mecanismos para diferenciar dados de um programa em relação aos dados de um outro programa.

Tabela 3.9: Multiprogramação com Gauss.

	1(gauss):	2(gauss):	3(gauss):	4(gauss):
mi0	143.363	289.179	432.795	577.005
p0	142.668	285.744	429.576	570.474
mi1	96.250	192.709	288.794	382.844
p1	100.134	198.246	299.322	395.442
mi2	22.066	41.184	62.887	84.425
p2	25.830	52.044	76.506	105.510
mi3	4.180	8.822	12.958	18.601
p3	5.562	12.360	17.040	24.510
mi4	803	1.430	2.552	3.773
p4	1.038	2.070	3.252	4.992
mi_0	143.820	287.640	431.460	575.280
ss	144.026	288.052	432.078	576.104
cd	143.820	287.640	431.460	575.280
cc	148.326	300.273	451.761	603.555
ms	141.924	283.929	425.835	567.798
tq	139.384	284.420	427.868	569.412
Relação	1,0	2,0	1,5	1,3
Primeiro	1,0	2,0	3,0	4,0
GT	284.326	562.421	843.459	1.125.590

A Tabela 3.9 mostra os resultados da execução do programa de “eliminação de Gauss” no simulador. Na primeira coluna da tabela, estão representadas as unidades do simulador (linha 2 à linha 17). Na linha 20 está representado o tempo total para execução do(s) programa(s) (**GT**). Na linha 18 (**Relação**), está representada a relação entre **GT** do experimento em questão e **GT** do experimento anterior (para **1(gauss)**, a regra é aplicada a ele mesmo). Na linha 19, está representada a relação (**Primeiro**) entre **GT** do experimento em questão e **GT** do primeiro experimento.

As colunas 2, 3, 4 e 5 contém os tempos de atividade de cada unidade para o programa de “eliminação de Gauss” executado uma, duas, três e quatro vezes simultâneas respectivamente. Uma análise da linha 19 mostra um aumento constante dos tempos necessários para execução dos programas. Este comportamento pode ser explicado considerando-se o mecanismo de escalonamento adotado no modelo de fluxo de dados (tudo que estiver pronto, é escalonado para execução). O coeficiente da reta formada pelos dados da linha 19 e pelo número de programas executados simultaneamente é 1, não havendo melhora no tempo final para executar **4(gauss)** se comparado ao tempo final para executar 4 vezes **1(gauss)**.

A Tabela 3.10 mostra o experimento de multiprogramação utilizando o programa “Fatorial”. A descrição das linhas e colunas é semelhante à descrição apresentada para a Tabela 3.9. A análise da linha 19 mostra um aumento linear do tempo total de execução dos programas. Diferentemente dos dados obtidos para “eliminação de Gauss”, o programa “Fatorial” mostra um aumento menos acentuado do tempo final de simulação para múltiplas execuções.

O coeficiente da reta formada pelos dados da linha 19 e pelo número de programas executados simultaneamente é 1,25. Portanto, temos uma melhora no tempo final para

Tabela 3.10: Multiprogramação com *Fatorial*.

	1(fat):	2(fat):	3(fat):	4(fat):	5(fat):
mi0	2.618	5.159	7.755	10.362	12.958
p0	2.130	4.302	6.510	8.688	11.064
mi1	1.991	4.070	6.094	8.096	10.109
p1	1.758	3.486	5.178	6.888	8.400
mi2	11	11	11	22	33
p2	12	12	12	24	36
mi3	0	0	0	0	0
p3	0	0	0	0	0
mi4	0	0	0	0	0
p4	0	0	0	0	0
mi_0	2.100	4.200	6.300	8.400	10.500
ss	0	0	0	0	0
cd	2.100	4.200	6.300	8.400	10.500
cc	1.992	4.092	6.189	8.361	10.551
ms	1.908	3.816	5.724	7.632	9.540
tq	1.804	3.804	5.660	7.596	9.560
Relação	1	1,8	1,5	1,3	1,2
Primeiro	1	1,8	2,6	3,4	4,2
GT	3.885	6.908	10.019	13.131	16.294

execução de **5(fat)** se comparado ao tempo final para executar 5 vezes **1(fat)** (a relação entre o tempo final para executar **5(fat)** e $5 \cdot 1(\text{fat})$ é de 83,9%).

A Tabela 3.11 apresenta os resultados obtidos com o experimento de multiprogramação utilizando o programa “Ackermann”. A descrição das linhas e colunas é semelhante à descrição apresentada para a Tabela 3.9. Os resultados apresentados na linha 19 são idênticos aos obtidos para o programa “Fatorial” e portanto foi apresentado apenas o primeiro e último experimentos (**1(ack)** e **5(ack)**).

A Tabela 3.12 apresenta os resultados obtidos com o experimento de multiprogramação para 5 programas diferentes: “Ackermann”, “Fatorial”, “Gauss”, “Laplace”, “FFT”. A descrição das linhas e colunas é semelhante à descrição apresentada para a Tabela 3.9. Nas colunas 8 e 9, estão representadas duas medidas de tempo de simulação: **T(separado)** e **T-multi** respectivamente.

Tabela 3.11: Multiprogramação com *Ackermann*.

	l(ack):	5(ack):
mi0	296.791	1.227.875
p0	228.054	966.324
mi1	160.479	1.058.079
p1	131.304	830.136
mi2	0	396
p2	0	330
mi3	0	0
p3	0	0
mi4	0	0
p4	0	0
mi_0	207.850	1.039.250
ss	0	0
cd	207.850	1.039.250
cc	175.947	926.190
ms	172.386	861.930
tq	82.304	892.808
Relação	1	4,2
Primeiro	1	4,2
GT	324.771	1.348.006

Tabela 3.12: Multiprogramação com 5 Programas Diferentes.

	l(ack):	l(fat):	l(gauss):	l(lapl):	l(fft):	5(multi):	T(separado)	T-multi
mi0	296.791	2.618	143.363	2.397.792	1.304.886	4.121.183		
p0	228.054	2.130	142.668	2.017.188	1.172.952	3.556.914		
mi1	160.479	1.991	96.250	1.503.183	926.288	2.695.627		
p1	131.304	1.758	100.134	1.332.324	834.966	2.404.656		
mi2	0	11	22.066	221.925	97.482	340.901		
p2	0	12	25.830	252.132	101.124	379.368		
mi3	0	0	4.180	87.516	18.315	114.884		
p3	0	0	5.562	110.544	23.574	141.012		
mi4	0	0	803	47.399	8.910	59.653		
p4	0	0	1.038	69.372	9.126	79.842		
mi_0	207.850	2.100	143.820	2.318.685	1.229.535	3.901.990		
ss	0	0	144.026	2.599.322	1.053.768	3.797.116		
cd	207.850	2.100	143.820	2.318.685	1.229.535	3.901.990		
cc	175.947	1.992	148.326	2.282.805	1.133.112	3.747.708		
ms	172.386	1.908	141.924	2.166.405	1.061.868	3.542.640		
tq	82.304	1.804	139.384	2.279.012	1.119.248	3.629.244		
GT	324.771	3.885	284.326	5.050.204	2.123.805	7.827.749	7.786.991	40.758

O dado da coluna 8 (**T(separado)**) foi calculado somando-se os tempos de simulação para todos os 5 programas executados sozinhos no simulador. O dado da coluna 9 foi obtido subtraindo-se **T(separado)** e o tempo obtido executando-se os 5 programas simultaneamente no simulador (**T(separado) - T(multi)**). O resultado obtido com este experimento não era esperado. Ocorre neste caso, o oposto ao que vinha ocorrendo quando executávamos apenas um programa simultaneamente, isto é o tempo final de simulação aumentou em 40758 *time steps*.

Tabela 3.13: Multiprogramação com 4 Programas Diferentes.

	l(gauss):	l(lapl):	l(fft):	l(mms):	4(multi):	T(separado)	T-multi
mi0	143.363	2.397.792	1.304.886	204.644	4.026.077		
p0	142.668	2.017.188	1.172.952	208.062	3.542.094		
mi1	96.250	1.503.183	926.288	125.598	2.663.089		
p1	100.134	1.332.324	834.966	133.878	2.400.672		
mi2	22.066	221.925	97.482	25.509	369.600		
p2	25.830	252.132	101.124	51.714	430.380		
mi3	4.180	87.516	18.315	11.209	123.332		
p3	5.562	110.544	23.574	24.888	165.600		
mi4	803	47.399	8.910	4.620	59.840		
p4	1.038	69.372	9.126	11.730	90.060		
mi_0	143.820	2.318.685	1.229.535	238.380	3.930.420		
ss	144.026	2.599.322	1.053.768	474.904	4.272.020		
cd	143.820	2.318.685	1.229.535	238.380	3.930.420		
cc	148.326	2.282.805	1.133.112	286.176	3.846.816		
ms	141.924	2.166.405	1.061.868	269.079	3.637.896		
tq	139.384	2.279.012	1.119.248	285.028	3.818.120		
GT	284.326	5.050.204	2.123.805	756.220	8.226.286	8.214.555	-11.731

A Tabela 3.13 apresenta os resultados obtidos com o experimento de multiprogramação para 4 programas diferentes: “Gauss”, “Laplace”, “FFT”, e “Multiplicação de Matrizes”. A descrição das linhas e colunas é semelhante à descrição apresentada para a Tabela 3.9. Nas colunas 7 e 8, estão representadas duas medidas de tempo de simulação: **T(separado)** e **T-multi** respectivamente.

A descrição das colunas 7 e 8 é idêntica à descrição das colunas 8 e 9 apresentada para a Tabela 3.12 respectivamente. Novamente obteve-se resultados não esperados ou seja, **T-multi** < 0. Os motivos que originaram estes resultados podem ser investigados usando-se os mesmos procedimentos a serem adotados para investigar o experimento da Tabela 3.12.

3.5.1 Investigação do Comportamento dos Tempos do Experimento

Um ponto inicial para investigar este comportamento seria a constatação de um aumento no *overhead* de fichas, causando atrasos e um aumento na utilização da Fila de Fichas e Memória de Dados. Este aumento se deve ao fato de que o modelo de fluxo de dados usa o critério de *eager evaluation* para escalonar fichas para serem executadas, gerando grande quantidade de fichas “prontas para serem executadas”.

O programa de “eliminação de Gauss” será utilizado para realização desta investigação. Apesar deste programa não apresentar resultados inesperados quando usado para multi-programação, ele se presta para a confirmação de que existe *overhead* devido ao mecanismo de *eager evaluation*.

Para averiguar a hipótese do aumento do *overhead* devido ao mecanismo de *eager evaluation*, o programa de “eliminação de Gauss” foi executado 5 vezes simultaneamente e comparado com a execução deste mesmo programa apenas 1 vez. A Figura 3.5 apresenta a ocupação da Memória de Dados para a execução do programa de “eliminação de Gauss” 5 vezes simultaneamente. A Figura 3.8 apresenta a ocupação da Memória de Dados para a execução do programa de “eliminação de Gauss” 1 vez apenas.

A análise das duas curvas mostra uma estreita relação entre seus formatos. A diferença

Figura 3.5: Ocupação da Memória de Dados para 5 *Gauss*.

Figura 3.6: Ocupação da Memória de Dados para 1 *Gauss*.

marcante entre elas está apenas na quantidade de fichas na Memória de Dados e no tempo total de execução do(s) programa(s). Sobrepondo as duas curvas e normalizando os valores representados nos eixos das ordenadas e abscissas, observa-se que a localização dos picos estão praticamente na mesma posição. As ocupações máximas observadas nas duas curvas são proporcionais e o fator de proporção é 5. Isto mostra que os grafos dos 5 programas são executados simultaneamente. O eixo das abscissas também mantém esta proporção. Os motivos são os já apresentados: as máquinas a fluxo de dados operam por *eager evaluation*, causando este comportamento.

A diferença na quantidade de fichas na Memória de Dados nos dois experimentos reflete o aumento na produção de fichas destinadas a instruções diádicas no experimento com 5 programas. Este aumento relaciona-se diretamente com a geração de um alto grau de paralelismo pela execução dos 5 programas se comparado à execução de apenas 1. O aumento no grau de paralelismo gerado é devido ao mecanismo de *eager evaluation* que escalona uma instrução para execução assim que seus operandos estejam disponíveis.

Figura 3.7: Ocupação da Fila de Fichas para 5 *Gauss*.

Figura 3.8: Ocupação da Fila de Fichas para 1 *Gauss*.

A Figura 3.7 apresenta a ocupação da Fila de Fichas para a execução do programa de “eliminação de Gauss” 5 vezes simultaneamente. A Figura 3.6 apresenta a ocupação da Fila de Fichas para a execução do programa de “eliminação de Gauss” 1 vez apenas. Nota-se que não existe proporção no eixo das ordenadas como observado para as duas figuras anteriores. Analisando proporcionalmente as duas curvas, verifica-se que ocorreu uma diminuição na utilização da Fila de Fichas com a execução de 5 *Gauss*.

O experimento permite concluir que o aumento no tempo de execução de vários programas simultaneamente comparado à soma dos tempos necessários para execução de um programa por vez, deve-se ao fato de haver um aumento na produção de fichas e conseqüentemente no *overhead*. Este *overhead* causa atrasos na execução dos programas devido à super-utilização das unidades do simulador.

3.6 Conclusões

Este capítulo apresentou inicialmente um mecanismo que implementa a detecção e a execução de código seqüencial na arquitetura Wolf. A implementação deste mecanismo no simulador mostrou que existe uma melhora no tempo de execução de um programa quando o mecanismo é utilizado. O experimento de balanceamento das unidades foi realizado para eliminação de unidades sobrecarregadas quando da execução do simulador. Este experimento mostrou que pode-se eliminar unidades sobrecarregadas e que o tempo (atraso) inserido para cada unidade é proporcional aos valores encontrados em processadores e memórias comerciais.

Finalmente, o experimento de multiprogramação mostra que arquiteturas a fluxo de dados precisam ser melhores quando submetidas à execução de vários programas ao mesmo tempo. O mecanismo de *eager evaluation* acaba gerando *overhead*, degradando assim o tempo de processamento. O uso de mecanismos de controle de processos como o *throttle* poderiam ser usados para melhorar o desempenho em multiprogramação.

Capítulo IV

Resultados Obtidos com o Saw

4.1 Introdução

Este capítulo apresenta os resultados obtidos com o simulador da arquitetura Wolf (Saw) já balanceado. Inicialmente apresenta resultados obtidos com a execução dos algoritmos em uma arquitetura von Neumann convencional e compara estes resultados com os resultados obtidos com o Saw. Em seguida, mostra as curvas de ocupação de memória seguidas das análises de tais curvas. Os resultados obtidos com a arquitetura von Neumann foram utilizados apenas para se ter um referencial de desempenho para a arquitetura Wolf.

4.2 Resultados

Todos os parâmetros (tamanho de matriz, número de iterações, etc) passados para os programas utilizados para obter os resultados apresentados nesta seção foram tomados de forma a maximizar o uso dos recursos disponíveis no computador hospedeiro¹. Esta

¹Computador PC com processador Pentium de 133 MHz (relógio com ciclo de 7 nano-segundos), 256 KB *Cache*, 32 MB RAM e 65 MB de *swap* em disco com Linux 2.0.30.

maximização possibilita que os recursos do simulador sejam testados exhaustivamente.

Tabela 4.1: Tempo de Execução em uma Arquitetura von Neumann e no Wolf

Algoritmo	von Neumann	Wolf (5 anéis)	Wolf (2 anéis)
Ackermann	644	2.677	2.677
Interpolação	789	5.498	7.132
Quicksort	1.553	1.099	1.342
Multiplicação de Matrizes	4.080	12.264	13.667
Somatorial	4.345	9.920	12.629
FFT	5.472	12.990	15.701
N-Queens	7.868	6.907	9.404
Gauss	8.517	1.863	2.115
Laplace	11.258	34.954	37.548
Dinâmica Molecular	200.816	193.710	226.986
Simple	658.947	520.839	575.976

Serão usados os dados apresentados na Tabela 4.1 para comparação com os apresentados nas seções subseqüentes. Esta tabela mostra os tempos em micro-segundos (10^{-6} segundos) obtidos com a execução de todos os programas usados neste trabalho em uma arquitetura von Neumann convencional e no Wolf. Os programas em SISAL foram compilados com um programa desenvolvido para geração de código em várias plataformas (PC, Sun, Cray, HP, dentre outras)². O código C gerado foi modificado de forma a produzir resultados de tempo de execução. Estes resultados foram obtidos usando-se o mesmo computador onde o simulador foi testado e os dados para este trabalho foram obtidos.

Cada ficha armazenada na Memória de Dados (MD) possui 20 *bytes* e cada ficha armazenada na Fila de Fichas (FF) possui 35 *bytes*. O tempo total de simulação será comparado ao tempo gasto por uma arquitetura von Neumann para executar os algoritmos através dos dados da Tabela 4.1. Para efeito de cálculo: 1 *time step* $\equiv$ 7 nano-segundos.

²O *software* OSC (*Optimizing SISAL Compiler*) foi desenvolvido no *Lawrence Livermore National Laboratory* e gera a partir de um programa em SISAL, código FORTRAN e/ou C.

Estes valores serão usados para cálculos de ocupação de recursos durante as simulações. Neste capítulo o termo *ciclo* será usado concomitantemente com *time step*, ambos possuindo o mesmo significado.

4.2.1 Comparação entre 5 e 2 Anéis no Saw

Esta seção apresenta os resultados obtidos com o simulador Saw executando os algoritmos com 5 anéis (coluna 3 da Tabela 4.1) e com 2 anéis (coluna 4 da Tabela 4.1). O experimento visa estudar qual é o comportamento da arquitetura se a capacidade de executar instruções for de apenas 40% (2 anéis é o mínimo para um computador paralelo) do total utilizado por este trabalho (5 anéis)³. Não se verificou mudança no tempo total para executar o programa de *Ackermann*.

Para *Interpolação*, os resultados para 5 e 2 anéis são 5.498 e 7.132 micro-segundos (μs) respectivamente. Nota-se um aumento de 1.634 μs no tempo necessário para executar o programa, representando um aumento de 1,3 vezes aproximadamente.

Para *Quicksort*, os resultados para 5 e 2 anéis são 1.099 e 1.342 μs respectivamente. Nota-se um aumento de 243 μs no tempo necessário para executar o programa, representando um aumento de 1,2 vezes aproximadamente.

Para *Multiplicação de Matrizes*, os resultados para 5 e 2 anéis são 12.264 e 13.667 μs respectivamente. Nota-se um aumento de 1.403 μs no tempo necessário para executar o programa, representando um aumento de 1,1 vezes aproximadamente.

Para *Somatorial*, os resultados para 5 e 2 anéis são 9.920 e 12.629 μs respectivamente.

³Foram utilizados 5 anéis para realização dos experimentos. Este número foi escolhido para permitir executar os algoritmos com dados de entrada "aceitáveis", já que a cada anel alocado no simulador são alocados recursos na máquina hospedeira diminuindo os recursos disponíveis para alocação das estruturas de dados exigidas pelos algoritmos.

Nota-se um aumento de $2.709 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,3 vezes aproximadamente.

Para *FFT*, os resultados para 5 e 2 anéis são 12.990 e $15.701 \mu s$ respectivamente. Nota-se um aumento de $2.711 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,2 vezes aproximadamente.

Para *N-Queens*, os resultados para 5 e 2 anéis são 6.907 e $9.404 \mu s$ respectivamente. Nota-se um aumento de $2.497 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,4 vezes aproximadamente.

Para *Gauss*, os resultados para 5 e 2 anéis são 1.863 e $2.115 \mu s$ respectivamente. Nota-se um aumento de $252 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,1 vezes aproximadamente.

Para *Laplace*, os resultados para 5 e 2 anéis são 34.954 e $37.548 \mu s$ respectivamente. Nota-se um aumento de $2.594 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,1 vezes aproximadamente.

Para *Dinâmica Molecular*, os resultados para 5 e 2 anéis são 193.710 e $226.986 \mu s$ respectivamente. Nota-se um aumento de $33.276 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,2 vezes aproximadamente.

Para *Simple*, os resultados para 5 e 2 anéis são 520.839 e $575.976 \mu s$ respectivamente. Nota-se um aumento de $55.137 \mu s$ no tempo necessário para executar o programa, representando um aumento de 1,1 vezes aproximadamente.

Nota-se que, enquanto houve uma diminuição das unidades funcionais para 40% (de 5 para 2 anéis) do total, não houve um aumento nesta mesma proporção para o tempo de processamento. Isto mostra um problema com a arquitetura Wolf: o problema de balanceamento de carga nos anéis MI-UF. Tem-se aqui um problema de *speedup*. Estes

resultados mostram que o *speedup* é baixo, sugerindo que deva haver problemas na exploração de paralelismo pela arquitetura. Estes problemas serão abordados no Capítulo 5.

4.2.2 Ackermann

Figura 4.1: Ocupação da Memória de Dados para Ackermann.

Para obtenção dos resultados foi usado $Ack(3,3)$ ⁴. Ocorreram 2.432 chamadas ao algoritmo (segundo fórmula apresentada no Capítulo 2). A Figura 4.1 apresenta a ocupação da Memória de Dados para a função de Ackermann. Nota-se um comportamento oscilatório característico deste algoritmo para a ocupação da MD. Tal comportamento se deve ao fato do algoritmo ser altamente recursivo e gerar uma árvore de processos bem ramificada. Existem três regiões oscilatórias na figura, refletindo “surto” de aumento e diminuição da ocupação da unidade.

Estes surtos refletem uma característica do algoritmo de produzir ramos da árvore

⁴Para $Ack(3,4)$ são 10.307 chamadas ao algoritmo, tornando inviável sua execução no computador hospedeiro devido à grande utilização de memória e o alto tempo de processamento.

recursiva e escalonar para execução simultânea as diversas folhas destes ramos. Tal escalonamento produz uma quantidade de fichas armazenadas na unidade e como estas fichas possuem parceira produzida também no mesmo ramo, obtém-se o comportamento oscilatório observado na Figura 4.1.

No caso de Ackermann, a Fila de Fichas possui uma ocupação “desprezível”. Sua ocupação máxima é uma ficha durante toda a simulação. Tal ficha fica armazenada por um curto período de tempo nesta unidade e, entre uma amostragem e outra (1.000 ciclos), esta ficha é enviada ao anel. Devido a este comportamento, não será mostrado o gráfico de ocupação da Fila de Fichas para o programa Ackermann. Esta ocupação reflete o baixo *overhead* de fichas devido às características do algoritmo de produzir uma árvore de processos estreita com grande dependência de dados existente entre folhas de um mesmo ramo.

O número total de fichas armazenadas na MD chega a 240 aproximadamente, totalizando 4.800 *bytes*. A FF armazena 35 *bytes*. O tempo total de processamento foi de 382.443 *time steps* que equivale a 2.677 micro-segundos (μs) aproximadamente.

4.2.3 Dinâmica Molecular

A Figura 4.2 apresenta a ocupação da Memória de Dados para Dinâmica Molecular. Os dados usados para executar o programa foram: $D = 2$; $M = 4$; $\rho = 0,45$; $T = 0,7$; $NumberOfSteps = 10$; $OutputInterval = 10$; $TableRemakeInterval = 3$; $\Delta = 0,0001$; $R_c = 2,5$; $R_v = 2,8XS$.

Pode-se observar um comportamento oscilatório com um padrão repetitivo. Estas repetições no padrão de ocupação da Memória de Dados, refletem um alto grau de

Figura 4.2: Ocupação da Memória de Dados para Dinâmica Molecular.

chamadas recursivas e construções iterativas sendo utilizadas constantemente durante todo o processo de execução do programa de Dinâmica Molecular mas com dependências de dados rapidamente sendo resolvidas. As oscilações são de tal forma que nos pontos de mínimo a quantidade de fichas na MD cai praticamente a zero. Este comportamento não é desejável, já que a MD fica sub-utilizada grande parte do tempo. Este comportamento poderia ser minimizado com o uso do *throttle*, permitindo que a unidade fosse mais uniformemente utilizada durante a execução do programa.

No caso de Dinâmica Molecular, a Fila de Fichas possui uma ocupação “desprezível”. Sua ocupação máxima é uma ficha durante toda a simulação. Tal ficha fica armazenada por um curto período de tempo nesta unidade e, entre uma amostragem e outra (10.000 ciclos), esta ficha é enviada ao anel. Devido a este comportamento, não mostraremos o gráfico de ocupação da Fila de Fichas para o programa Dinâmica Molecular. Esta ocupação reflete o baixo *overhead* de fichas devido às características do algoritmo de produzir uma árvore de processos estreita com grande dependência de dados existente

entre folhas de um mesmo ramo.

O número total de fichas armazenadas na MD chega a 76.000 aproximadamente, totalizando 1.520.000 *bytes*. A FF armazena 35 *bytes*. O tempo total de processamento foi de 27.672.917 *time steps* que equivale a 193.710 μ s aproximadamente.

4.2.4 Somatorial

Os resultados foram obtidos calculando *Somatorial*(10.000). A Figura 4.3 apresenta a ocupação da Memória de Dados para o cálculo do somatorial. Observa-se que a ocupação desta unidade acontece de forma a refletir o comportamento duplo-recursivo do algoritmo. Este tipo de algoritmo tem a característica de produzir um número grande de fichas cuja dependência de dados ultrapassa os limites de uma instanciação, ficando estas fichas armazenadas na Memória de Dados até o ponto médio da execução do programa.

Figura 4.3: Ocupação da Memória de Dados para Somatorial.

A Figura 4.4 apresenta a ocupação da Fila de Fichas para o cálculo do somatorial. Neste caso, pode-se observar uma ocupação máxima para esta unidade de aproxima-

Figura 4.4: Ocupação da Fila de Fichas para Somatorial.

mente 15.800 fichas. Este pico de ocupação se encontra aproximadamente no ponto médio da simulação (considerando-se o tempo total gasto para a execução do programa que é de 1.417.114 *time steps*).

Tal pico reflete uma característica do algoritmo utilizado. Por seu comportamento duplo-recursivo, neste ponto médio as duas sub-árvores de processos geradas possuem sua largura máxima, isto é possuem um número máximo de ramos e folhas, fazendo com que haja um pico na produção de fichas. Após este ponto médio, as dependências de dados começam a ser resolvidas e a ocupação da MD começa a diminuir. A correspondente diminuição da FF mostra que mesmo com o esvaziamento da MD, não é gerado *overhead* graças à dependência de dados entre as fichas armazenadas na FF e as armazenadas na MD.

O número total de fichas armazenadas na MD chega a 20.500 aproximadamente, totalizando 410.000 *bytes*. A FF armazena 15.800 fichas aproximadamente, totalizando 553.000 *bytes*. O tempo total de processamento foi de 1.417.114 *time steps* que equivale

a $9.920 \mu\text{s}$ aproximadamente.

4.2.5 FFT

O programa de FFT foi executado tomando-se 256 pontos. A Figura 4.5 apresenta a ocupação da Memória de Dados para a execução do cálculo da transformada rápida de Fourier (FFT). A ocupação máxima da Memória de Dados para a execução do programa de FFT é de aproximadamente 34.500 fichas e acontece aproximadamente a um terço do início do programa. Esta característica se deve ao fato de haver uma produção inicial (primeiro terço) grande de fichas destinadas a operações diádicas e a partir do segundo terço e até o final da simulação, uma tendência em consumir as fichas armazenadas na unidade.

Figura 4.5: Ocupação da Memória de Dados para FFT.

A Figura 4.6 apresenta a ocupação da Fila de Fichas para a execução do cálculo da transformada rápida de Fourier (FFT). Nota-se um pico de ocupação em aproximadamente 5.600 fichas. Como no caso da Memória de Dados, este pico se encontra também a

Figura 4.6: Ocupação da Fila de Fichas para FFT.

um terço do início do programa, refletindo uma produção de fichas intensa até este ponto. Esta produção pode ser explicada já que no código, existem instruções denominadas *proliferate* que, a partir de uma única ficha, produz um *stream* de fichas.

O número total de fichas armazenadas na MD chega a 34.500 aproximadamente, totalizando 690.000 *bytes*. A FF armazena 5.600 fichas aproximadamente, totalizando 196.000 *bytes*. O tempo total de processamento foi de 1.855.663 *time steps* que equivale a 12.990 μs aproximadamente.

4.2.6 Gauss

Para obter os resultados foi usado um sistema linear 8×8 . A Figura 4.7 apresenta a ocupação da Memória de Dados para a execução do algoritmo de Gauss com escalonamento parcial para resolução de sistemas lineares. Existem três regiões de pico na figura: um pico em aproximadamente 5.200 fichas, outro pico em aproximadamente 1.500 fichas e um terceiro pico em aproximadamente 50 fichas.

Estes picos refletem a produção de fichas destinadas a operações diádicas. Tais picos originam-se devido à dependência de dados existente no algoritmo. Existe portanto uma tendência do algoritmo de produzir fichas de vida longa e que podem esperar vários ciclos para serem consumidas.

Figura 4.7: Ocupação da Memória de Dados para Gauss.

A Figura 4.8 apresenta a ocupação da Fila de Fichas para a execução do algoritmo de Gauss com escalonamento parcial para resolução de sistemas lineares. Nota-se pela figura que existem inúmeros picos de ocupação para a Fila de Fichas.

Isto se deve principalmente pela grande quantidade de instruções do tipo *proliferate* existentes no código. A figura mostra também que apesar da ocupação grande da Memória de Dados (Figura 4.7), não ocorre uma ocupação muito significativa da Fila de Fichas, refletindo o comportamento de que quase tudo que é produzido pelos anéis, são processados pela Memória de Dados quase que simultaneamente.

O número total de fichas armazenadas na MD chega a 5.200 aproximadamente, totalizando 104.000 *bytes*. A FF armazena 480 fichas aproximadamente, totalizando 16.800

Figura 4.8: Ocupação da Fila de Fichas para Gauss.

bytes. O tempo total de processamento foi de 266.220 *time steps* que equivale a 1.863 micro-segundos aproximadamente.

4.2.7 Interpolação

Para obtenção dos resultados foi utilizado uma tabela com 14 pontos e o polinômio interpolador de grau máximo (13). A Figura 4.9 apresenta a ocupação da Memória de Dados para a execução do algoritmo de Interpolação Polinomial. Os vários picos na ocupação da Memória de Dados implicam numa dependência de dados “oscilatória” durante toda a execução do programa. Uma possibilidade para explicar este comportamento é que existem chamadas recursivas e/ou construções iterativas que geram fichas cujas parceiras não são produzidas no mesmo nível da iteração e/ou recursão.

A Figura 4.10 apresenta a ocupação da Fila de Fichas para a execução do algoritmo de Interpolação Polinomial. Nota-se uma particularidade nesta figura: existe uma correspondência entre os picos de ocupação da Memória de Dados (Figura 4.9) e os picos de

Figura 4.9: Ocupação da Memória de Dados para Interpolação.

Figura 4.10: Ocupação da Fila de Fichas para Interpolação.

ocupação da Fila de Fichas, inclusive as duas curvas são muito semelhantes na forma e pontos de oscilação.

Uma explicação para esta característica das duas curvas é que existe uma estreita correspondência entre a produção de fichas com vida longa e a execução de instruções “proliferate”. Esta característica reflete a tendência do algoritmo em usar construções iterativas para resolver a interpolação. O código fonte que mostra esta tendência encontra-se no Apêndice A.

O número total de fichas armazenadas na MD chega a 3.400 aproximadamente, totalizando 68.000 *bytes*. A FF armazena 1.700 fichas aproximadamente, totalizando 59.500 *bytes*. O tempo total de processamento foi de 785.470 *time steps* que equivale a 5.498 μ s aproximadamente.

4.2.8 Laplace

Foi utilizada uma matriz 20X20 para obtenção dos resultados. A Figura 4.11 apresenta a ocupação da Memória de Dados para a execução da função de Laplace. Temos um pico de ocupação desta unidade em aproximadamente metade do tempo total de simulação, com 80.000 fichas. Salvo pequenas variações observadas, a curva reflete uma dependência de dados que só começa a ser resolvida após a primeira metade da execução do programa. Não se pode afirmar que a dependência de dados após 2.500.000 ciclos seja pequena, mas o gráfico indica que antes deste tempo, a dependência de dados aparenta ser maior, gerando muitas fichas de vida longa.

A Figura 4.12 apresenta a ocupação da Fila de Fichas para a execução da função de Laplace. Pode-se dividir esta curva em duas regiões distintas. Na primeira região,

Figura 4.11: Ocupação da Memória de Dados para Laplace.

Figura 4.12: Ocupação da Fila de Fichas para Laplace.

observa-se um comportamento oscilatório irregular que vai do início até aproximadamente a metade da simulação. Na segunda região, observa-se um comportamento oscilatório regular que vai até o final da execução do programa.

Considerando a ocupação da Memória de Dados (MD) (Figura 4.11, nota-se que o comportamento irregular da ocupação da Fila de Fichas acontece durante o aumento da ocupação da MD (primeira metade da simulação). A partir deste ponto (segunda metade da simulação, quando a MD começa a ser esvaziada), a Fila de Fichas começa a ter um comportamento “regular”. Este fato deve-se a uma regularidade no algoritmo utilizado.

O número total de fichas armazenadas na MD chega a 80.000 aproximadamente, totalizando 1.600.000 *bytes*. A FF armazena 6.500 fichas aproximadamente, totalizando 227.500 *bytes*. O tempo total de processamento foi de 4.993.382 *time steps* que equivale a 34.954 μ s aproximadamente.

4.2.9 Multiplicação de Matrizes

Foi multiplicada uma matriz 20x20 para obtenção dos resultados. A Figura 4.13 apresenta a ocupação da Memória de Dados para a execução do algoritmo de multiplicação de matrizes. Na figura observa-se um pico de ocupação da unidade em aproximadamente 400.000 ciclos. Até este ponto há um constante aumento na ocupação e a partir deste ponto, a unidade começa a ser esvaziada.

Em torno de 1.600.000 ciclos, existe uma quebra na tendência de desocupação da unidade, ocasionando a partir deste ponto, uma desocupação mais acentuada. Considerando o algoritmo utilizado (o código fonte pode ser visto no Apêndice A), pode-se dizer que é possível que em torno de 1.600.000 ciclos, a multiplicação das duas matrizes

efetivamente termina, e os últimos ciclos ficando a cargo da construção da matriz resultante e apresentação dos resultados.

Figura 4.13: Ocupação da Memória de Dados para Multiplicação de Matrizes.

A Figura 4.14 apresenta a ocupação da Fila de Fichas para a execução do algoritmo de multiplicação de matrizes. Na figura notam-se dois picos distintos. O primeiro em aproximadamente 250.000 ciclos e o segundo em aproximadamente 1.100.000 ciclos. A ocupação máxima da Fila de Fichas está em torno de 23.000 fichas. Uma característica marcante desta curva é o ponto em 400.000 ciclos onde a ocupação cai para quase zero fichas e logo após sobe novamente para a ocupação máxima da unidade.

Uma forma de explicar este comportamento, seria supor que a execução do programa de 0 a 400.000 ciclos gera muitas fichas cuja dependência de dados não é resolvida de imediato (fichas de vida longa). Isto se reflete na crescente ocupação da Memória de Dados (MD) (Figura 4.13) até 400.000 ciclos. Existe portanto uma transferência de fichas da Fila de Fichas para a MD a partir de 250.000 até 400.000 ciclos aproximadamente. Esta transferência ocorre devido à existência de um gargalo no algoritmo. Apenas em torno de

Figura 4.14: Ocupação da Fila de Fichas para Multiplicação de Matrizes.

400.000 ciclos é que começam a ser produzidas fichas para resolver esta dependência.

A partir de 400.000 ciclos, existe uma produção maior de fichas do que a MD consegue tratar causando um novo aumento da utilização da Fila de Fichas. Note-se que apesar do aumento da ocupação desta unidade, a MD continua sendo desocupada. Por conseguinte, pode-se concluir que a partir do ponto de 400.000 ciclos, existe uma tendência do algoritmo em resolver dependências de dados geradas até este ponto.

O número total de fichas armazenadas na MD chega a 43.000 aproximadamente, totalizando 860.000 *bytes*. A FF armazena 23.000 fichas aproximadamente, totalizando 805.000 *bytes*. O tempo total de processamento foi de 1.751.936 *time steps* que equivale a 12.264 μs aproximadamente.

4.2.10 N-Queens

Um tabuleiro 6x6 foi usado para obtenção destes resultados. A Figura 4.15 apresenta a ocupação da Memória de Dados para a execução do algoritmo das N rainhas no ta-

buleiro de xadrez. Na figura observa-se quatro picos de ocupação. O primeiro pico em aproximadamente 2.100 fichas e 90.000 ciclos. O segundo pico em 4.700 fichas e 220.000 ciclos aproximadamente. O terceiro em 7.000 fichas e 490.000 ciclos aproximadamente.

O quarto pico em aproximadamente 7.500 fichas e 800.000 ciclos. Da curva conclui-se que o algoritmo utilizado para resolver o problema das N rainhas gera, em pelo menos quatro momentos diferentes durante a execução do programa, fichas cuja dependência de dados é quase que imediatamente resolvida. Como exemplo, considere-se o segundo pico: entre 180.000 e 300.000 ciclos, aproximadamente 2.000 fichas diádicas são produzidas e consumidas neste intervalo. Consideração análoga pode ser feita para os outros picos. Finalmente, notam-se protuberâncias entre cada pico representando pequenas dependências de dados.

Figura 4.15: Ocupação da Memória de Dados para N-Queens.

A Figura 4.16 apresenta a ocupação da Fila de Fichas para a execução do algoritmo das N rainhas no tabuleiro de xadrez. Há duas formas de interpretar a curva da Figura 4.16. Na primeira, que será a utilizada neste trabalho, considere-se apenas quatro picos de

Figura 4.16: Ocupação da Fila de Fichas para N-Queens.

ocupação. Na segunda, considere-se dez picos de ocupação. A primeira forma foi escolhida para permitir comparação com a curva de ocupação da Memória de Dados (Figura 4.15). Os quatro picos da figura ocorrem quase que simultaneamente com os picos de ocupação da MD.

Isto pode ser explicado considerando-se que existe uma produção acentuada de fichas nestes intervalos de pico e que estas fichas são destinadas a operações com dois operandos. As protuberâncias existentes entre cada pico também podem ser observadas na curva de ocupação da Memória de Dados. Estas protuberâncias se devem ao fato do algoritmo utilizado gerar fichas cuja dependência de dados é rapidamente resolvida.

O número total de fichas armazenadas na MD chega a 7.500 aproximadamente, totalizando 150.000 *bytes*. A FF armazena 3.800 fichas aproximadamente, totalizando 133.000 *bytes*. O tempo total de processamento foi de 986.652 *time steps* que equivale a 6.907 μs aproximadamente.

4.2.11 Quicksort

Foi ordenado um vetor com 20 elementos para obtenção dos resultados. A Figura 4.17 apresenta a ocupação da Memória de Dados para a execução do algoritmo de ordenação Quicksort. Na figura, observa-se um comportamento oscilatório com vários picos de ocupação, sendo que em nenhum deles é ultrapassada a ocupação máxima de 350 fichas.

A partir de 250 fichas de ocupação, cada pico se divide em dois, mostrando que parte da dependência de dados que cada pico reflete, é resolvida quase que imediatamente após sua geração. Durante grande parte da simulação (após 40.000 ciclos), a ocupação da Memória de Dados nunca cai para menos de 100 fichas (salvo no final da simulação). Isto reflete a característica do algoritmo de possuir uma dependência de dados que só é resolvida no final da simulação.

Figura 4.17: Ocupação da Memória de Dados para Quicksort.

A Figura 4.18 apresenta a ocupação da Fila de Fichas para a execução do algoritmo de ordenação Quicksort. A ocupação máxima da unidade é de aproximadamente 230 fichas.

Figura 4.18: Ocupação da Fila de Fichas para Quicksort.

Ocorre aqui também uma ocupação oscilatória da Fila de Fichas, sendo que em cada pico a ocupação da unidade chega quase em zero. Tais picos se originam devido à presença de instruções “proliferate” no código executado pelo simulador.

Comparando-se as duas curvas (Figura 4.17 e Figura 4.18), nota-se uma semelhança entre elas: as duas curvas possuem picos “duplos”. Esta característica da segunda curva vem apoiar o que foi dito para o comportamento da primeira curva: estes picos são gerados pela execução de instruções “proliferate”, já que a execução de instruções deste tipo provoca um alto tráfego de fichas pela Chave de Coleta, geralmente provocando uma grande demanda de fichas destinadas à Memória de Dados.

O número total de fichas armazenadas na MD chega a 340 aproximadamente, totalizando 6.800 *bytes*. A FF armazena 230 fichas aproximadamente, totalizando 8.050 *bytes*. O tempo total de processamento foi de 157.026 *time steps* que equivale a 1.099 μ s aproximadamente.

4.2.12 Simple

Para obtenção dos resultados foram usados: $NCycles = 10$ e $ArraySize = 16$. A Figura 4.19 apresenta a ocupação da Memória de Dados para a execução de Simple. O número máximo de fichas na Memória de Dados é de aproximadamente 84.000 fichas. Nota-se uma ocupação oscilatória e periódica na curva de ocupação da Memória de Dados. Ao final de cada período, a ocupação cai praticamente para zero. Esta característica reflete uma dependência de dados que é resolvida a cada 8.000.000 ciclos aproximadamente (duração de cada período), durante a simulação.

Figura 4.19: Ocupação da Memória de Dados para Simple.

A Figura 4.20 apresenta a ocupação da Fila de Fichas para a execução de Simple. A ocupação máxima para a unidade é de aproximadamente 27.500 fichas. Nota-se um comportamento oscilatório e um período de aproximadamente 8.000.000 ciclos entre cada oscilação. Os picos de ocupação são provocados pela execução de instruções do tipo “proliferate”. Existe uma variação intensa na ocupação em todos os intervalos entre 0 e

Figura 4.20: Ocupação da Fila de Fichas para Simple.

5.000 fichas. Uma hipótese para explicar este comportamento seria a execução de fichas “proliferate” gerando um pequeno *stream* de fichas.

Comparando-se as duas curvas (Figura 4.19 e Figura 4.20), nota-se uma semelhança entre o número de períodos (10), o momento em que cada período ocorre e também no tamanho de cada período (8.000.000 ciclos). As pequenas oscilações que estão em torno de 35.000 fichas na curva de ocupação da Memória de Dados (MD) e as variações intensas na ocupação da Fila de Fichas (FF) entre 0 e 5.000 fichas, ocorrem quase que simultaneamente na simulação, levando-se a acreditar que tais variações na FF são devido à produção de fichas diádicas cuja parceira foi ou será quase que imediatamente produzida (fichas cuja dependência de dados não é muito longa).

O número total de fichas armazenadas na MD chega a 84.000 aproximadamente, totalizando 1.680.000 *bytes*. A FF armazena 27.500 fichas aproximadamente, totalizando 962.500 *bytes*. O tempo total de processamento foi de 74.405.592 *time steps* que equivale a 520.839 μs aproximadamente.

4.3 Conclusões

As curvas obtidas mostram que existe uma relação entre as características dos algoritmos utilizados e a utilização dos recursos da arquitetura Wolf. Os dados obtidos com a execução dos programas em uma arquitetura von Neumann padrão mostram que o Wolf é mais rápido em alguns programas e mais lento em outros. O desempenho da arquitetura é profundamente dependente das características intrínsecas do algoritmo empregado. Apesar da arquitetura Wolf ter sido proposta para resolver problemas de uso geral, poderão ser identificadas características particulares destes algoritmos às quais o Wolf trata mais eficientemente.

Os resultados obtidos neste capítulo mostram que a arquitetura Wolf ainda precisa ser muito mais eficiente na execução de códigos seqüenciais. O mecanismo de detecção de código seqüencial ajuda, mas não resolve o problema por completo. Este mecanismo não está explorando toda a potencialidade da dependência de dados nos grafos a fluxo de dados. Por exemplo, quando dois ramos possuem trechos seqüenciais, apenas um deles é escalonado pelo mecanismo para execução imediata, o outro é enviado ao anel. A implementação de mecanismos tanto “a nível de” *hardware* quanto “a nível de” *software* poderia melhorar o desempenho da arquitetura.

Capítulo V

Distribuição de Carga e a Arquitetura Wolf II

5.1 Introdução

Este capítulo estuda o problema de distribuição de carga nos anéis MI-UF no Wolf. Esta distribuição não é homogênea para todos os anéis, provocando uma sobrecarga nos primeiros anéis. Serão apresentadas várias hipóteses para explicar este problema. Para cada hipótese serão apresentados também alguns resultados de maneira a comprovar a validade ou não de tal hipótese. Como consequência dos resultados obtidos, serão apresentadas algumas propostas para alteração da arquitetura de forma que o problema de distribuição seja minimizado.

5.2 Problemas de Distribuição de Carga

A arquitetura Wolf possui um mecanismo totalmente dependente da Chave de Distribuição (CD), responsável pela distribuição das fichas que chegam até à CD para os vários anéis Memória-Processador (MI-UF). Seu funcionamento é: a CD ao receber uma

ficha, identifica para qual dos três anéis (Memória de Estruturas, Unidade de Controle de Paralelismo ou Memória de Instruções) a ficha se destina.

As curvas de distribuição de carga obtidas com as simulações não corresponderam às expectativas. Esperaria-se obter uma distribuição de carga mais uniforme entre cada um dos anéis MI-UF e o que encontrou-se foi uma carga bem maior do primeiro anel. Os resultados da Figura 5.1 mostram isto. Esta distribuição não uniforme ocasiona o aparecimento de unidades sobrecarregadas na arquitetura e isto não é desejável.

Os resultados foram obtidos utilizando o programa de “eliminação de Gauss”. Tais resultados são sintomáticos do comportamento aferido nos demais algoritmos pela arquitetura. No restante deste trabalho apresentar-se-ão apenas os resultados com *Gauss*.

Figura 5.1: Exemplo de utilização dos anéis.

Na Figura 5.1 estão representados no eixo das abscissas o anel a ser considerado. No eixo das ordenadas, o número de instruções executadas. A porcentagem de utilização do segundo anel em relação ao primeiro anel é de 67,4%. O terceiro anel foi menos utilizado ainda (26,2% em relação ao primeiro). O quinto e último anel, 2,8% do primeiro. Na

próxima seção serão apresentadas algumas hipóteses para a má distribuição de carga nos anéis.

Aparentemente o problema de distribuição de carga apresentado pela arquitetura Wolf não é proveniente do modelo de fluxo de dados do qual ela se baseia. As experiências de diversos grupos mostram que, devido à sua natureza, o modelo de fluxo de dados possui boas características de distribuição de carga entre os processadores [14, 27]. O desempenho apresentado no capítulo anterior pode ser devido às ineficiências da arquitetura Wolf no tratamento de estruturas de dados e no controle de recursos do sistema. Nas seções seguintes serão discutidos os pontos que nos levaram à estas conclusões.

5.3 Hipóteses para a má Distribuição de Carga

Esta seção apresenta quatro hipóteses para a má distribuição de carga entre os anéis MI-UF da arquitetura Wolf:

1. Comportamento Anômalo da Fila de Fichas;
2. Comportamento Anômalo da Chave de Coleta;
3. Comportamento Anômalo da Temporização;
4. Variação na Velocidade da Memória de Instruções Interna;

5.3.1 Comportamento Anômalo da Fila de Fichas

Uma hipótese para explicar os resultados seria um mau comportamento da Fila de Fichas (FF) por esta estar centralizada, não sendo capaz de atender a todos os anéis ao

mesmo tempo. Esta incapacidade poderia privilegiar o primeiro anel em detrimento do segundo e assim sucessivamente. Tal comportamento não se verifica, visto que é a Chave de Coleta (CC) a responsável pelo recebimento das fichas dos anéis e portanto qualquer anomalia neste sentido seria devido a CC e não à FF. A hipótese de anomalias na CC será averiguada na subseção seguinte.

O experimento de descentralização da FF mostra a verificação de ausências de anomalias na FF. A Figura 5.2 mostra como ficaria a arquitetura Wolf com a FF descentralizada.

Figura 5.2: Arquitetura Wolf com a FF descentralizada.

Para a realização do experimento, a FF foi desativada e implementou-se FF's em todas as entradas da CC. Se a hipótese de anomalia na FF estivesse correta, obter-se-ia uma distribuição mais uniforme entre os anéis. Os resultados da Figura 5.3 mostram que isto efetivamente não ocorre.

Figura 5.3: Distribuição de Carga com Descentralização da FF.

Pode-se notar que a descentralização da FF não melhora significativamente a distribuição de carga nos anéis MI-UF mesmo com uma FF por anel. A relação entre o número de instruções executadas pelo primeiro anel MI-UF e o número total de instruções é 40,5%. A relação entre o número de instruções executadas pelo segundo anel MI-UF e o número total de instruções é 30,9%. Estes valores estão acima de 20,0% esperado para cada anel.

A relação entre o número de instruções executadas pelo segundo anel e o número de instruções executadas pelo primeiro anel é de 76,4%, isto é o segundo anel executou 76,4% do número de instruções executadas pelo primeiro anel (com a FF centralizada, obteve-se 67,4%). Os resultados não melhoram à medida que os outros anéis são acrescentados. Portanto, com base nestes resultados, conclui-se que estes dados não são aceitáveis para uma boa distribuição de carga. A Tabela 5.1 apresenta todos os dados obtidos com o experimento.

Com este resultado pode-se afirmar que a FF centralizada não é a causa da distribuição

Tabela 5.1: Distribuição de Carga nos Anéis MI-UF.

Anel	Número de Instruções Executadas	% do Total
1	9.189	40,5
2	7.020	30,9
3	3.744	16,5
4	1.871	8,2
5	874	3,8
Total	22.698	-

de carga apresentada na Figura 5.1, e que portanto a hipótese é inválida como foi mencionado no início desta discussão.

5.3.2 Comportamento Anômalo da Chave de Coleta

Uma outra hipótese para explicar os resultados obtidos é que o mecanismo usado na Chave de Coleta (CC) para enviar as fichas de volta ao anel não se comporta como o esperado, dando prioridade para fichas originadas no primeiro anel. Isto faz com que haja uma grande utilização deste anel. O mecanismo deveria se comportar da seguinte forma: a CC deveria identificar a ficha em suas entradas que tivesse o menor tempo e se a saída para a MD estivesse desocupada, enviá-la de volta ao anel. Se a saída estivesse ocupada, a ficha seria enviada para a fila de fichas (FF).

Tabela 5.2: Comportamento da CC.

Tempo das fichas vindas dos anéis (<i>time steps</i>)	7.082 7.077 7.074 7.084 7.084 7.074(FF)
Tempo da ficha enviada ao anel (<i>time steps</i>)	7.074
Tempo da ficha enviada à FF (<i>time steps</i>)	7.077

O problema então consistiria em verificar se esta prioridade estava sendo obedecida.

Os resultados apresentados na Tabela 5.2 mostram um instante da execução do programa de “eliminação de Gauss”. As cinco primeiras fichas são oriundas dos anéis MI-UF e a sexta ficha (7.074) é oriunda da FF. Esta última ficha nunca é enviada novamente à FF. Não se faz distinção entre duas fichas com mesmo tempo, e a ficha enviada de volta ao anel é sempre a ficha que foi primeiramente tratada (no nosso caso, a ficha oriunda do anel MI-UF).

Através dos resultados apresentados na Tabela 5.2, nota-se que a CC comporta-se como o esperado, dando prioridade à ficha de menor tempo. Portanto, a CC está funcionando com as prioridades corretas e a hipótese é inválida.

5.3.3 Comportamento Anômalo da Temporização

A terceira hipótese é que o mecanismo de temporização implementado não estava funcionando como o esperado, causando uma anomalia no mecanismo usado na CC para identificar fichas de menor tempo. O comportamento do mecanismo de temporização pode ser resumido da seguinte maneira: existe um relógio global no simulador que serve para controlar a utilização de fichas durante as simulações e um campo tempo em cada ficha produzida. Uma ficha é considerada pronta para ser consumida se e somente se o seu campo tempo possuir um valor igual ou inferior ao valor do relógio global.

Tabela 5.3: Comportamento da Temporização.

Tempo da ficha na entrada da CC (<i>time steps</i>)	7.074
Tempo da ficha resultante (<i>time steps</i>)	7.091
Relógio Global (<i>time steps</i>)	7.086

O experimento considera um *pipeline* e portanto cada unidade demora um tempo

característico (discutido no Capítulo 3) para executar uma ficha em sua entrada e soltar um resultado em sua saída.

Portanto, se o atraso de uma unidade é cinco *time steps* e o valor do relógio global é, em um determinado momento igual a T , e finalmente se o tempo da ficha na entrada de uma unidade é também igual ou menor que T , então esta ficha está pronta para ser executada e o resultado produzido será uma ficha com tempo igual a $T + 5$. A Tabela 5.3 mostra um exemplo de experimento para verificar problemas com a temporização. Tal experimento mostra que o mecanismo de temporização está funcionando como esperado e portanto a hipótese é inválida.

5.3.4 Variação na Velocidade da Memória de Instruções Interna

Como última hipótese, considere-se que haja um desbalanceamento entre a velocidade da Memória de Instruções Interna (MI) e a velocidade das Unidades Funcionais (UF). Numa primeira análise esta hipótese parece não proceder já que na seção 3.4 é descrito um experimento que balanceia todas as unidades da arquitetura, eliminando unidades sobrecarregadas e unidades sub-utilizadas (*hot spots* e *cold spots*).

O experimento a seguir explora os limites de velocidade da MI, mostrando que existe fundamento na hipótese do desbalanceamento entre a velocidade da MI e da UF. Para desenvolvê-lo, as unidades do simulador serão divididas em dois grupos: o primeiro grupo consistindo das MI's, e o segundo grupo das unidades restantes.

O tempo característico de cada unidade do segundo grupo permanece sempre constante enquanto que o tempo característico das unidades do primeiro grupo (MI's) será aumentado a cada passo do experimento, introduzindo um atraso nestas unidades. O

objetivo será mostrar através de um gráfico, que a partir de um determinado valor para o tempo das unidades do primeiro grupo, a distribuição de carga permanece praticamente constante. Os dados apresentados na Figura 5.4 mostram os resultados deste experimento. Este é um gráfico envolvendo três grandezas que se inter-relacionam: **Anel**, **Velocidade**, e **Número de Instruções**.

Figura 5.4: Variação do tempo da MI.

O eixo **Anel** representa cada anel envolvido no processamento (neste experimento usa-se cinco anéis MI-UF). O eixo **Velocidade**, representa o fator de aumento do tempo (toma-se como velocidade base da MI, os resultados apresentados no Capítulo 3) das unidades do primeiro grupo (MI's). O eixo **Número de Instruções** representa a quantidade de fichas executadas pelos anéis. A Figura 5.4 é composta de quatro conjuntos de experimentos (tome-se o eixo **Velocidade** como referência).

No primeiro conjunto, o tempo das MI's é duas vezes maior que o tempo inicial de 11 *time steps* obtido no Capítulo 3. O número de instruções executadas no primeiro até

o quinto anel é respectivamente 7.542, 6.286, 4.839, 2.762 e 1.269, totalizando 22.698 instruções. O segundo anel corresponde a 83,3% do primeiro anel ($6.286 * 100 \div 7.542$) e o quinto anel corresponde a 16,8% do primeiro anel. Em termos de distribuição de carga entre os anéis, estes resultados ainda deixam muito a desejar (está-se procurando uma distribuição praticamente uniforme entre cada anel). Com base nesta conclusão, o experimento pode ser continuado.

No segundo conjunto, o tempo das MI's é cinco vezes maior que o tempo inicial de 11 *time steps*. O número de instruções executadas no primeiro anel até o quinto anel é respectivamente 5.143, 4.753, 4.531, 4.231 e 4.040. Com estes dados, calculou-se que o segundo anel corresponde a 92,4% do primeiro anel e que o quinto anel corresponde a 78,5% do primeiro anel. Em termos de distribuição de carga e os objetivos do experimento conclui-se que o valor aceitável para a distribuição de carga está convergindo para um valor aceitável.

No terceiro conjunto de experimentos, o tempo das MI's é dez vezes maior que o tempo inicial de 11 *time steps*. O número de instruções executadas no primeiro anel até o quinto anel é respectivamente 4.892, 4.662, 4.443, 4.458 e 4.243. Com estes dados, calculou-se que o segundo anel corresponde a 95,3% do primeiro anel e que o quinto anel corresponde a 86,7% do primeiro anel.

No quarto e último experimento, o tempo das MI's é vinte vezes maior que o tempo inicial de 11 *time steps*. O número de instruções executadas no primeiro anel até o quinto anel é respectivamente 4.747, 4.647, 4.507, 4.421 e 4.376. Com estes dados, calculou-se que o segundo anel corresponde a 97,9% do primeiro anel e que o quinto anel corresponde a 92,2% do primeiro anel.

A Tabela 5.4 apresenta o resultado do balanceamento de todas as unidades após o ex-

perimento descrito nesta seção. Nota-se aqui o surgimento de três categorias de unidades baseadas na porcentagem do tempo total que a unidade ficou ativa. Na primeira categoria estão as Memória de Instruções Interna (MI^*); na segunda categoria estão todos os processadores (UF^*); finalmente, na terceira categoria estão o restante das unidades (X°).

Tabela 5.4: Utilização de cada unidade.

Unidade	$mi0^*$	$p0^*$	$mi1^*$	$p1^*$	$mi2^*$	$p2^*$	$mi3^*$	$p3^*$
% Ativ.	97,1	4,7	93,9	4,6	90,7	4,4	89,0	4,2
Unidade	$mi4^*$	$p4^*$	mi_0°	ss°	cd°	cc°	ms°	tq°
% Ativ.	87,2	4,2	10,3	12,2	10,3	11,1	10,9	13,0
	$Média^*$	91,6	$Média^*$	4,4	$Média^\circ$	11,3		
	DPD^*	3,9	DPD^*	0,2	DPD°	1,2		
	$\% \%$	4,3	$\% \%$	5,2	$\% \%$	10,6		

5.3.4.1 Análise dos Resultados

O experimento descrito na seção 5.3.4 permite concluir que é possível melhorar a distribuição de carga entre os anéis MI-UF se a velocidade da MI for maior que cinco vezes a velocidade inicial de 11 *time steps* obtida no Capítulo 3.

O resultado do balanceameto das unidades após o experimento de ajuste da distribuição de carga leva a refletir sobre o comportamento geral da arquitetura. Atendo-se aos resultados de % de atividade das unidades da terceira categoria (X°), pode-se observar algumas relações entre estes resultados. Por exemplo, se o coeficiente de variação ($\% \%$) desta categoria for observado, percebe-se que este é superior a 10%, sugerindo que deva existir uma subdivisão das unidades desta categoria em sub-categorias derivadas desta.

Existe uma divisão clara para as unidades da terceira categoria. Chamando de “Unidades Seriais” (US) as unidades MI0, CD, CC e MS por estas estarem dispostas de forma seqüencial (uma após a outra, seqüencialmente) na arquitetura e chamando de “Unidades não Seriais” (UnS) as unidades SS e TQ, refaz-se os cálculos para a média, dpd e %% para estas duas sub-categorias de unidades, encontra-se: $Média^{US} = 10,6\%$, $DPD^{US} = 0,4$ e $\%^{US} = 3,8$. $Média^{UnS} = 12,6\%$, $DPD^{UnS} = 0,6$ e $\%^{UnS} = 4,8$. Estes dados mostram que a divisão da terceira categoria em duas sub-categorias (US e UnS) é fundamental para entender o comportamento das unidades envolvidas.

5.4 Justificativa para a Mudança na Arquitetura Wolf

Para que haja uma boa distribuição de carga entre os anéis MI-UF, é necessário que as MI's sejam mais lentas que as demais unidades (20 vezes o valor de 11 *time steps* encontrado no Capítulo 3). Com esta imposição, ocorrem efeitos indesejáveis como por exemplo a diminuição da utilização dos processadores (média de 4,4% como mostra a Tabela 5.4). É claro que a idéia inicial de ter todas as unidades trabalhando praticamente o mesmo número de ciclos, não pode ser considerada quando adota-se estes tempos para as MI's, aparecendo novamente unidades super-utilizadas e unidades sub-utilizadas (*hot spot* e *cold spot*).

Parece não haver uma solução satisfatória que compreenda igualdade de utilização das unidades e balanceamento de carga dos anéis MI-UF simultaneamente. Um meio termo também não é satisfatório, já que o balanceamento dos anéis começa a ser aceitável para velocidade das MI's de 10 vezes a velocidade original, fazendo com que a porcentagem de utilização dos processadores (UF) deixe a desejar. Isto não é aceitável, necessitando de

uma proposta para solucionar o problema.

Uma suposição aceitável seria dizer que no simulador com as unidades balanceadas visto no Capítulo 3, o conjunto MI-UF é muito eficiente se comparado às outras unidades, já que as outras unidades não conseguem suprir com fichas todos os anéis de forma adequada. Esta ineficiência da arquitetura no que compreende o balanceamento de carga nos anéis MI-UF, pode ser explicada considerando-se que um dado produzido em uma UF deva passar pela CC, (FF), MD, MI0, CD e MI até chegar à UF novamente para ser executado, gerando uma latência considerável (latência devido ao tamanho do *pipeline*). É lógico que esta latência é inerente à arquitetura utilizada, podendo ser usada como ponto principal para explicar a baixa utilização dos anéis MI-UF.

Com base nos resultados obtidos com os experimentos da seção 5.3 e nas discussões desta seção, a próxima seção apresenta uma nova proposta para a arquitetura a fluxo de dados Wolf que será chamada de **Wolf II**.

5.5 A Arquitetura Wolf II

Uma forma de diminuir a latência discutida na seção 5.4, é diminuir a quantidade de “estágios” que o dado percorre no *pipeline*. Para isto é necessário redefinir alguns pontos da arquitetura Wolf e alguns mecanismos já previamente discutidos. Obviamente não serão investigadas a fundo todas as características da nova arquitetura a ser proposta, como foi feito para a arquitetura Wolf, mas apenas discutir alguns pontos essenciais para guiar futuros trabalhos. A Figura 5.5 mostra a proposta para a arquitetura Wolf II.

Observando a Figura 5.5, nota-se algumas modificações na arquitetura original Wolf (Figura 2.1):

Figura 5.5: A Arquitetura Wolf II.

1. **Memória de Estruturas - ME:** A ME foi reposicionada entre a Chave de Coleta e a Memória de Dados.
2. **Memória de Instruções Externa - MI0:** A MI0 foi eliminada da nova arquitetura.
3. **Chave de Coleta - CC:** A CC possui agora duas saídas a mais. Uma para a Memória de Estruturas e a outra para a Unidade de Controle de Paralelismo.
4. **Unidade de Controle de Paralelismo - THR:** O THR foi reposicionado entre a Chave de Coleta e a Memória de Dados.

Na seção 5.5.1, será apresentada uma breve descrição funcional da arquitetura Wolf II.

5.5.1 Descrição Funcional

Fichas são inseridas no Wolf II pela via entrada (IN) da Chave de Coleta (CC). Da Chave de Coleta, a ficha vai para a Memória de Dados (MD) se esta estiver disponível.

Caso a MD não esteja disponível, a ficha vai para a Fila de Fichas (FF) onde é armazenada para ser posteriormente enviada para a MD. Na MD podem ocorrer duas situações: a ficha que chega é destinada a uma operação monádica (apenas um operando é necessário) ou a ficha que chega é destinada a uma operação diádica (dois operandos são necessários).

No primeiro caso, a ficha emparelha com uma ficha nula e é enviada para a Chave de Distribuição. No segundo caso, a ficha é armazenada na MD se o outro operando ainda não está presente ou a ficha emparelha com o outro operando e é enviada para a Chave de Distribuição. Os mecanismos que permitem o emparelhamento são idênticos aos descritos para a Memória de Dados da arquitetura Wolf.

Na Chave de Distribuição (CD) as fichas são distribuídas para o primeiro anel MI-UF que estiver livre. A Memória de Instruções armazena os detalhes do nó para o qual as fichas são dirigidas, e cria a partir do pacote que chega através da CD, um pacote executável com todas as informações necessárias para a execução da instrução enviando-o à sua Unidade Funcional associada.

Os resultados produzidos pela unidade funcional podem ser do tipo *livre* ou *encadeado*. Resultados livres são passados para a Chave de Coleta recirculando no pipeline. Resultados encadeados permanecem nos registradores internos da Unidade Funcional para serem, num estágio posterior, usados como operandos de instruções subseqüentes.

Um dos detalhes dos nós armazenados pela Memória de instruções é a informação de que um dado nó é parte de uma cadeia. Quando um pacote executável de uma cadeia é produzido, a Memória de instruções inicia por conta própria, um ciclo de busca usando o endereço de destino do pacote executável produzido como endereço de busca. Desta forma, quando a Unidade Funcional termina o consumo do pacote executável anterior, um novo pacote executável está em condição de ser produzido pela Memória de Instru-

ções. Este novo pacote executável usará os resultados intermediários do pacote executável anterior como entrada.

Os dados que chegam até à MI não são necessariamente destinados à instruções cuja UF seja capaz de executar. Existem também os dados destinados a operações em estruturas de dados que só podem ser executados na ME. Estes dados, ao chegarem na MI são empacotados com a instrução respectiva e enviados à UF. A UF por sua vez identifica que a instrução é uma instrução de manipulação de estrutura de dados e envia então o pacote para a CC. Só então na CC que este pacote é enviado para a ME onde é tratado. O mesmo ocorre para dados destinados às instruções tratadas pelo THR.

5.5.2 O *Interleaving* da Memória de Dados

Como mostra a Figura 5.5, existem três unidades que enviam fichas à Memória de Dados (MD) a saber: a Chave de Coleta (CC), a Memória de Estruturas (ME) e a Unidade de Controle de Paralelismo (THR). Portanto, faz-se necessária a implementação de uma chave na entrada da MD que trate as fichas vindas destes canais. A Figura 5.6 mostra a nova arquitetura da Memória de Dados.

A MD é uma “memória” e no momento com a tecnologia atual, não é possível implementar memórias que supram em termos de desempenho, a necessidade do conjunto de processadores. Alguns mecanismos foram propostos em várias arquiteturas comerciais para contornar este problema tais como: *caches*, *prefetch* de instruções e heurísticas para execução de *branches*.

O mecanismo proposto para a solução do problema na arquitetura Wolf II é o *interleaving* da MD. Este mecanismo consiste em segmentar a MD em vários bancos onde

Figura 5.6: A Arquitetura da Memória de Dados.

cada banco é responsável por um intervalo de endereços válidos. Por exemplo, se o maior endereço possível for 300 e a MD for dividida em 10 bancos idênticos, cada banco será responsável por 30 posições válidas.

Considerando que o acesso a cada banco poderá ser feito de forma assíncrona, pode-se garantir que só haverá bloqueio quando duas fichas tentarem ter acesso ao mesmo banco simultaneamente (o conceito de simultaneidade aqui é definido como sendo qualquer tempo menor que o ciclo mínimo de memória, ou seja o tempo necessário para uma operação completa de memória). Para minimizar o bloqueio pode-se implementar heurísticas no mecanismo da FF de forma a garantir que fichas destinadas a um mesmo banco sejam enviadas ao anel com um intervalo mínimo de ciclos. Com este mecanismo a MD poderá ser capaz de suprir a demanda de fichas nos anéis MI-UF.

5.5.3 Eliminação da Memória de Instruções Externa

Antes de propor uma solução para o problema da MI0, é interesse rever a funcionali-

dade desta unidade, isto é o por que é necessário ter uma memória de instruções fora do anel na arquitetura Wolf.

A MI0 existe para que a arquitetura Wolf seja capaz de tratar vetores e matrizes. Quando um dado chega até a MI0, esta detecta se o dado se destina a uma instrução de manipulação de vetor. Se o dado se destinar a uma instrução deste tipo, a MI0 cria uma ficha executável que consiste do dado, da instrução e de campos de controle e envia esta ficha à CD que, identificando o tipo da ficha, a envia até à Memória de Estruturas (ME). Este mecanismo evita que as fichas destinadas à ME recirculem pelo *pipeline* para obterem a instrução correspondente em alguma MI.

Com o objetivo de diminuir a quantidade de estágios do *pipeline* e conseqüentemente diminuir a latência das fichas produzidas, a arquitetura Wolf II não possui a MI0. Nesta arquitetura, as fichas destinadas à instruções de manipulação de vetores não precisam recircular pelo *pipeline* para chegarem à ME, já que esta está reposicionada. O Wolf II utiliza as MI's internas para construção de fichas destinadas à ME. A seguir será apresentada uma proposta de arquitetura que supra as necessidades de processamento vetorial da arquitetura Wolf II.

5.5.3.1 Proposta de uma Arquitetura Vetorial para o WOLF II

O objetivo desta seção é propor um processador vetorial agrupado à uma memória vetorial que sejam capazes de executar instruções vetoriais independentemente dos anéis MI-UF. A arquitetura Wolf usa o mesmo mecanismo usado pela Máquina Fluxo de Dados de Manchester (MFDm) para tratar estruturas de dados como vetores e matrizes. Este mecanismo não é ideal para este tipo de tratamento, já que induz à geração de *overhead* [26], visto que as operações sobre vetores e matrizes são geradas pela ME mas

executadas pelas UF's.

Com a implementação desta arquitetura vetorial, seria impressindível também alterar a linguagem usada pela arquitetura Wolf II. Apesar da semelhança com a arquitetura Wolf que usa como linguagem o mesmo *assembler* usado pela MFDm, a arquitetura Wolf II deverá utilizar uma linguagem que possibilite operações com matrizes e vetores usando as características disponíveis em um processador vetorial. Também não é objetivo deste trabalho desenvolver uma linguagem para estes propósitos ficando esta tarefa para algum trabalho futuro.

O mecanismo a ser implementado para garantir a correta execução de um trecho de código vetorial seria o seguinte: um *thread* vetorial seria inicialmente disparado pela execução de um grafo fluxo de dados usual, onde o resultado da execução de uma determinada instrução dispararia tal *thread*. Após o disparo, cada instrução vetorial seria executada internamente ao conjunto processador-memória vetorial sendo os resultados destinados a instruções não vetoriais, liberados de volta ao anel principal. A Figura 5.7 mostra a arquitetura proposta para o processador e memória vetoriais.

Figura 5.7: Arquitetura para o Processador e Memória vetoriais.

O dado que dispara o *thread* vetorial entra na sub-unidade vetorial pela entrada CC em C3. C3 é uma chave que recebe fichas vindas do anel principal e também dos processadores vetoriais e envia estas fichas para a ME. Na ME estão armazenados toda a parte vetorial dos dados (vetores e matrizes) referentes ao programa sendo executado. A ME após identificar a instrução para a qual o dado de entrada se destina, cria uma ficha executável e envia esta ficha para um dos processadores livres. Após a operação vetorial ser executada, os resultados são enviados novamente para C3.

5.6 Conclusões

Este capítulo apresentou o problema de balanceamento de carga nos anéis MI-UF da arquitetura Wolf. Alguns experimentos foram apresentados no intuito de identificar os motivos deste comportamento. O principal ponto identificado como responsável pelo comportamento da distribuição de carga foi a existência de unidades cujo desempenho deixa muito a desejar. As duas principais unidades identificadas como sendo as mais problemáticas da arquitetura foram a Memória de Dados e a Memória de Instruções Externa.

Para resolver o problema foi proposta uma nova arquitetura denominada Wolf II. Esta arquitetura possui várias unidades que fazem o papel da Memória de Dados, permitindo que várias fichas sejam tratadas simultaneamente. Outra característica da arquitetura Wolf II foi a eliminação da Memória de Instruções Externa. Esta última unidade tornou-se desnecessária devido ao reposicionamento da Memória de Estruturas e da Unidade de Controle de Paralelismo.

Com estas modificações espera-se que a nova arquitetura supra as necessidades dos

anéis MI-UF, tornando a utilização destes anéis a mais homogênea possível. Com todas estas novas características implementadas, esperamos que a arquitetura Wolf II possua um desempenho melhor que o Wolf, tanto na utilização de memória quanto no tempo de processamento.

Conclusões e Trabalhos Futuros

A arquitetura a fluxo de dados Wolf foi bem detalhada neste trabalho. Este detalhamento permitiu a implementação de um simulador para o Wolf. A simulação não é sistêmica, isto é cada unidade opera ignorante do funcionamento das demais. O conjunto se comporta como um multicomputador, fornecendo maior confiança no simulador como um todo.

A implementação do mecanismo de detecção de código seqüencial no simulador mostrou que existe uma melhora no tempo de execução de um programa quando o mecanismo é utilizado. Este aumento no desempenho está diretamente relacionado com o melhor aproveitamento dos recursos disponíveis na arquitetura.

O experimento de balanceamento das unidades mostrou que pode-se eliminar unidades sobrecarregadas e que o tempo (atraso) de cada unidade é proporcional aos valores encontrados em processadores e memórias comerciais. Diferentemente das arquiteturas comerciais, arquiteturas a fluxo de dados possuem características que facilitam a implementação de multiprogramação.

Os resultados obtidos com o experimento de multiprogramação, sugerem que o uso de mecanismos para controle de ativação de processos pode melhorar o desempenho da arquitetura. Alguns problemas foram identificados na arquitetura Wolf. Tomando-se como base estes problemas, foram propostas algumas modificações na arquitetura. A nova arquitetura, denominada Wolf II, propõe-se a solucionar tais problemas.

Pode-se afirmar que arquiteturas a fluxo de dados estão caminhando para a mesma direção que as arquiteturas von Neumann caminham. Hoje em dia, já há processadores híbridos que possuem características de um processador paralelo, mas que ainda usa mecanismos convencionais, refletindo a convergência de arquiteturas von Neumann e arquiteturas paralelas em direção a estas arquiteturas híbridas.

Apesar desta convergência, existe um problema sério que ainda está longe de ser resolvido: a programação destes processadores híbridos. Cada estilo de linguagem (Funcional, Orientada a Objetos, etc) usada para programar estas máquinas, oferece um conjunto de *facilities* que pode ser explorado tanto pelo programador quanto pelo compilador que gerará código objeto para o processador. O ideal seria se o programador ficasse livre para escrever seu código sem se preocupar em paralelizá-lo, deixando a cargo do compilador esta tarefa. Infelizmente, as linguagens mais populares para programação (Fortran, C, C++, etc) não permitem que seja extraído paralelismo facilmente de um código escrito nestas linguagens.

A falta de uma linguagem que seja largamente utilizada como C ou Fortran e que ao mesmo tempo deixe o programador isento da preocupação com o paralelismo, piora a situação pois provoca uma heterogenização entre os grupos de pesquisa quanto à linguagem a ser utilizada. Neste caso, observa-se que existe praticamente uma linguagem adotada para cada grupo de pesquisa (HPF, DPCE, etc) e que todas elas ainda deixam a desejar, dependendo da aplicação. Até o momento, não há solução definitiva para o problema de programação de máquinas híbridas.

5.7 Trabalhos Futuros

Esta seção apresenta os trabalhos principais originados do estudo da arquitetura Wolf.

1. **Arquitetura Vetorial:** Este trabalho consistiria em implementar a arquitetura vetorial proposta na seção 5.5.3.1 para a arquitetura Wolf II. Com esta implementação, poder-se-ia executar programas que usam estruturas de dados (vetores e matrizes) localmente na arquitetura. Esta localidade evitaria o *overhead* causado por operações com este tipo de dado (lembramos que no Wolf, todas as operações são executadas pelas unidades funcionais (UF)). O problema que surge é a falta de um compilador que implemente esta característica.
2. **Linguagem Vetorial:** Este trabalho consistiria em implementar uma linguagem com características vetoriais para a arquitetura Wolf II. Como ponto de partida, sugere-se o estudo das construções em SISAL. Esta linguagem de alto nível poderia ser adaptada para tratar operações vetoriais sem muitas dificuldades devido às suas características de linguagem paralela. Após a implementação da linguagem, seria necessária a construção de um compilador capaz de gerar código para o Wolf II.
3. **Compilador para o Wolf:** Este trabalho consistiria em implementar um compilador capaz de gerar código para o Wolf II. O compilador usado para o Wolf é o mesmo criado para a MFDM, não sendo adequado para explorar as características da arquitetura Wolf. O trabalho de construir um compilador para o Wolf II deveria ser feito tomando-se como base todas as características da arquitetura. Não seria aceitável fazer pequenas alterações no compilador atual para adaptá-lo à geração de código vetorial apenas.

4. **Memória de Dados:** Este trabalho consistiria em implementar as modificações (*interleaving*) na Memória de Dados propostas na seção 5.5.2. Com estas alterações a MD poderia tratar paralelamente várias fichas diminuindo a latência do *pipeline*. O problema de saber para qual módulo será enviada uma ficha pode ser resolvido usando-se o próprio endereço de destino do dado. Pode ser também implementado um mecanismo na Fila de Fichas para que esta não mande duas fichas consecutivas para um mesmo banco da MD, evitando assim que haja muitas condições de espera dentro da unidade. Estas esperas podem degradar o paralelismo. Um possível problema que aparecerá e que deverá ser tratado a contento, é o desbalanceamento na utilização dos bancos da MD. Isto se deve ao fato de existirem regiões do grafo que são executadas mais vezes que outras.
5. **Calibragem:** Este trabalho compreenderia a calibragem do simulador da arquitetura Wolf II da mesma maneira que foi descrita para o Saw (Capítulo 3). Uma versão do simulador para o Wolf II já se encontra operacional com excessão de alguns ajustes no mecanismo de temporização. A calibragem faz-se necessária para que não existam unidades sub-utilizadas e unidades super-utilizadas na arquitetura.
6. **Unidade de Controle de Paralelismo:** Neste trabalho seria feito um estudo para averiguar a necessidade da Unidade de Controle de Paralelismo na arquitetura Wolf II. Constatada a sua necessidade, esta seria então implementada. Esta unidade seria parecida com a implementada para a MFDM, já que existe a necessidade de reaproveitamento de “nomes de ativação” e detecção de terminação de processos para a execução de programas realísticos.
7. **Multiprogramação:** Averiguar porque o experimento de multiprogramação apre-

sentado no Capítulo 3 não apresenta bons resultados. Baseando-se em resultados da averiguação feita, implementar modificações na arquitetura Wolf para que esta seja eficiente no tratamento de multiprogramação. Com esta característica implementada, poderia-se utilizar o simulador para executar tarefas diversas com o compartilhamento dos recursos disponíveis.

Bibliografia

- [1] Garcia Neto, A.; Cavenaghi, M.A.; *Primeiros Resultados da Proto-Arquitetura a Fluxo de Dados Wolf*; IV Simpósio Brasileiro de Arquiteturas de Computadores e Processamento de Alto Desempenho; São Paulo; 1992
- [2] Cavenaghi, M.A.; Garcia Neto, A.; Ruggiero, C.A.; *Arquiteturas a Fluxo de Dados e Processamento de Sinais*; I Workshop Sobre Computação de Alto Desempenho para Processamento de Sinais; São Carlos; 1993
- [3] Cavenaghi, M.A.; *Simulação e Estudo da Proto-Arquitetura Wolf*; I Simpósio Interno da Pós-Graduação do DFCM; São Carlos; 1994
- [4] Cavenaghi, M.A.; *Estudo do Comportamento do Wolf em uma Aplicação de Dinâmica Molecular*; II Simpósio Interno da Pós-Graduação do DFCM; São Carlos; 1995
- [5] Cavenaghi, M.A.; Garcia Neto, A.; *Wolf - Uma Arquitetura para Processamento de Alto Desempenho*; XVIII Encontro Nacional de Física da Matéria Condensada; Caxambu; 1995
- [6] Cavenaghi, M.A.; Garcia Neto, A.; *Desempenho do Simulador Saw em Problemas de Métodos Numéricos*; XVI Congresso Ibero Latino Americano sobre Métodos Computacionais para Engenharia; Curitiba; 1995
- [7] Cavenaghi, M.A.; et al; *Using PVM in the Simulation of a Hybrid Dataflow Architecture*; Third Euro-PVM Users' Group Meeting; Muenchen; 1996
- [8] Cavenaghi, M.A.; *Implementação de um Simulador para a Arquitetura Fluxo de Dados WOLF*; Dissertação de Mestrado; USP-IFQSC-DFCM; São Carlos-SP; 1992
- [9] Gurd, J.R.; Kirkham, C.C.; Watson, I.; *The Manchester Prototype Dataflow Computer*; Communication of ACM 28(1):34-52; 1985

- [10] Arvind; Nikhil, R.S.; *Executing a Program on the MIT Tagged-Token Dataflow Architecture*; IEEE Trans. on Computers; 39(3): 300-318; 1990
- [11] Nikhil, R.S.; *Id (version 88.0) Reference Manual*; Technical Report CSG Memo 284; LCS-MIT; 1988
- [12] Arvind; Thomas, R.E.; *I-Structures: An Efficient Data Structure for Functional Languages*; Laboratory of Computer Science - MIT; 1981
- [13] Papadopoulos, G.M.; *Implementation of a General Purpose Dataflow Multiprocessor*; PhD Thesis; Technical Report TR 432; LCS-MIT; 1988
- [14] Hicks, J. et al; *Performance Studies of Id on the Monsoon Dataflow System*; Journal of Parallel and Distributed Computing (18):273-300; 1993
- [15] Culler, D.E.; Papadopoulos, G.M.; *The Explicit Token Store*; Journal of Parallel and Distributed Computing (10); 1990
- [16] Nikhil, R.S.; Papadopoulos, G.M.; Arvind; **T: A Multithreaded Massively Parallel Architecture*; ACM SigArch 20(2): 156-167; 1992
- [17] Ang, B.S.; Arvind; Chio, D.; *StarT the Next Generation: Integrating Global Caches and Dataflow Architectures*; Proceedings; ISCA 1992 Dataflow Workshop; 1992
- [18] Chio, D. et al; *StarT-NG: Delivering Seamless Parallel Computing*; Proceedings; EURO-PAR'95; 1995
- [19] Papadopoulos, G.M.; Traub, K.R.; *Multithreading: A Revisionist view of Dataflow Architectures*; Proc. ISCA91:342-351; 1991
- [20] Iannucci, R.A.; *Towards a Dataflow/von Neumann Hybrid Architecture*; Proc. 15th Ann. Int'l Symp. Computer Architecture; IEEE CS Press, Los Alamitos, CA, 1988
- [21] Sakai, S. et al; *Reduced Interprocessor-Communication Architecture and its Implementation on EM-4*; Parallel Computing (21):753-769; Elsevier Science B.V.; 1995
- [22] Sekiguchi, S.; Shimada, T.; Hiraki, K.; *A Design of Practical Dataflow Language DFCII and its Data Structures*; ETL Technical Report; ETL-TR-90-16; 1990
- [23] Sakai, S.; Kodama, Y.; Yamaguchi, Y.; *Architectural Design of a Parallel Supercomputer EM-5*; Proc. JSPP91:149-156; 1991

- [24] Szwarcfiter, Jayme L.; *Grafos e Algoritmos Computacionais*; Ed. Campus; 2a. edição
- [25] Schauser, K.E.; *Compiling Dataflow into Threads*; MSc Thesis; Computer Science Division; University of California; 1991
- [26] Sargeant, J.; *Efficient Stored Data Structures for Dataflow Computing*; PhD Thesis; Department of Computer Science; University of Manchester; 1985
- [27] Ruggiero, C.A.; *Throttle Mechanisms for the Manchester Dataflow Machine*; PhD Thesis; Department of Computer Science; University of Manchester; 1987
- [28] Dongen, V.; Bonello, C.; Gao, G.; *Data Parallelism with High Performance C*; CASCON'94: Integrated Solutions; 1994
- [29] Koebel, C. et al; *The High Performance FORTRAN Handbook*; Scientific and Engineering Computation Series; MIT Press 1994
- [30] McGraw, J.R.; *SISAL: Streams and Iterations in a Single Assignment Language*; Language Reference Manual; LLNL; Tech. Report; M-146; 1985
- [31] Vetterling, W.T. et al; *Numerical Recipes: Example Book (FORTRAN)*; Cambridge University Press; 1992
- [32] Press, W.H. et al; *Numerical Recipes in C: The Art of Scientific Computing*; 2nd Edition; Cambridge University Press; 1993
- [33] Skedzielewski, S.K.; Glauert, J.R.W.; *IF1, an Intermediate Form for Aplicative Language*; Reerence Manual M-170; LLNL; jul. 1985
- [34] Teo, Y.M.; *Concurrency control in the Multi-Ring Manchester Dataflow Machine*; Technical Report Series UMCS-87; DCS; University of Manchester; 1989
- [35] Travieso, G.; *Proposta e Implementação de um Sistema Paralelo para Dinâmica Molecular*; Tese de doutorado; DFCM-IFQSC-USP 1993
- [36] Brigham, E.O.; *The Fast Fourier Transform*; Prentice Hall; 1974
- [37] Ruggiero, M.A.G.; Lopes, V.L.R.; *Cálculo Numérico: Aspectos Teóricos e Computacionais*; Makron Books; 1988
- [38] Garcia Neto, A.; *Distributed Simulation Using "Relaxed Timing"*; PhD Thesis; DCS; University of Manchester; 1990

- [39] Wirth, N.; *Algoritmos e Estruturas de Dados*; Prentice Hall do Brasil; 1989
- [40] Sargeant, J.; *Simulator Users Guide*; DCS; Manchester; 1985
- [41] Böhn, A.P.W.; Teo, Y.M.; *The Heterogeneous Multi-Ring Dataflow Machine Simulator*; Technical Report Series; Manchester; 1988
- [42] Gaudiot, J.L.; Bie, L.; *Advanced Topics in Dataflow Computing*; Prentice Hall; Englewood Cliffs; 1991
- [43] Foley, J.F.; *Manchester Dataflow Machine: Benchmark Test Evaluation Report*; Internal Report UMCS-89-11-1; DCS; University of Manchester; 1989
- [44] Iannucci, R.A.; *A Dataflow/von Neumann Hybrid Architecture*; Technical Report MIT-LCS-TR-418; 1988
- [45] Arvind; Culler, D.E.; Ekanadham, E.; *The Price of Fine-Grained Asynchronous Parallelism: An Analysis of Dataflow Methods*; Proc. CONPAR 88; 1988
- [46] Castro, L.S.V.; *Pontos de Estatística*; Ed. Científica; RJ; 1967

Apêndice A - Código dos *Benchmarks* em *SISAL*

Ackermann

```
export main

function ackerm(m,n:integer returns integer)
  if m=0 then
    n+1
  elseif n=0 then
    ackerm(m-1,1)
  else
    ackerm(m-1,ackerm(m,n-1))
  end if
end function

function main(a,b:integer returns integer)
  ackerm(a,b)
end function
```

Dinâmica Molecular

```

DEFINE Main

TYPE Double      = DOUBLE_REAL;
TYPE Cartesian   = ARRAY [ Double ];
TYPE Energy      = Double;
TYPE ParticleNumber = INTEGER;
TYPE Pair        = ARRAY [ ParticleNumber ];
TYPE Cell        = ARRAY [ ParticleNumber ];
TYPE CellNumber  = INTEGER;
TYPE CellCoordinates = ARRAY [ INTEGER ];
TYPE Position    = Cartesian;
TYPE Velocity    = Cartesian;
TYPE Acceleration = Cartesian;
type ParticlesData = array [ array [ cartesian ] ];

type Four_Plex = array[ integer ];      % The four 12-bit integers that when
                                         % used together simulate a 48-bit
                                         % integer or the mantissa of a 64-bit
                                         % floating point number.

type Seed_Array = array[ Four_Plex ];   % One seed per process.

type Bit_Array = array[ integer ];      % The binary expansion of a Four_Plex.

global sqrt( X : double returns double )
global sqrt ( x : double returns double )
global cos ( x : double returns double )
global log ( x : double returns double )
%global rans ( N, Seed : integer returns array [Four_Plex] )
%global ranf ( Seed : Four_Plex returns double, Four_Plex )

%define rans, ranf
%%$entry=rans, ranf

% This package is a test module for the SISAL version of the RANF random
% number generator, as implemented by J. Feo and T. DeBoni. It accepts a
% single number, divides the RANF sequence into at least that number of
% subsequences, and produces a new random number and seed for each call.
% The user is expected to keep the subsequences separate by keeping the
% seeds for the subsequences separated. This allows the user to utilize
% as much parallelism as desired.

%=====
% FUNCTION DEFINITIONS
%=====

function ranf_mod_mult( a, b: Four_Plex returns Four_Plex )

% Returns a * b where both a and b are 4 12-bit integers and the
% multiplication is done piecemeal.

let
  j0 := a[0] * b[0];
  j1 := a[0] * b[1] + a[1] * b[0];
  j2 := a[0] * b[2] + a[1] * b[1] + a[2] * b[0];
  j3 := a[0] * b[3] + a[1] * b[2] + a[2] * b[1] + a[3] * b[0];
  k0 := j0;
  k1 := j1 + k0 / 4096;
  k2 := j2 + k1 / 4096;
  k3 := j3 + k2 / 4096;
in
  array [0: mod(k0, 4096), mod(k1, 4096), mod(k2, 4096), mod(k3, 4096)]
end let

end function % ranf_mod_mult

%=====

function ranf_k( n: integer returns Four_Plex )

% Returns the floor of (2**46) / nn, where 2**46 is represented by
% 4 12-bit integers, in base ten [1024, 0, 0, 0], and the division
% is done piecemeal. For statistical reasons the divisor is made odd.
%
let
  nn := if mod(n, 2) = 0 then n + 1 else n end if;

  q3 := 1024 / nn;
  r3 := 1024 - (nn * q3);
  q2 := (r3 * 4096) / nn;
  r2 := (r3 * 4096) - (nn * q2);
  q1 := (r2 * 4096) / nn;
  r1 := (r2 * 4096) - (nn * q1);
  q0 := (r1 * 4096) / nn;
in
  array [0: q0, q1, q2, q3]
end let

end function % ranf_k

%=====

function ranf_k_binary( k: Four_Plex returns Bit_Array )

% Returns the binary representation of k, where k is represented
% by 4 12-bit integers. The bits are stored in the returns array
% from least significant to most significant.

for i in 0, 3
  returns value of catenate

  for initial
    j := 1;
    x := k[i] / 2;
    bit := mod(k[i], 2);

```



```

        while j < 12 repeat
            j := old j + 1;
            x := old x / 2;
            bit := mod(old x, 2)
            returns array of bit
        end for

    end for

end function % ranf_k_binary

%=====
function ranf_a_to_k( a: Four_Plex; k: Bit_Array returns Four_Plex )

% Returns a to the k mod 2**48. k is a binary number, so it is
% equivalent to the sum of k[i] * 2**i. a to the k mod 2**48 is
% thus the product of a ** (k[i] * 2**i) mod 2**48. We calculate
% iteratively a ** (2**i) mod 2**48, and form the result by mul-
% tipling together those terms (mod 2**48) for which k[i] is 1.

    for initial
        i := 0;
        a_i := a;
        a_k := array [0: 1, 0, 0, 0]
        while i < 46 repeat
            i := old i + 1;
            a_k := if k[i] = 0 then
                old a_k
            else
                ranf_mod_mult( old a_k, old a_i )
            end if;
            a_i := ranf_mod_mult( old a_i, old a_i )
        returns value of a_k
    end for

end function % ranf_a_to_k

%=====
function rans( N_in, Seed1: integer returns array[ Four_Plex ] )

% Returns n seeds by dividing the wheel into n segments and
% returning the start seed of each segment. If seed1 is 0, the
% routine will return all zeros; in such cases, seed1 is set to
% [3281, 4041, 595, 2376] which is statistically the best start-
% ing seed. The wheel is divided into n pieces by dividing its
% period (2**46) by n (= k). Then
%
% seed(i) = seed(i - 1) * a ** k (mod 2**48)
%
% where a is the multiplier of the random number generator.

    for initial
        N := if mod(N_in, 2) = 1 then N_in else N_in + 1 end if;
        i := 1;
        seed := if Seed1 = 0 then
            array [0: 3281, 4041, 595, 2376]
        else
            array [0: abs(Seed1), 0, 0, 0]
        end if;
        a := array [0: 373, 3707, 1442, 647];
        a_k := if N > 1 then
            ranf_a_to_k( a, ranf_k_binary( ranf_k( N ) ) )
        else % Don't bother to raise a to the first power.
            a
        end if

        while i < N repeat
            i := old i + 1;
            seed := ranf_mod_mult(old seed, a_k)

        returns array of seed
    end for

end function % rans

%=====
function ranf( Seed: Four_Plex
    returns double,           % Random Number
    Four_Plex)                % New Seed

% Returns the next random number and seed in the series. The seed's
% bits shifted to the right 48 places are the bits of the random
% number. The next seed is the product of the current seed and the
% series' multiplier.

    double_real(Seed[3]) / 4096.0d0 +
    double_real(Seed[2]) / 16777216.0d0 +
    double_real(Seed[1]) / 68719476736.0d0 +
    double_real(Seed[0]) / 281474976710656.0d0,

    ranf_mod_mult( array [0: 373, 3707, 1442, 647], Seed )

end function % ranf

%
% Extracts the r-th root of x
%
function root( r, x: double returns double)
    exp( x, 1.0d0/r )
end function % root

%
% Returns number of particles, given number of dimensions (D) and

```

```

% number of lattice unities (M).
%
function NumberOfParticles (D, M : integer returns integer)
    M * M * (if D=2 then 2 else 4*M end if)
end function % NumberOfParticles

%
% Generate particle positions on a face centered cubic distribution
% D is the number of dimensions (2 or 3)
% M is the number of lattice elements to generate in each dimension
% LatticeSeparation is the separation between two adjacent lattice elements
%
function FCClattice ( D, M : integer; LatticeSeparation: double
returns array [ Cartesian ] )

    function Combine2to2 (N: integer returns array [ array [integer] ])
        for i in 1, N
            pairs := for j in i+1, N
                pair := array [i: i, j]
                returns array of pair
            end for
            returns value of catenate pairs
        end for
    end function % Combine2to2

    function Cube (D, Repetitions : integer; Offset, Delta : Cartesian
returns array [ Cartesian ])
        if (D = 1) then
            for i in 1, Repetitions
                Element := array [1: double_real(i-1) * Delta[1] + Offset[1] ]
                returns array of Element
            end for
        else
            let
                ThisOffset := Offset[array_size(Offset)];
                ThisDelta := Delta[array_size(Delta)];
                BaseElement := Cube (D-1, Repetitions,
                    array_remh (Offset), array_remh (Delta));
            in
                for i in 1, Repetitions
                    ThisCoordinate := double_real(i-1) * ThisDelta + ThisOffset;
                    OneRepetition :=
                        for Element in BaseElement
                            NewElement := array_addh (Element, ThisCoordinate)
                            returns array of NewElement
                        end for
                    returns value of catenate OneRepetition
                end for
            end let
        end if
    end function % Cube

    let
        Pairs := Combine2to2(D);
        Offsets := array_add1 (
            for p in Pairs
                NewBase := for initial
                    Base := array_fill (1, D, 0.0d0);
                    i := 1
                    while (i<=2) repeat
                        Base := old Base [ p[old i] : LatticeSeparation/2.0d0 ];
                        i := old i + 1;
                    returns value of Base
                    end for
                    returns array of NewBase
                end for , array_fill (1, D, 0.0d0) );
        Delta := array_fill (1, D, LatticeSeparation);
    in
        for Offset in Offsets
            ACube := Cube (D, M, Offset, Delta)
            returns value of catenate ACube
        end for
    end let
end function % FCClattice

%
% Generate velocities with a Maxwell-Boltzmann distribution of temperature T
% M is the number of particles, and D the number of dimensions
%
function MaxwellDistribution (D, M : integer; T : double
returns array [ Cartesian ] )
    let
        Randoms := rans ( D * M, 0);
        pi := 3.141592653898d0;
        V0 := sqrt (T);
    in
        for particle in 1, M
            vel := for coord in 1, D
                v := let
                    X1, seed1 := ranf ( Randoms[ 2*particle + coord - 2 ] );
                    X2, seed2 := ranf (seed1);
                in
                    cos ( 2.0d0 * pi * X1 ) * sqrt ( -2.0d0 * log (X2) ) * V0
                end let
                returns array of v
            end for
            returns array of vel
        end for
    end let
end function % MaxwellDistribution

%
% Generates initial positions, velocities and accelerations for a
% fcc lattice, for given dimensions and temperature
% NOTE: accelerations are initialized in zero.
%
% D -> number of dimensions
% M -> dimension of lattice

```

```

%
function GenPos ( D, M: integer; rho, T: double
returns Double, ParticlesData )

let
  N := NumberOfParticles (D, M);
  Vol := double_real(N) / rho;
  BoxSize := root ( double_real(D), Vol);
  LatticeSeparation := BoxSize / double_real(M);
  r0_box := FCClattice (D, M, LatticeSeparation);
  r := FOR i IN 1, N
    rh_h := FOR k IN 1, D
      rk := r0_box[i][k] - BoxSize/2.0d0
      RETURNS ARRAY OF rk
    END FOR
    RETURNS ARRAY OF rh_h
  END FOR;
  v := MaxwellDistribution (D, M, T);
  a := array_fill (1, N, array_fill (1, D, 0.0d0));
in
  BoxSize, ARRAY ParticlesData [ 1: r, v, a ]
end let
end function % main

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      Adds two cartesian dimension by dimension
%
FUNCTION AddCartesian ( x, y : Cartesian RETURNS Cartesian )

  FOR k IN 1, ARRAY_SIZE ( x )
    RETURNS ARRAY OF x[k] + y[k]
  END FOR

END FUNCTION % AddCartesian

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      Subtracts cartesian y from cartesian x, dimension by dimension
%
FUNCTION SubtractCartesian ( x, y : Cartesian RETURNS Cartesian )

  FOR k IN 1, ARRAY_SIZE ( x )
    RETURNS ARRAY OF x[k] - y[k]
  END FOR

END FUNCTION % SubtractCartesian

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      Adds up all components of array x
%
FUNCTION AddComponents ( x : ARRAY [ Acceleration ]; D : INTEGER
returns Acceleration )

  IF ARRAY_SIZE ( x ) /= 0
  THEN
    FOR k IN 1, D
      sk := FOR v IN x
        RETURNS VALUE OF SUM v[k]
      END FOR
      RETURNS ARRAY OF sk
    END FOR
  ELSE
    ARRAY_FILL ( 1, D, 0.0d0 )
  END IF

END FUNCTION % AddComponents

FUNCTION InBox ( rr, BoxSize : Double RETURNS Double )
  rr - DOUBLE_REAL ( INTEGER ( rr/BoxSize ) ) * BoxSize
END FUNCTION % InBox

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%      First pass of move ( Velocity-Verlet Algorithm )
%
%      Generates new position and a half step in velocity:
%
%      v := v + 0.5 a delta t
%      r := r + v delta t + 0.5 a delta t
%
FUNCTION FirstPass ( r : ARRAY [ Position ];
  v : ARRAY [ Velocity ];
  a : ARRAY [ Acceleration ];
  BoxSize, Delta : Double
returns ARRAY [ Position ], ARRAY [ Velocity ] )

LET
  N := ARRAY_SIZE (r);
  D := ARRAY_SIZE (r[1]);
  HalfDelta := Delta/2.0d0;
IN
  FOR i IN 1, N
    ri, vi := FOR j IN 1, D
      vj := v[i][j] + HalfDelta * a[i][j];
      rj := InBox ( r[i][j] + Delta * vj, BoxSize );
      RETURNS ARRAY OF rj
    END FOR
    RETURNS ARRAY OF vi
  END FOR

```

```

RETURNS ARRAY OF ri
ARRAY OF vi

END FOR

END LET

END FUNCTION % FirstPass

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Second pass of move ( Velocity-Verlet Algorithm )
%
% Terminates update of velocity:
%
%  $v := v + 0.5 a \Delta t$ 
%
FUNCTION SecondPass ( v : ARRAY [ Velocity ];
                    a : ARRAY [ Acceleration ];
                    Delta : Double
                    RETURNS ARRAY [ Velocity ] )

LET

N := ARRAY_SIZE ( v );
D := ARRAY_SIZE ( v[1] );
HalfDelta := Delta/2.0d0;

IN

FOR i IN 1, N
  FOR j IN 1, D
    vj := v[i][j] + HalfDelta * a[i][j]
  RETURNS ARRAY OF vj
  END FOR
  RETURNS ARRAY OF vi
END FOR

END LET

END FUNCTION % SecondPass

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Given a cell number, generates cell coordinates on cell grid.
%
% Correspondence:
%
% +-----+ +-----+
% | 3,0 | 3,1 | 3,2 | 3,3 | | 13 | 14 | 15 | 16 |
% +-----+ +-----+
% | 2,0 | 2,1 | 2,2 | 2,3 | | 9 | 10 | 11 | 12 |
% +-----+ +-----+
% | 1,0 | 1,1 | 1,2 | 1,3 | | 5 | 6 | 7 | 8 |
% +-----+ +-----+
% | 0,0 | 0,1 | 0,2 | 0,3 | | 1 | 2 | 3 | 4 |
% +-----+ +-----+
%
FUNCTION ToCellCoordinates ( c : CellNumber; Ncellsk, D : INTEGER
                          RETURNS CellCoordinates )
FOR INITIAL
  i := 1;
  rest := (c-1) / Ncellsk;
  ck := MOD ( c-1, Ncellsk );
  WHILE i < D REPEAT
    rest := old rest / Ncellsk;
    ck := MOD ( old rest, Ncellsk );
    i := old i + 1;
  RETURNS ARRAY OF ck
END FOR
END FUNCTION % ToCellCoordinates

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Given cell coordinates on grid, find the cell number.
% See "Correspondence" above.
%
FUNCTION ToCellNumber ( coords : CellCoordinates; Ncellsk : INTEGER
                      RETURNS CellNumber )
FOR INITIAL
  i := ARRAY_LIMH ( coords );
  c := 0;
  WHILE i >= ARRAY_LIML ( coords ) REPEAT
    c := old c + Ncellsk + coords[old i];
    i := old i - 1;
  RETURNS VALUE OF c+1
END FOR
END FUNCTION % ToCellNumber

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Generate neighbouring cells (of all cells) to be considered on the
% calculation of the Verlet table.
%
FUNCTION GenerateNeighbours ( Ncellsk, D : INTEGER
                             RETURNS ARRAY [ ARRAY [ CellNumber ] ] )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Find all neighbours of a given cell
%
FUNCTION AllNeighbours ( central : CellCoordinates; Ncellsk : INTEGER
                       RETURNS ARRAY [ CellCoordinates ] )

LET

```

```

First := central [ ARRAY_LIML ( central ) ];
Rest := ARRAY_REWL ( central );
Previous := MOD ( First-1+Ncellsk, Ncellsk );
Same := First;
Next := MOD ( First+1, Ncellsk );

IN

IF ARRAY_SIZE ( central ) = 1
THEN
  ARRAY [ 1: ARRAY [ 1: Previous ], ARRAY [ 1: Same ], ARRAY [ 1: Next ] ]
ELSE
  LET
    Base := AllNeighbours ( Rest, Ncellsk );
  IN
    FOR b IN Base
      newb := FOR element IN ARRAY [ 1: Previous, Same, Next ]
        neigh := ARRAY_ADDL ( b, element )
        RETURNS ARRAY OF neigh
      END FOR
    RETURNS VALUE OF CATENATE newb
  END FOR
END LET
END IF
END LET

END FUNCTION % AllNeighbours

LET
  Ncells := INTEGER ( EXP ( DOUBLE_REAL ( Ncellsk ), DOUBLE_REAL ( D ) ) );
IN
  FOR cell IN 1, Ncells
    coords := ToCellCoordinates ( cell, Ncellsk, D );
    neighcoords := AllNeighbours ( coords, Ncellsk );
    neighs := FOR ng IN neighcoords
      number := ToCellNumber ( ng, Ncellsk )
      RETURNS ARRAY OF number WHEN number < cell
    END FOR;
  RETURNS ARRAY OF neighs
END FOR
END LET

END FUNCTION % GenerateNeighbours

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Distribute particles in cells.
% That is: for each particle, find the cell that it belongs, and
% assemble list of particles per cell
%
FUNCTION Distribute ( r: ARRAY [ Position ]; CellSize, BoxSize : Double;
  Ncellsk, D : INTEGER
  RETURNS ARRAY [ Cell ] )

LET
  Ncells := INTEGER ( EXP ( DOUBLE_REAL ( Ncellsk ), DOUBLE_REAL ( D ) ) );
  CellOfParticle := FOR p IN r
    cc := FOR k IN 1, D
      ck := FLOOR ((p[k] + BoxSize/2.0d0) / CellSize)
      RETURNS ARRAY OF ck
    END FOR;
    cell := ToCellNumber ( cc, Ncellsk )
    RETURNS ARRAY OF cell
  END FOR;
IN
  FOR cell IN 1, Ncells
    particlesincell := FOR cp IN CellOfParticle AT i
      RETURNS ARRAY OF i WHEN cp = cell
    END FOR
  RETURNS ARRAY OF particlesincell
END FOR

END LET

END FUNCTION % Distribute

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Given cell structure, find the Verlet "Table of neighbours"
% ( pairs of possibly interacting particles )
%
FUNCTION VerletTable ( r : ARRAY [ Position ];
  Cells : ARRAY [ Cell ];
  Neighbours : ARRAY [ ARRAY [ CellNumber ] ];
  Rv, BoxSize : Double
  RETURNS ARRAY [ Pair ] )

FUNCTION Interacts ( ri, rj : Position; Rv2, BoxSize : Double
  RETURNS BOOLEAN )

LET
  r2 := FOR k IN ARRAY_LIML ( ri ), ARRAY_LIMH ( ri )
    dk := InBox ( ri[k]-rj[k], BoxSize )
  RETURNS VALUE OF SUM dk * dk
END FOR;
IN
  r2 < Rv2
END LET

END FUNCTION % Interacts

LET
  Rv2 := Rv * Rv;
IN
  FOR cell IN Cells AT cellnumber
    withcell := FOR part IN cell AT i

```

```

withpart :=
LET
  internals := FOR j IN i+1, ARRAY_LIMH ( cell )
  f := Interacts ( r[part], r[cell[j]], Rv2, BoxSize )
  RETURNS ARRAY OF ARRAY [1: part, cell[j] ] WHEN f
END FOR;
externals := FOR neighbour IN Neighbours[cellnumber]
oneneigh := FOR other IN Cells[neighbour]
  f := Interacts ( r[part], r[other], Rv2, BoxSize )
  RETURNS ARRAY OF ARRAY [1: part, other ] WHEN f
END FOR
  RETURNS VALUE OF CATENATE oneneigh
END FOR

IN
  internals || externals
END LET
RETURNS VALUE OF CATENATE withpart
END FOR
  RETURNS VALUE OF CATENATE withcell
END FOR

END LET

END FUNCTION % VerletTable

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Given coordinates and Verlet table, find accelerations in each particle,
% potential energy, virial and number of interactions.
%
FUNCTION CalculateForces ( r : ARRAY [ Position ];
  Pairs : ARRAY [ Pair ];
  Rc, BoxSize : Double
  RETURNS ARRAY [ Acceleration ], Energy, Double, INTEGER )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Calculate Lennard-Jones interaction between particles with
% positions ri and rj.
%
FUNCTION LennardJones ( ri, rj : Position; Rc2, BoxSize : Double
  RETURNS Acceleration, Energy, Double, INTEGER )

  LET
    D := ARRAY_SIZE ( ri );
    dij, Rij2 := FOR k IN 1, D
    dk := ri[k] - rj[k];
    dk_wrap := InBox ( dk, BoxSize );
    RETURNS ARRAY OF dk_wrap
    VALUE OF SUM dk_wrap*dk_wrap
  END FOR;

  IN
    IF Rij2 < Rc2 THEN

  LET
    sr2 := 1.0d0 / Rij2;
    sr6 := sr2 * sr2 * sr2;
    sr12 := sr6 * sr6;
    tmp := sr12 - sr6;
    Uij := 4.0d0 * tmp;
    Wij := 8.0d0 * ( tmp + sr12 );
    aij := 3.0d0 * Wij * sr2;
    aij := FOR k IN 1, D
    aijk := aij * dij[k]
    RETURNS ARRAY OF aijk
    END FOR

  IN
    aij, Wij, Uij, 1

  END LET

  ELSE

  ARRAY_FILL (1, D, 0.0d0), 0.0d0, 0.0d0, 0

  END IF

  END LET

END FUNCTION % LennardJones

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Sum all accelerations in array a that refers to each particle
% (according to data in array particles)
%
FUNCTION SumUpInteractions ( particles : ARRAY [ ParticleNumber ];
  a : ARRAY [ Acceleration ];
  N, D : INTEGER
  RETURNS ARRAY [ Acceleration ] )

  FOR i IN 1, N

    ai := LET
      a_over_i := FOR p IN particles AT k
      RETURNS ARRAY OF a[k] WHEN p = i
      END FOR;

    IN
      AddComponents ( a_over_i, D )
    END LET

    RETURNS ARRAY OF ai

```

```

END FOR

END FUNCTION % SumUpInteractions

LET

  N := ARRAY_SIZE ( r );
  D := ARRAY_SIZE ( r[1] );
  Rc2 := Rc * Rc;
  centrals, others, a_pairs, W, U, Ncut :=
    FOR pair IN Pairs
      central := pair[1];
      other := pair[2];
      ap, wp, up, np := LennardJones ( r[central], r[other], Rc2, BoxSize )
    RETURNS
      ARRAY OF central WHEN np = 1
    ARRAY OF other WHEN np = 1
    ARRAY OF ap WHEN np = 1
    VALUE OF SUM wp
    VALUE OF SUM up
    VALUE OF SUM np
    END FOR;
  a_centrals := SumUpInteractions ( centrals, a_pairs, N, D );
  a_others := SumUpInteractions ( others, a_pairs, N, D );
  a := FOR i IN 1, N
    ai := SubtractCartesian ( a_centrals[i], a_others[i] )
    RETURNS ARRAY OF ai
  END FOR;

IN

  a, W, U, Ncut

END LET

END FUNCTION % CalculateForces

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Calculates kinetic energy, given the velocities
%
FUNCTION KineticEnergy ( v : ARRAY [ Velocity ]
  RETURNS Energy )

  FOR i IN 1, ARRAY_SIZE ( v )

    K := FOR d IN 1, ARRAY_SIZE ( v[i] )
      RETURNS VALUE OF SUM v[i][d]*v[i][d]
    END FOR

    RETURNS VALUE OF SUM K/2.0d0

  END FOR

END FUNCTION % KineticEnergy

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Main function
%
% Reads particle and simulation data, and performs simulation,
% The algorithm used is a mixed Cells-Verlet Table one:
% A table of neighbours (here an array of possibly interacting pairs)
% is used in the calculation of forces (accelerations). This reduces
% the time for force calculations to O(N).
% This table includes particles at distances until Rv > cutoff radius (Rc)
% and is infrequently updated (controlled by TableRemakeInterval).
% In the updates, to avoid a O(N^2) search in all pairs, a cell
% structure is build.
%
%
FUNCTION Main ( D, N : INTEGER;
  rho, T : Double;
  NumberOfSteps, OutputInterval, TableRemakeInterval : INTEGER;
  Delta, Rc, Rv : Double
  RETURNS Double, Double,
  ARRAY [ ParticlesData ], % Intermediate data
  ARRAY [ Energy ], ARRAY [ Energy ], % Energies
  ARRAY [ Double ], % Pressure
  ARRAY [ INTEGER ] ) % Interacting pairs

  LET
    BoxSize, InitialData := GenPos ( D, N, rho, T );
    N := ARRAY_SIZE ( InitialData[1] ); % Number of particles
    rD := DOUBLE_REAL ( D ); % idem
    Ncellsk := FLOOR ( BoxSize / Rv ); % Number of divisions
    CellSize := BoxSize / DOUBLE_REAL ( Ncellsk );
    Neighbours := GenerateNeighbours ( Ncellsk, D );

  IN

    Rc, EXP ( BoxSize, rD ),

  FOR INITIAL

    step := 1;
    r := FOR p IN InitialData[1]
      newp := FOR k IN 1, D
        RETURNS ARRAY OF InBox ( p[k], BoxSize )
      END FOR
      RETURNS ARRAY OF newp
    END FOR;

    Pairs := LET
      Cells := Distribute ( r, CellSize, BoxSize, Ncellsk, D );
    IN
      VerletTable ( r, Cells, Neighbours, Rv, BoxSize )
    END LET;

    v := InitialData[2];
    K := KineticEnergy ( v );

```

```

a, W, U, Ncut := CalculateForces ( r, Pairs, Rc, BoxSize );

WHILE step <= NumberOfSteps REPEAT
    r, v_half := FirstPass ( old r, old v, old a, BoxSize, Delta );
    Pairs := IF MOD ( old step, TableRemakeInterval ) = 0
THEN
    LET
        Cells := Distribute ( r, CellSize, BoxSize, Ncellsk, D );
    IN
        VerletTable ( r, Cells, Neighbours, Rv, BoxSize )
    END LET
ELSE
    old Pairs
END IF;
a, W, U, Ncut := CalculateForces ( r, Pairs, Rc, BoxSize );
v := SecondPass ( v_half, a, Delta );
K := KineticEnergy ( v );
step := old step + 1;

RETURNS ARRAY OF ARRAY ParticlesData [ 1: r, v, a ]
        WHEN MOD (step, OutputInterval) = 0
        ARRAY OF U / DOUBLE_REAL ( N )
        ARRAY OF K / DOUBLE_REAL ( N )
        ARRAY OF W / DOUBLE_REAL ( N )
        ARRAY OF Ncut

END FOR

END LET

END FUNCTION % Main

```


Somatorial

```
EXPORT MAIN

FUNCTION FACT( X,Y:INTEGER RETURNS INTEGER)

  IF X<=Y THEN Y
  ELSE FACT(X,(X+Y)/2 +1) + FACT((X+Y)/2,Y)
  END IF

END FUNCTION

FUNCTION MAIN(N:integer returns integer)

  FACT(N,1)

END FUNCTION
```

FFT

```

EXPORT MAIN

TYPE COMPLEX=RECORD[RL:REAL;IM:REAL]
TYPE POLY=ARRAY[COMPLEX]
IMPORT DF_COS(R:real returns real)
IMPORT DF_SIN(R:real returns real)
FUNCTION CNEG(CP:COMPLEX RETURNS COMPLEX)

    RECORD COMPLEX [RL:-CP.RL;IM:-CP.IM]

END FUNCTION

FUNCTION CINV(CP:COMPLEX RETURNS COMPLEX)

    LET
        MD:=CP.RL*CP.RL+CP.IM*CP.IM
    IN
        RECORD COMPLEX [RL:CP.RL/MD;IM:-CP.IM/MD]
    END LET

END FUNCTION

FUNCTION CADD(CP1,CP2:COMPLEX RETURNS COMPLEX)

    RECORD COMPLEX [RL:CP1.RL+CP2.RL;IM:CP1.IM+CP2.IM]

END FUNCTION

FUNCTION CMULT(CP1,CP2:COMPLEX RETURNS COMPLEX)

    LET
        CREAL:=CP1.RL*CP2.RL - CP1.IM*CP2.IM;
        CIM:=CP1.RL*CP2.IM + CP1.IM*CP2.RL
    IN
        RECORD COMPLEX [RL:CREAL;IM:CIM]
    END LET

END FUNCTION

FUNCTION PRUNROOT(N:INTEGER RETURNS COMPLEX)

    LET
        PI:=3.141592
    IN
        RECORD COMPLEX [RL:DF_COS(2.0*PI/REAL(N));IM:DF_SIN(2.0*PI/REAL(N))]
    END LET

END FUNCTION

FUNCTION ROOTS(N:INTEGER;W:COMPLEX RETURNS POLY)

    FOR INITIAL
        WP:=RECORD COMPLEX [RL:1.0;IM:0.0];
        I:=0
        WHILE I <= N-2
            REPEAT
                I:=OLD I+1;
                WP:=CMULT(W,OLD WP)
            RETURNS ARRAY OF WP
        END FOR

END FUNCTION

FUNCTION FFT(N:INTEGER;W:COMPLEX;AP:POLY RETURNS POLY)

    IF N=1 THEN ARRAY POLY [0:AP[0]]
    ELSE
        LET
            A1,A2:= LET
                N1:=N/2;
                W2:=CMULT(W,W);
                BP,CP:=FOR J IN 0,N1-1
                e11:=AP[2*J];
                e12:=AP[2*J+1]
                returns array of e11 array of e12
            end for;
            B,C:=FFT(N1,W2,BP),FFT(N1,W2,CP);
            WP:=ARRAY_SETL(ROOTS(N1,W),0)
            IN
                FOR I IN 0,N1-1
                e11:=let
                    waux:=cmult(wp[I],c[I])
                in
                    CADD(B[I],WAUX)
                end let
                returns array of e11
            END FOR,
            FOR I IN 0,N1-1
            e11:=let
                waux:=cmult(wp[I],c[I])
            in
                CADD(B[I],CNEG(WAUX))
            end let
            returns array of e11
        END LET
    END IF

END FUNCTION

FUNCTION MAIN(N:integer RETURNS POLY)

```

```

LET
%      N:=256;
      CO:=RECORD COMPLEX [RL:1.0;IM:0.0];
      C1:=RECORD COMPLEX [RL:0.0;IM:0.0];
      AP1:=ARRAY_FILL(0,N-1,CO);
      AP2:=ARRAY_FILL(0,N-1,C1);
      AP:=AP1||AP2
IN
      FFT(N,PRIMROOT(N),AP)
END LET

END FUNCTION

```

Gauss

```

export Main

type vector = array [real]
type matrix = array [vector]

function TwoToOne(M : Matrix returns vector)
for E1 in M
Row := E1
returns value of concatenate Row
end for
end function % TwoToOne

function SetUpEqns (ne, nr: integer; r: real returns matrix, matrix)

let
eqns :=
for I in 1, ne cross J in 1, ne
e1 := if I = J then r + 1.0 else r end if
returns array of e1
end for;

rhs :=
for I in 1, ne cross J in 1, nr
e1 := r
returns array of e1
end for

in
eqns, rhs
end let

end function % SetUpEqns

function SwapRows (m: matrix; n, r1, r2: integer returns matrix)

% swap rows r1 and r2 of matrix m

for i in 1, n
rownum := if i = r1 then r2
elseif i = r2 then r1
else i end if;
row := m [rownum]
returns array of row
end for

end function % SwapRows

function Eliminate (ne, nr: integer; eqns, rhs: matrix; pivotflag: boolean
returns matrix, matrix, boolean)

% The elimination part - the boolean result is false if matrix is singular

function FindPivot (m: matrix; n, i: integer returns real, integer)

% Find the pivot i.e. the largest number to divide by

for initial
j := i;
pivot := m [i,i];
pivotrow := i
while j < n
repeat
j := old j + 1;
pivot, pivotrow := if abs (old pivot) < abs (m[i,j])
then m[i,j], j
else old pivot, old pivotrow
end if
returns value of pivot
value of pivotrow
end for

end function % FindPivot

for initial
i := 0;
e1 := eqns;
r1 := rhs;
ok := true

repeat

i := old i + 1;

% find pivot and row in which it occurs
pivot, prow := if PivotFlag then FindPivot (old e1, ne, i)
else old e1 [i,i], i end if;

% swap current row with pivotal row if different
e2, r2 := if prow = i | ~ PivotFlag then old e1, old r1
else SwapRows (old e1, ne, i, prow),
SwapRows (old r1, ne, i, prow)
end if;

% a zero pivot indicates a singular matrix
ok := old ok & (abs(pivot) > 1e-4);

% create the next generation of equations and right hand sides
e1, r1 :=

for j in 1, ne
ratio := if ok then e2[j,i] / pivot else 0.0 end if;

```

```

    erow, rrow := if j <= i then e2[j], r2[j] else
        for k in 1, ne
            e1 := if k < i then 0.0 else e2[j,k] - ratio * e2[i,k] end if
            returns array of e1
        end for ,
        for k in 1, nr
            r1 := r2[j,k] - ratio * r2[i,k]
            returns array of r1
        end for
    end if
    returns array of erow
    array of rrow
end for

until i = ne

returns value of e1
value of r1
value of ok

end for

end function % Eliminate

function BackSubstitute (ne, nr: integer; eqns, rhs: matrix returns matrix)

for initial
    i := ne + 1;
    answers := array matrix []

repeat
    i := old i - 1;
    arow :=
    for j in 1, nr
        asum :=
        if i = ne
            then 0.0
        else
            for k in i+1, ne
                partsum := eqns [i,k] * old answers [ne-k+1, j]
            returns value of sum partsum
            end for
        end if;
        ael := (rhs [i,j] - asum) / eqns [i,i]
    returns array of ael
    end for;
    answers := array_addh (old answers, arow)

until i = 1

returns value of answers

end for

end function % BackSubstitute

function Main( returns vector, boolean)

let
    eq, rh := SetUpEqns (8, 8, 2.0);
    triangle, newrhs, ok := Eliminate (8, 8, eq, rh, true);
    answ := if ok
        then TwoToOne ( BackSubstitute (8, 8, triangle, newrhs))
        else array vector [] end if

in

    answ, ok

end let

end function % Gauss

```

Interpolação

```

EXPORT MAIN

TYPE POLY=ARRAY[real]
TYPE VECTOR=ARRAY[real]

FUNCTION HORNER(A:VECTOR;N:INTEGER;V:REAL RETURNS REAL)

  for initial
    sm:=A[N+1];
    i:=N
    while i>=1
      repeat
        i:=old i - 1;
        sm:=v*old sm + A[old i]
      returns value of sm
    end for
  END FUNCTION

FUNCTION PMULT(A,B:POLY RETURNS POLY)

let
  na:=array_size(a);
  nb:=array_size(b)
in
  for k in 1,na+nb-1
    cf:= for i in 1,na
      el:= for j in 1,nb
        el1:=if i+j=(k+1) then A[i]*B[j]
        else 0.0
        end if
      returns value of sum el1
    end for
  returns value of sum el
end for
  returns array of cf
end for
end let
END FUNCTION

FUNCTION PADD(A,B:POLY RETURNS POLY)

let
  na:=array_size(a);
  nb:=array_size(b)
in
  if na<nb then for i in 1,nb
    el:=if i<=na then A[i]+B[i]
    else B[i]
    end if
  returns array of el
end for
  else for i in 1,na
    el:=if i<=nb then A[i]+B[i]
    else A[i]
    end if
  returns array of el
end for
end if
end let
END FUNCTION

FUNCTION INTERP(X,Y:VECTOR;N:INTEGER RETURNS POLY)

for initial
  i:=2;
  AMS:=array [1:Y[i]];
  D:=array [1:-X[i],1.0]
while i<=n
  repeat
    i:=old i + 1;
    AMS,D:=let
      denom:=horner(old D,old i-1,X[old i]);
      num:=horner(old AMS,old i-2,X[old i])
    in
      PADD(PMULT(ARRAY[1:(Y[old i]-num)/denom],old D),old AMS),
      PMULT(old D,array[1:-X[old i],1.0])
    end let
  returns value of AMS
end for

END FUNCTION

FUNCTION MAIN (RETURNS POLY)

LET
  A:=ARRAY[1:1.0,2.0,3.0,4.0,5.0,6.0,-1.0,0.0,9.0,-3.45,0.0302,12.76,
    34.9,45.0];
  B:=ARRAY[1:13.0,56.0,1377.0,12178.0,62741.0,234948.0,-7.0,6.0,-20.0,
    0.3,3.5456,-84.0,29.0,8.634]
IN
  INTERP(A,B,ARRAY_SIZE(A))
END LET

END FUNCTION

```

Laplace

```

export Main

type Vector = array[real]
type Matrix = array[Vector]

function Fill(N : integer; R:real returns Matrix)

let
Row := array_fill(1,N,r)
in array_fill(1,N,Row)
end let

end function

function lapl(M:matrix;N,KK:integer returns matrix)
for initial
K:=1;
M1:=M
REPEAT
K := OLD K +1;
M1:=
for I in 1,N
Row:=
for J in 1,N
el:=
if (((I>1)&(I<=N-1)) & ((J>1)&(J<=N-1))) then
(old M1[I,J])/2.0
+ (old M1[I-1,J])/8.0
+ (old M1[I+1,J])/8.0
+ (old M1[I,J-1])/8.0
+ (old M1[I,J+1])/8.0
else
(old M1[I,J])
end if
returns array of el
end for
returns array of row
end for
UNTIL K=KK
returns value of M1
end for
end function

function Main (N:integer returns real)
LET
M1:=fill(N,0.5);
M2:=lapl(M1,N,N+1) % N era 6 e N+1 era 7 na versao original
IN
M2[N,N]
end let
end function

```

Multiplicação de Matrizes

```
export Main

type Vector = array[real]
type Matrix = array[Vector]

function Fill(N1, N2 : integer; R:real returns Matrix)

let
Row := array_fill(1,N2,R)
in array_fill(1,N1,Row)
end let

end function

function matmult(A, B : Matrix; N1, N2, N : integer returns Matrix)

for I in 1,N1
Row :=
for J in 1, N2
E1 :=
for K in 1,N
R := A[I,K] * B[K,J]
returns value of sum R
end for
returns array of E1
end for
returns array of Row
end for
end function

function Main(N:integer returns real)

let
N1 := Fill( N,N, 0.5);
N2 := Fill( N,N, 1.2);
N3 := Matmult(N1, N2 , N,N,N)
in N3[N,N]\
end let
end function
```


N-Queens

```
% This program solves the N-Queens problem with a divide-and-conquer
% algorithm. Any time a queen is successfully placed in a row, it
% searches, in parallel, solutions that have a queen placed either on the
% left or on the right side of the next row. The solutions are eventually
% merged. This is a highly parallel algorithm that can be used to test
% the throttle mechanism.

% The program receives as input the size of the board, N. Positions of
% the board are numbered left-to-right, top-down from 1 to N*N. The
% solutions are coded as N-digit integers, each digit indicating the
% columns on which the queens are placed in successive rows.

export main

type ivector = array [integer]
type istream = stream [integer]

function row ( dim, position : integer returns integer )
    (position - 1) / dim + 1
end function

function col ( dim, position : integer returns integer )
    mod ( (position - 1), dim ) + 1
end function

function leftmostposition ( dim, line, side: integer returns integer )
    dim * ( line - 1 ) + 1 + side * (dim / 2)
end function

function rightmostposition ( dim, line, side: integer returns integer )
    dim * ( line - 1 ) + ( side + 1 ) * dim / 2
end function

function compatible ( dim, position, queen : integer returns boolean )
% Checks if a position is compatible with a particular queen
    let
        pcol := col (dim,position);
        qcol := col (dim,queen);
        prow := row (dim,position);
        qrow := row (dim,queen)
    in
        % the position is compatible with a queen if they are in
        (pcol ~= qcol) & % different columns and in
        (prow ~= qrow) & % different rows and in
        (abs (pcol-qcol) ~= abs (prow-qrow) ) % different diagonals
    end let
end function

function possible ( dim, position: integer; board: ivector returns boolean )
% Checks if a position is compatible with the queens already placed
    for queen in board
        ok := compatible ( dim, position, queen )
        returns value of product ok
    end for
end function

function code ( dim : integer; board: ivector returns integer )
% Codes a solution from a vector of queens into a integer
    for initial
        c := 0;
        i := 0
        while i < dim repeat
            i := old i + 1;
            c := old c * 10 + col ( dim, board [i])
            returns value of c
        end for
    end function

function find ( dim, line, side: integer; board: ivector returns istream )
% For a BOARD that has already LINE queens placed, this function finds
% solutions with a queen placed either in the left or right SIDE of the
% next line. Whenever a solution is found the function recurses. If a
% solution is not found an empty array is returned that will later be
% absorbed by the catenate reduction operator.
    for position in leftmostposition ( dim, line+1, side ),
        rightmostposition ( dim, line+1, side )
    next :=
        if possible ( dim, position, board ) then
            let
                nb := array_addh ( board, position )
            in
                if line = dim-1 then
                    stream [ code ( dim, nb ) ]
                else
                    find (dim, line+1, 0, nb) || find (dim, line+1, 1, nb)
                end if
            end let
        else
            stream istream []
        end if
    end if
end function
```

```
returns value of catenate next
end for
end function
```

```
function findall ( dim: integer; board: ivector returns istream )
```

```
% Finds all the solutions starting from an empty board
```

```
let
  ll := 0;
  rr := 1;
  line := 0
in
  find ( dim, line, ll, board ) || find ( dim, line, rr, board )
end let
end function
```

```
function main (dim:integer returns istream )
```

```
let
  dim := 6;
  board := array ivector □
in
  findall ( dim, board )
end let
end function
```

Quicksort

```
export main
type vector = array[integer]

FUNCTION QUICKSORT( A:vector returns vector)

    FUNCTION SPLIT ( A: vector returns vector,vector,vector)

    for X IN A
    left_elem:=if X < A[1] then array[1:x]
                else array vector []
                end if;
    mid_elem:=if X = A[1] then array[1:x]
                else array vector []
                end if;
    right_elem:=if X > A[1] then array[1:x]
                 else array vector []
                 end if
    returns value of catenate left_elem
    value of catenate mid_elem
    value of catenate right_elem
    end for
    END FUNCTION
    IF array_size(A) < 2 then A
        else let
            l,middle,r:=split(A)
            in
                quicksort(l)||middle||quicksort(r)
            end let
        end if
    END FUNCTION

FUNCTION MAIN (returns vector)

let

    A:=ARRAY [1:1,2,4,5,8,10,13,5,6,7,9,11,12,14,15,16,17,18,19,20]

in

    quicksort(A)

end let

END FUNCTION
```

Simple

```
%
% The following program is a version of the Simple code written in
% the data flow language SISAL. This version of Simple is actually
% Simple-Version 01, at LLNL. This program has been run through a
% formatter to help highlight several aspects of the program. All
% reserved words are typed in lower case letters and boldface font.
% Function names are listed on the right side of the page in italic
% font. Comments throughout the program are in italics.
%
% WARNINGS :
%
% 1. This is a first draft of the program—we have not yet
% run the program through a syntax checker to verify the
% consistency of the program with the reference manual.
%
% 2. The writing was done by primarily transliterating a version
% of Simple written in VAL ( WBA March 3, 1980 ). I have made
% changes to incorporate the special features of SISAL, but
% also changed the structures of some data for added convenience.
% The code inside the "reflection" functions has also been altered
% for simplicity—these routines could have been done this way
% in VAL as well.
%
% 3. I have made a few algorithm adjustments from the VAL version
% based on discussions with Chris Hendrickson, regarding simpler
% and/or more parallel ways of expressing some calculations.
%
%
% The primary purpose of this program is to illustrate SISAL in a
% non-trivial setting. Please DO NOT view it as a completed work
% with all of the bugs out -- I cannot guarantee that, in fact I am
% certain that there are a few remaining technical problems with
% correctly implementing the Simple algorithm. If you have questions,
% comments, or suggestions please contact me.
%
%
% James R. McGraw
% Lawrence Livermore National Laboratory
% Box 808, L-306
% Livermore, CA 94550
%
% (415) 422-0541
% mcgraw@lbl-csam
%
%
% EXPORT Main
%
% TYPE areal = ARRAY [ REAL ]
%
% TYPE aareal = ARRAY [ Areal ]
%
% TYPE RZpair = RECORD [ R, Z : REAL ] % Cylindrical coord. system
%
% TYPE aRZpair = ARRAY [ RZpair ]
%
% TYPE aaRZpair = ARRAY [ aRZpair ]
%
% TYPE UWpair = RECORD [ U, W : REAL ] % Velocities in R,Z directions
%
% TYPE aUWpair = ARRAY [ UWpair ]
%
% TYPE aaUWpair = ARRAY [ aUWpair ]
%
% TYPE Data = RECORD [
% Cycles_Out : INTEGER; Time_Out : REAL;
% dT_Out : REAL; dTC_Out : REAL;
% dTE_Out : REAL; ETot_Out : REAL;
% IE_Out : REAL; KE_Out : REAL;
% HM_Out : REAL; WM_Out : REAL;
% CM_Out : REAL; dCM_Out : REAL;
% dTNP_Out : REAL; dTM_Out : REAL;
% RZ_Out : aRZpair; UW_Out : aaUWpair;
% Rho_Out : aareal; E_Out : aareal;
% P_Out : aareal; Q_Out : aareal;
% AJ_Out : aareal; Theta_Out : aareal ]
%
% TYPE adata = ARRAY [ Data ]
%
% TYPE Coeffs = RECORD [ Aes, Bes, Ces, Des, Ees, Fes, Ges, Hes, Pes : aareal ]
%
% TYPE Bdy = RECORD [ Reflect : BOOLEAN;
% Press : REAL ]
%
% TYPE PE_Info = RECORD [ CC : Coeffs;
% Tes,
% Res : areal;
% Topt,
% Topr : INTEGER ]
%
% TYPE Global = RECORD [
% Pressure : PE_Info; Energy : PE_Info;
% Kmin : INTEGER; Kmax : INTEGER;
% Lmin : INTEGER; Lmax : INTEGER;
% Nbcu : Bdy; Nbcv : Bdy;
% Nbcd : Bdy; Nbcd : Bdy ]
%
% TYPE Indx = RECORD [ Nt, Nr : INTEGER ]
%
% TYPE aaindx = ARRAY [ ARRAY [ Indx ] ]
%
% import DF_Square ( x : real RETURNS real)
%
% import DF_Sqrt( x : real RETURNS real)
%
% import DF_Cos ( x : real RETURNS real)
%
% import DF_Sin ( x : real RETURNS real)
```

```

FORWARD FUNCTION AJ_Calc ( X : aaRZpair;
K,
L : INTEGER
RETURNS REAL,
REAL )

FORWARD FUNCTION Boundary_Heat_Flow ( Temp,
Cbb,
Dbb : saareal;
World : Global
RETURNS REAL )

FORWARD FUNCTION Conduct ( X : aaRZpair;
Rho,
CC,
Sigma,
Temp,
Mass : saareal;
Locs : aaindx;
dTNPB : REAL;
World : Global
RETURNS saareal,
saareal,
REAL,
REAL,
saareal,
aaindx,
REAL )

FORWARD FUNCTION Gen ( World : Global
RETURNS aaRZpair,
saareal,
saareal,
saareal,
saareal,
saareal,
saareal,
saareal,
aaindx )

FORWARD FUNCTION Hwork ( X : aaRZpair;
V : aaUWpair;
P,
Q : saareal;
dTNPB : REAL;
World : Global
RETURNS REAL )

FORWARD FUNCTION Hydro_Energy ( X : aaRZpair;
V : aaUWpair;
Rho,
Mass,
P,
Energy,
UV2 : saareal;
Last_Indx : aaindx;
dTNPB : REAL;
World : Global
RETURNS saareal,
saareal,
saareal,
saareal,
saareal,
saareal,
REAL,
REAL,
aaindx,
REAL )

FORWARD FUNCTION Lookup_E_dEdTh ( Theta,
Rho : REAL;
N : Indx;
World : Global
RETURNS REAL,
REAL,
Indx )

FORWARD FUNCTION Lookup_Energy ( Theta,
Rho : REAL;
N : Indx;
World : Global
RETURNS REAL,
Indx )

FORWARD FUNCTION Lookup_EOS ( Theta,
Rho : REAL;
N : Indx;
Tables : PE_Info
RETURNS REAL,
REAL,
REAL,
Indx )

FORWARD FUNCTION Lookup_Pressure ( Theta,
Rho : REAL;
N : Indx;
World : Global
RETURNS REAL,
Indx )

FORWARD FUNCTION New_Q (X : aaRZpair;
V : aaUWpair;
P_KL,
Rho_KL,
Mass_KL,
AJ_KL : REAL;
K,
L : INTEGER;
Gamma : REAL

```

```

RETURNS REAL,
REAL )

FORWARD FUNCTION Node_Reflect ( X : aaRZpair;
World : Global
RETURNS aaRZpair )

FORWARD FUNCTION Poly_Eval ( CC : Coeffs;
Tind,
Rind : INTEGER;
Targ,
Rarg : REAL
RETURNS REAL )

FORWARD FUNCTION Project ( RZO,
RZ1,
RZp : RZpair
RETURNS RZpair )

FORWARD FUNCTION Reflect ( Info : aareal;
World : Global
RETURNS aareal )

FORWARD FUNCTION RZ_Calc ( X : aaRZpair;
V : aaUPpair;
P,
Q,
Rho_J : aareal;
dTH,
dTMPH : REAL;
World : Global
RETURNS aaRZpair,
aaUPpair,
aareal )

FORWARD FUNCTION Table_Lookup ( Table : aareal;
Elem : REAL;
Start : INTEGER
RETURNS INTEGER )

FORWARD FUNCTION Temp_Calc ( Evalve,
Rho : REAL;
M : Indx;
World : Global
RETURNS REAL,
Indx )

FORWARD FUNCTION Zone_Reflect ( Rho,
Rho_J,
Q,
P : aareal;
World : Global
RETURNS aareal,
aareal,
aareal,
aareal )

FUNCTION Simple ( Cycles, Kmin, Kmax, Lmin, Lmax : INTEGER
RETURNS Data )

LET
Gamma := 1.6667; % Thermodynamic coefficient for an ideal gas
CsubV := 0.1; % Specific heat at constant volume of ideal gas
Const := CsubV * ( Gamma - 1.0 );
Zeroes := ARRAY_FILL( 1, 2, ARRAY_FILL( 1, 3, 0.0 ) );
Consts := ARRAY_FILL( 1, 2, ARRAY_FILL( 1, 3, Const ) );
CsubVs := ARRAY_FILL( 1, 2, ARRAY_FILL( 1, 3, CsubV ) );

World := RECORD Global [
Pressure: RECORD PE_Info [
CC : RECORD Coeffs [
Aes : Zeroes; Bes : Zeroes;
Ces : Zeroes; Des : Zeroes;
Ees : Zeroes; Fes : Consts;
Ges : Zeroes; Hes : Zeroes;
Pes : Zeroes ];
Tes : ARRAY areal [ 1 : 0.0, 1.0, 100.0 ];
Res : ARRAY areal [ 1 : 0.0, 3.0, 300.0, 3.0e+10 ];
Topt : 3;
Topr : 4 ];
Energy : RECORD PE_Info [
CC : RECORD Coeffs [
Aes : Zeroes; Bes : Zeroes;
Ces : CsubVs; Des : Zeroes;
Ees : Zeroes; Fes : Zeroes;
Ges : Zeroes; Hes : Zeroes;
Pes : Zeroes ];
Tes : ARRAY areal [ 1 : 0.0, 1.0, 100.0 ];
Res : ARRAY areal [ 1 : 0.0, 3.0, 300.0, 3.0e+10 ];
Topt : 3;
Topr : 4 ];
Kmin : Kmin;
Kmax : Kmax;
Lmin : Lmin;
Lmax : Lmax;
Nbcu : RECORD Bdy [ Reflect : TRUE; Press : 0.0 ];
Nbcr : RECORD Bdy [ Reflect : TRUE; Press : 0.0 ];
Nbcl : RECORD Bdy [ Reflect : TRUE; Press : 0.0 ];
Nbcd : RECORD Bdy [ Reflect : TRUE; Press : 0.0 ];
dTmax := 0.01;
dTmin := 0.001;
Cycles_Per_Print:= 1
IN
FOR INITIAL
X, P, Rho, Rho_J, Temp, Energy, Mass, Locs := Gen(World);

```



```

:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;
PI := 3.14159265;
% EDDT := 1.0e-8;
% Tmax := 1.0e8; } these three values never used
% Cmax := 10;
Rzero := 1.4; % initial density
Tzero := 10.0; % initial temperature

% Generate the coordinates r and z
% This algorithm generates a spherical shell between -pi/2
% and pi/2 with a radius of 10 cm

RP := REAL(Lmax - Lmin);

X := FOR K IN Kmin, Kmax CROSS L IN Lmin, Lmax
Z1 := REAL(10 + K - Kmin);
ZZ := (-0.5 + REAL(L - Lmin) / RP) * PI;
R_Data := Z1 * DF.Cos(ZZ);
Z_Data := Z1 * DF.Sin(ZZ);
X_Data := RECORD [ R : R_Data; Z : Z_Data ]
RETURNS
ARRAY OF X_Data
END FOR;

Start := RECORD [ Nt : 1; Nr : 1 ];
Pzero, S1
:= Lookup_Pressure(Tzero, Rzero, Start, World);
Ezero, S3
:= Lookup_Energy(Tzero, Rzero, Start, World);
P := ARRAY_FILL( Kmin+1, Kmax, ARRAY_FILL(Lmin+1, Lmax, Pzero) );
Rho := ARRAY_FILL( Kmin+1, Kmax, ARRAY_FILL(Lmin+1, Lmax, Rzero) );
Temp := ARRAY_FILL( Kmin+1, Kmax, ARRAY_FILL(Lmin+1, Lmax, Tzero) );
Energy := ARRAY_FILL( Kmin+1, Kmax, ARRAY_FILL(Lmin+1, Lmax, Ezero) );
Locs := ARRAY_FILL( Kmin+1, Kmax, ARRAY_FILL(Lmin+1, Lmax, S1 ) );

Rho_J, Mass
:= FOR K IN Kmin+1, Kmax CROSS L IN Lmin+1, Lmax
AJval, Vol := AJ_Calc(X, K, L);
Mvalu := Rzero * Vol;
RJval := Rzero * AJval
RETURNS
ARRAY OF RJval
ARRAY OF Mvalu
END FOR
IN
X, P, Rho, Rho_J, Temp, Energy, Mass, Locs
END LET
END FUNCTION % Gen

FUNCTION AJ_Calc ( X : aaZpair;
K,
L : INTEGER
RETURNS REAL,
REAL )

LET
Curr := X [ K , L ];
Back := X [ K-1, L ];
Down := X [ K , L-1 ];
BkDn := X [ K-1, L-1 ];

AJ1 := Curr.R * ( Back.Z - Down.Z ) +
Back.R * ( Down.Z - Curr.Z ) +
Down.R * ( Curr.Z - Back.Z );

AJ3 := Back.R * ( BkDn.Z - Down.Z ) +
BkDn.R * ( Down.Z - Back.Z ) +
Down.R * ( Back.Z - BkDn.Z );

AJval := (AJ1 + AJ3) / 2.0;

Vol := ( AJ1 * ( Curr.R + Back.R + Down.R ) +
AJ3 * ( Back.R + BkDn.R + Down.R ) ) / 6.0
IN
AJval, Vol
END LET
END FUNCTION

FUNCTION Lookup_Pressure ( Theta,
Rho : REAL;
N : indx;
World : Global
RETURNS REAL,
indx )

LET
Func, Extt, Extr, NewN := Lookup_EDS(Theta, Rho, N, World.Pressure)
IN
Func * Extt * Extr, NewN
END LET
END FUNCTION % Lookup_Pressure

FUNCTION Lookup_Energy (Theta,
Rho : REAL;
N : indx;
World : Global
RETURNS REAL,
indx )

LET
Func, Extt, Extr, NewN := Lookup_EDS(Theta, Rho, N, World.Energy)
IN
Func * Extt, NewN
END LET
END FUNCTION % Lookup_Energy

FUNCTION Lookup_EDS ( Theta,
Rho : REAL;
N : indx;

```



```

Tables : PE_Info
RETURNS  REAL,
        REAL,
        REAL,
        Indx )

LET
Tes := Tables.Tes;
Res := Tables.Res;
Topt := Tables.Topt;
Topr := Tables.Topr;
CC := Tables.CC;
NT_Loc := Table_Lookup(Tes, Theta, N.Nt);
NR_Loc := Table_Lookup(Res, Rho, N.Nr);

NewT, ExtT, Targ:= IF NT_Loc = 0
THEN
1, Theta / Tes[1], Tes[1]
ELSEIF NT_Loc = Topt
THEN
Topt-1, Theta / Tes[Topt], Tes[Topt]
ELSE
NT_Loc, 1.0, Theta
END IF;

NewR, ExtR, Rarg:= IF NR_Loc = 0
THEN
1, Rho / Res[1], Res[1]
ELSEIF NR_Loc = Topr
THEN
Topr - 1, Rho / Res[Topr], Res[Topr]
ELSE
NR_Loc, 1.0, Rho
END IF;

Energy := Poly_Eval(CC, NewT, NewR, Targ, Rarg)
IN
Energy, ExtT, ExtR, RECORD [ Nt : NewT; Nr : NewR ]
END LET
END FUNCTION % Lookup_EOS

FUNCTION Poly_Eval ( CC : Coeffs;
Tind,
Rind : INTEGER;
Targ,
Rarg : REAL
RETURNS  REAL )

% The following calculation is actually done using Horner's rule
% in the VAL version of Simple, and I assume it is done that way
% in the Fortran version as well. The version below is just more
% readable (in my humble opinion).

CC.Aes[ Tind, Rind ]
+ CC.Bes[ Tind, Rind ] * Rarg
+ CC.Ces[ Tind, Rind ] * Targ
+ CC.Des[ Tind, Rind ] * Rarg * Rarg
+ CC.Ees[ Tind, Rind ] * Targ * Targ
+ CC.Fes[ Tind, Rind ] * Rarg * Targ
+ CC.Ges[ Tind, Rind ] * Rarg * Rarg * Targ
+ CC.Hes[ Tind, Rind ] * Rarg * Targ * Targ
+ CC.Pes[ Tind, Rind ] * Rarg * Rarg * Targ * Targ

END FUNCTION % Poly_Eval

FUNCTION Lookup_E_dEdTh ( Theta,
Rho : REAL;
N : Indx;
World : Global
RETURNS  REAL,
        REAL,
        Indx )

LET
Tes := World.Energy.Tes;
Res := World.Energy.Res;
Topt := World.Energy.Topt;
Topr := World.Energy.Topr;
CC := World.Energy.CC;
Nt := N.Nt;
Nr := N.Nr;
NT_Loc := Table_Lookup(Tes, Theta, Nt);% WBA::Simple used res
NR_Loc := Table_Lookup(Res, Rho, Nr);

NewR, Rarg := IF NR_Loc = 0
THEN
1, Res[1]
ELSEIF NR_Loc = Topr
THEN
Topr - 1, Res[Topr]
ELSE
NR_Loc, Rho
END IF;

NewT, ExtT, Targ, Limit_Case
:= IF NT_Loc = 0
THEN
1, Theta / Tes[1], Tes[1], TRUE
ELSEIF NT_Loc = Topt
THEN
Topt - 1, Theta / Tes[Topt], Tes[Topt], TRUE
ELSE
NT_Loc, 1.0, Theta, FALSE
END IF;

Energy := Poly_Eval(CC, NewT, NewR, Targ, Rarg) * ExtT;

dE_dTheta := IF Limit_case
THEN

```

```

Energy / Theta % ideal gas approx.
ELSE % derivative of energy poly.
% faster with Horner's rule
CC.Ces[ NewT, NewR ] +
CC.Fes[ NewT, NewR ] * Rarg +
CC.Ges[ NewT, NewR ] * Rarg * Rarg +
( CC.Ees[ NewT, NewR ] +
CC.Hes[ NewT, NewR ] * Rarg +
CC.Pes[ NewT, NewR ] * Rarg * Rarg ) *
(2.0 * Theta)
END IF
IN
Energy, dE_dTheta, RECORD [ Nt : NewT; Nr : NewR ]
END LET
END FUNCTION % Lookup_E_dEdTh

FUNCTION Temp_Calc ( Evalue,
Rho : REAL;
N : Indx;
World : Global
RETURNS REAL,
Indx )

LET
CVideal := 0.1;
pim10 := 1.0e-6;
Efloor, NewIndx := Lookup_Energy(0.0, Rho, N, World)
IN
IF Evalue < Efloor
THEN
0.0, NewIndx
ELSE
FOR INITIAL
Targ := Evalue / CVideal;
Locs := NewIndx
REPEAT
Energy, dEdTheta, Locs
:= Lookup_E_dEdTh(OLD Targ, Rho, OLD Locs, World);
dTemp := (Evalue - Energy) / dEdTheta;
Targ := OLD Targ + dTemp
UNTIL (ABS(dTemp) < pim10) | (Targ < pim10)
RETURNS
VALUE OF Targ
VALUE OF Locs
END FOR
END IF
END LET
END FUNCTION % Temp_Calc

FUNCTION Hwork (X : aaRZpair;
V : aaUWpair;
P,
Q : aaReal;
dTNPH : REAL;
World : Global
RETURNS REAL )

% This routine calculates the work done on the boundary by
% the hydrodynamics.

LET
Kmin, Kmax, Lmin, Lmax
:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;

Work1 :=
FOR K IN Kmin+1, Kmax
Wrk_K :=
( P[K,Lmin+1] + P[K,Lmin] + Q[K,Lmin+1] + Q[K,Lmin])
* ( (V[K-1,Lmin].U + V[K,Lmin].U)*(X[K,Lmin].Z - X[K-1,Lmin].Z)
-(V[K-1,Lmin].W + V[K,Lmin].W)*(X[K,Lmin].R - X[K-1,Lmin].R))
+ ( X[K-1,Lmin].R + X[K,Lmin].R )
- ( P[K,Lmax+1] + P[K,Lmax] + Q[K,Lmax+1] + Q[K,Lmax])
+ ( (V[K-1,Lmax].U + V[K,Lmax].U)*(X[K,Lmax].Z - X[K-1,Lmax].Z)
-(V[K-1,Lmax].W + V[K,Lmax].W)*(X[K,Lmax].R - X[K-1,Lmax].R))
* ( X[K-1,Lmax].R + X[K,Lmax].R )
RETURNS
VALUE OF SUM Wrk_K
END FOR;

Work2 :=
FOR L IN Lmin+1, Lmax
Wrk_L :=
( P[Kmin+1,L] + P[Kmin,L] + Q[Kmin+1,L] + Q[Kmin,L])
* ( (V[Kmin,L-1].U + V[Kmin,L].U)*(X[Kmin,L].Z - X[Kmin,L-1].Z)
-(V[Kmin,L-1].W + V[Kmin,L].W)*(X[Kmin,L].R - X[Kmin,L-1].R))
+ ( X[Kmin,L-1].R + X[Kmin,L].R )
- ( P[Kmax+1,L] + P[Kmax,L] + Q[Kmax+1,L] + Q[Kmax,L])
* ( (V[Kmax,L-1].U + V[Kmax,L].U)*(X[Kmax,L].Z - X[Kmax,L-1].Z)
-(V[Kmax,L-1].W + V[Kmax,L].W)*(X[Kmax,L].R - X[Kmax,L-1].R))
+ ( X[Kmax,L-1].R + X[Kmax,L].R )
RETURNS
VALUE OF SUM Wrk_L
END FOR
IN
(Work1 + Work2) * dTNPH / 8.0
END LET
END FUNCTION % Hwork

FUNCTION Project ( RZO,
RZ1,
RZp : RZpair
RETURNS RZpair )

LET
WW := 2.0 * ( RZ1.Z - RZO.Z )
/ ((RZ1.R - RZO.R)*(RZ1.R - RZO.R)
+ (RZ1.Z - RZO.Z)*(RZ1.Z - RZO.Z) );

```

```

Alpha := 1.0 - ( RZ1.Z - RZO.Z ) * WW;
Beta := ( RZ1.R - RZO.R ) * WW
IN
RECORD [ R : RZO.R + (RZp.R - RZO.R)*Alpha + (RZp.Z - RZO.Z)*Beta;
Z : RZO.Z + (RZp.R - RZO.R)*Beta + (RZp.Z - RZO.Z)*Alpha ]
END LET
END FUNCTION % Project

FUNCTION Node_Reflect ( X : aaRZpair;
World : Global
RETURNS aaRZpair )

% This FUNCTION builds a boundary around the existing RZ structure.
% There are quite a few special case nodes that require individual
% handling. The picture below should help decode the actions of this
% particular FUNCTION.
%
% Kmin --K Direction-- Kmax
%
% ' C1 : Top_Row : TL : C2 '
% '.....|-----|-----|-----|.....'
% Lmax ' LF | RF '
% '.....|-----|-----|-----|.....'
% L ' L | R '
% ' L | R '
% D ' e | i '
% i ' f | Old RZ ARRAY | g '
% r ' t | h '
% e ' | t '
% c ' S | '
% t ' i | S '
% i ' d | i '
% o ' e | d '
% n ' | e '
% ' Lmin '
% '.....|-----|-----|-----|.....'
% ' C3 : Bottom Row BL : C4 '
% '-----|-----|-----|-----|-----'
%
%
% Also note, that there is additional complexity caused by having
% ARRAYS stored as vectors of vectors. In this case, the result
% must be an array of columns (to accomodate later subscripting).

LET
Kmin, Kmax, Lmin, Lmax
:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;

LF := Project( X[Kmin,Lmax], X[Kmin,Lmax-1], X[Kmin+1,Lmax] );
TL := Project( X[Kmax,Lmax], X[Kmax-1,Lmax], X[Kmax,Lmax-1] );
RF := Project( X[Kmax,Lmax], X[Kmax,Lmax-1], X[Kmax-1,Lmax] );
BL := Project( X[Kmax,Lmin], X[Kmax-1,Lmin], X[Kmax,Lmin+1] );

Left_Side := FOR L IN Lmin, Lmax-1
Elem := Project( X[Kmin,L], X[Kmin,L+1],
X[Kmin+1,L] )
RETURNS
ARRAY OF Elem
END FOR;

Right_Side := FOR L IN Lmin, Lmax-1
Elem := Project( X[Kmax,L], X[Kmax,L+1],
X[Kmax-1,L] )
RETURNS
ARRAY OF Elem
END FOR;

Full_Left := Left_Side || ARRAY [ Lmax : LF ];
Full_Right := Right_Side || ARRAY [ Lmax : RF ];

Some_Cols := FOR K IN Kmin, Kmax-1
Top := Project( X[K,Lmax], X[K+1,Lmax],
X[K,Lmax-1] );
Bot := Project( X[K,Lmin], X[K+1,Lmin],
X[K,Lmin+1] );
Col := ARRAY [ Lmin-1 : Bot ]
|| X[K] % old center
|| ARRAY [ Lmax+1 : Top ]
RETURNS
ARRAY OF Col
END FOR;

Center := array_addh ( Some_Cols, ( ARRAY [ Lmin-1 : BL ]
|| X[Kmax]
|| ARRAY [ Lmax+1 : TL ]));

% The following assignments may be necessary in a slightly different
% version of the Simple code, but in this particular program, the corner
% values are never accessed (in the RZ information). Therefore, the
% corner values have actually been set to zeroes.
%
% C1 := Project( Center[Kmin,Lmax], Full_Left[Lmax],
Full_Left[Lmax-1] );
% C2 := Project( Center[Kmax,Lmax], Full_Right[Lmax],
Full_Right[Lmax-1] );
% C3 := Project( Center[Kmin,Lmin], Center[Kmin, Lmin-1],
Center[Kmin+1,Lmin-1] );
% C4 := Project( Center[Kmax,Lmin], Full_Right[Lmin],
Full_Right[Lmin-1] );

zero_RZ := record RZpair [R : 0.0; Z : 0.0];
C1, C2, C3, C4 := zero_RZ, zero_RZ, zero_RZ, zero_RZ;

Lft := ARRAY [ Lmin-1: C3 ] || Full_Left ||
ARRAY [ Lmax+1: C1 ];
Rght := ARRAY [ Lmin-1: C4 ] || Full_Right ||

```

```

ARRAY [ Lmax+1: C2 ]
IN
ARRAY [ Kmin-1 : Lft ] || Center || ARRAY [ Kmax+1 : Rght ]
END LET
END FUNCTION % Node_Reflect

FUNCTION Reflect ( Info : aareal;
World : Global
RETURNS aareal )

LET
Kmin, Kmax, Lmin, Lmax
:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;
Top_L := ARRAY [ Lmax+1 : Info [ Kmin+1, Lmax ] ];
Top_R := ARRAY [ Lmax+1 : Info [ Kmax, Lmax ] ];
Bot_L := ARRAY [ Lmin : Info [ Kmin+1, Lmin+1 ] ];
Bot_R := ARRAY [ Lmin : Info [ Kmax, Lmin+1 ] ];

Left_Side := Bot_L || Info[ Kmin+1 ] || Top_L;
Right_Side := Bot_R || Info[ Kmax ] || Top_R;

Middle := FOR K IN Kmin+1, Kmax
Col := ARRAY [ Lmin : Info [ K, Lmin+1 ] ]
|| Info [ K ]
|| ARRAY [ Lmax+1:Info[K,Lmax]]
RETURNS
ARRAY OF Col
END FOR
IN
ARRAY [ Kmin : Left_Side ] || Middle || ARRAY[ Kmax+1 : Right_Side ]
END LET
END FUNCTION % Reflect

FUNCTION Zone_Reflect ( Rho,
Rho_J,
Q,
P : aareal;
World : Global
RETURNS aareal,
aareal,
aareal )

% Computing reflections for Rho and Rho_J are straightFORWARD; the
% the existing boundaries are copied to the outer edge. Notice that
% each old corner gets used three times: once for a side boundary,
% once for a top or bottom edge and once for the new corner.
%
% Refelctions for P and Q are more difficult. Each edge can reflect
% or not, independently (based on World information). The structure
% of the solution is similar to the picture in Node_Reflect, but here
% there are generally fewer cases.

LET
New_Rho := Reflect(Rho, World);
New_RhoJ := Reflect(Rho_J, World);
Kmin, Kmax, Lmin, Lmax
:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;

P_C1, Q_C1 := IF World.Wbcu.Reflect THEN
P[ Kmin+1, Lmax ], Q[ Kmin+1, Lmax ]
ELSE World.Wbcu.Press, 0.0
END IF;

P_C2, Q_C2 := IF World.Wbcr.Reflect THEN
P[ Kmax, Lmax ], Q[ Kmax, Lmax ]
ELSE World.Wbcr.Press, 0.0
END IF;

P_C3, Q_C3 := IF World.Wbcd.Reflect THEN
P[ Kmin+1, Lmin+1 ], Q[ Kmin+1, Lmin+1 ]
ELSE World.Wbcd.Press, 0.0
END IF;

P_C4, Q_C4 := IF World.Wbcr.Reflect THEN
P[ Kmax, Lmin+1 ], Q[ Kmax, Lmin+1 ]
ELSE World.Wbcr.Press, 0.0
END IF;

P_LS, Q_LS := IF World.Wbcl.Reflect THEN
P[ Kmin+1 ], Q[ Kmin+1 ]
ELSE ARRAY_FILL(Lmin+1, Lmax, 0.0),
ARRAY_FILL(Lmin+1, Lmax, World.Wbcl.Press)
END IF;

P_RS, Q_RS := IF World.Wbcr.Reflect THEN
P[ Kmax ], Q[ Kmax ]
ELSE ARRAY_FILL(Lmin+1, Lmax, 0.0),
ARRAY_FILL(Lmin+1, Lmax, World.Wbcr.Press)
END IF;

P_Left := ARRAY[ Lmin : P_C3 ] || P_LS || ARRAY[ Lmax+1 :
P_C1 ];
Q_Left := ARRAY[ Lmin : Q_C3 ] || Q_LS || ARRAY[ Lmax+1 :
Q_C1 ];
P_Right := ARRAY[ Lmin : P_C4 ] || P_RS || ARRAY[ Lmax+1 :
P_C2 ];
Q_Right := ARRAY[ Lmin : Q_C4 ] || Q_RS || ARRAY[ Lmax+1 :
Q_C2 ];

P_middle, Q_Middle
:= FOR K IN Kmin+1, Kmax
P_Top, Q_Top := IF World.Wbcu.Reflect THEN
P[ K, Lmax ],
Q[ K, Lmax ]
ELSE World.Wbcu.Press, 0.0
END IF;

P_Bot, Q_Bot := IF World.Wbcd.Reflect THEN

```

IFSC-USP SERVIÇO DE BIBLIOTECA
INFORMAÇÃO

```

LET
cOf := 0.375;
cif := 0.250;

dRK := X[K,L].R - X[K-1,L-1].R + X[K,L-1].R - X[K-1,L].R;
dRL := X[K,L].R - X[K-1,L-1].R - X[K,L-1].R + X[K-1,L].R;
dZK := X[K,L].Z - X[K-1,L-1].Z + X[K,L-1].Z - X[K-1,L].Z;
dZL := X[K,L].Z - X[K-1,L-1].Z - X[K,L-1].Z + X[K-1,L].Z;
dUK := V[K,L].U - V[K-1,L-1].U + V[K,L-1].U - V[K-1,L].U;
dUL := V[K,L].U - V[K-1,L-1].U - V[K,L-1].U + V[K-1,L].U;
dWK := V[K,L].W - V[K-1,L-1].W + V[K,L-1].W - V[K-1,L].W;
dWL := V[K,L].W - V[K-1,L-1].W - V[K,L-1].W + V[K-1,L].W;

w1 := dRK * dWL - dZK * dUL;
w2 := dUK * dZL - dWK * dRL;

w3 := IF w1 < 0.0
THEN
(w1 * w1) / ( dRK*dRK + dZK*dZK )
ELSE
0.0
END IF;

w4 := IF w2 < 0.0
THEN
(w2*w2) / ( (dRL*dRL) + (dZL*dZL) )
ELSE
0.0
END IF;

cs2 := Gamma * P_KL / Rho_KL;
cs := DF_Sqrt(cs2); % cs is the speed of sound in zone K,L

Q_KL := IF (P_KL = 0.0) | ( w3 + w4 = 0.0 )
THEN
0.0
ELSE
cOf * Rho_KL * (w3 + w4) + cif * cs * Rho_KL *
DF_Sqrt(w3 + w4)
END IF;

tao := (AJ_KL*AJ_KL) /
(cs2 * ((dRK*dRK) + (dRL*dRL) + (dZK*dZK) + (dZL*dZL)))
IN
Q_KL, tao
END LET
END FUNCTION % New_Q

FUNCTION Conduct ( X : aaRZpair;
Rho,
CC,
Sigma,
Temp,
Mass : aaReal;
Locs : aaIndex;
dTMPH : REAL;
World : Global
RETURNS aaReal,
aaReal,
real,
real,
aaReal,
aaIndex,
real )

% Output comments:
% Results are New Pressure, New Energy, Internal energy,
% dHN, New Temp, New indices, and dTE.

LET
Kmin, Kmax, Lmin, Lmax
:= World.Kmin, World.Kmax, World.Lmin, World.Lmax;
Tflr := 0.0001;

% Coupling constants in the K direction :

dbb := FOR K IN Kmin, Kmax CROSS L IN Lmin+1, Lmax
Stuff := ( X[K,L-1].R + X[K,L].R )
* ( ( X[K,L].R - X[K,L-1].R ) * ( X[K,L].R - X[K,L-1].R )
+ ( X[K,L].Z - X[K,L-1].Z ) * ( X[K,L].Z - X[K,L-1].Z ) );

bb_KL := IF K = Kmin
THEN
CC[Kmin+1,L] * 0.5 * stuff
ELSEIF K = Kmax
THEN
CC[Kmax,L] * 0.5 * stuff
ELSE
( CC[K+1,L] * CC[K,L] / (CC[K+1,L] +
CC[K,L]) ) * stuff
END IF
RETURNS
ARRAY OF bb_KL
END FOR;

% Coupling constants on the L direction :

cbb := FOR K IN Kmin+1, Kmax CROSS L IN Lmin, Lmax
cb_KL := IF (L = Lmin) | (L = Lmax)
THEN
0.0
ELSE
( CC[K,L] * CC[K,L+1] / (CC[K,L] +
CC[K,L+1]) )
* ( X[K-1,L].R + X[K,L].R )
*
( (X[K,L].R - X[K-1,L].R) * (X[K,L].R - X[K-1,L].R)
+ (X[K,L].Z - X[K-1,L].Z) * (X[K,L].Z - X[K-1,L].Z) )

```

```

END IF
RETURNS
ARRAY OF cb_KL
END FOR;

% Alpha, Beta FORWARD sweep ( L direction )

nTemp := FOR K IN Kmin+1, Kmax
A, B := FOR INITIAL
A_elem := 0.0;
B_elem := 0.0;
L := Lmin + 1
UNTIL L > Lmax
REPEAT
Denom := Sigma[K,OLD L] +
cbb[K,OLD L]
+ cbb[K,OLD L-1] *
(1.0 - OLD A_elem);
A_elem := cbb[K,OLD L] / Denom;
B_elem := ( Sigma[K,OLD L] *
Temp[K,OLD L]
+ cbb[K,OLD L-1] *
OLD B_elem ) / Denom;
L := OLD L + 1
RETURNS
ARRAY OF A_elem
ARRAY OF B_elem
END FOR;

% This is done by creating an array with returns array of, and then
% reversing it to get Tval, rather than the repeated ADDLs done in the
% original code. This has several advantages: It avoids the copying in
% the ADDLs. It is more parallel, and it means that the loop becomes simple,
% and will therefore be optimised.

Trev := FOR INITIAL
T_elem := 0.0;
L := Lmax
UNTIL L < Lmin + 1
REPEAT
T_elem := A[ OLD L -
Lmin + 1] *
OLD T_elem +
B[ OLD L - Lmin + 1];
L := OLD L - 1
RETURNS
ARRAY OF T_elem
END FOR;

% Now reverse the array and give it the correct bounds, Lmin+1..Lmax+1

Tval :=
FOR I IN Lmin+1, Lmax+1
T_EI := Trev[Lmax + 2 - I]

RETURNS ARRAY OF T_EI
END FOR

RETURNS
ARRAY OF Tval
END FOR;

% Alpha, Beta forward sweep ( K direction )
% WARNING: The result array, Temp_T, is transposed from the K,L order

Temp_T := FOR L IN Lmin+1, Lmax
A, B := FOR INITIAL
A_elem := 0.0;
B_elem := 0.0;
K := Kmin + 1
UNTIL K > Kmax
REPEAT
Denom := Sigma[OLD K,L] +
dbb[OLD K,L]
+ dbb[OLD K-1,
L] * (1.0 -
OLD A_elem);
A_elem := dbb[OLD K,L] /
Denom;
B_elem := (Sigma[OLD K,L] *
nTemp[OLD K,L]
+ dbb[OLD K-1,
L] * OLD B_elem) /
Denom;
K := OLD K + 1
RETURNS
ARRAY OF A_elem
ARRAY OF B_elem
END FOR;

% same trick as before to remove the ADDLs.

Trev := FOR INITIAL
T_elem := 0.0;
K := Kmax
UNTIL K < Kmin + 1
REPEAT
T_elem := A[ OLD K - Kmin + 1]
* OLD T_elem + B[ OLD K - Kmin
+ 1];
K := OLD K - 1
RETURNS
ARRAY OF T_elem
END FOR;

% Now reverse the array and give it the correct bounds, Kmin+1..Kmax+1

Trow :=
FOR I IN Kmin+1, Kmax+1
T_EI := Trev[Kmax + 2 - I]

```

```

        RETURNS ARRAY OF T_E1
    END FOR

    RETURNS
    ARRAY OF Trow
    END FOR;

P, Energy, Temp3, yE, Eint, New_Indices
:= FOR K IN Kmin+1, Kmax CROSS L IN Lmin+1, Lmax
E_KL, N1 := Lookup_Energy(Temp_T[L,K],
Rho[K,L], Locs [K,L], World);
Temp3_KL := MAX(Temp_T[L,K], Tflr);
TempR := ABS( (Temp3_KL - Temp[K,L])
/ Temp[K,L] );
P_KL, N2 := Lookup_Pressure(Temp3_KL,
Rho[K,L], N1, World);
Eint_KL := Mass[K,L] * E_KL
    RETURNS
    ARRAY OF P_KL
    ARRAY OF E_KL
    ARRAY OF Temp3_KL
    VALUE OF GREATEST TempR
    VALUE OF SUM Eint_KL
    ARRAY OF N2
    END FOR;

dTE := 0.1 * dTNP / yE;
Eloss := dTNP * Boundary_Heat_Flow(Temp3, cbb, dbb, World)
    IN
P, Energy, Eint, Eloss, Temp3, New_Indices, dTE
    END LET
END FUNCTION % Conduct

FUNCTION Boundary_Heat_Flow ( Temp,
cbb,
dbb : aareal;
World : Global
RETURNS REAL )

% There are some problems with this FUNCTION. Ackerman notes in his
% VAL version that the original Fortran program went out of bounds
% on the Temp ARRAY. Apparently, the "BandT" equation originally was
% something like "Temp[K,Lmin] - Temp[K,Lmin+1]" in parens. The
% problems I see are two-fold, one we are no longer taking a difference
% of temperatures so I doubt we are computing a boundary flow; second,
% the cbb array has zeroes stored in its Lmin and Lmax rows!
%
% Chris ---- HELP !!!!!
%

    LET
    Kmin, Kmax, Lmin, Lmax
    := World.Kmin, World.Kmax, World.Lmin, World.Lmax;

    UpDn_flow := FOR K IN Kmin+1, Kmax
    BandT := cbb[K,Lmin] * (- Temp[K,Lmin+1] )
    + cbb[K,Lmax] * (- Temp[K,Lmax ] )
    RETURNS
    VALUE OF SUM BandT
    END FOR;

    Side_flow := FOR L IN Lmin+1, Lmax
    LandR := dbb[Kmin,L] * (- Temp[Kmin+1,L] )
    + dbb[Kmax,L] * (- Temp[Kmax,L ] )
    RETURNS
    VALUE OF SUM LandR
    END FOR
    IN
    UpDn_flow + Side_flow
    END LET
END FUNCTION % Boundary_Heat_Flow

FUNCTION RZ_Calc ( X : aaRZpair;
V : aaWpair;
P,
Q,
Rho_J : aareal;
dTn,
dTNP : REAL;
World : Global
RETURNS aaRZpair,
aaWpair,
aareal )

    LET
    Kmin, Kmax, Lmin, Lmax
    := World.Kmin, World.Kmax, World.Lmin, World.Lmax;
    Vcut := 1.0e-10
    IN
    FOR K IN Kmin, Kmax CROSS L IN Lmin, Lmax
    AUW := Rho_J[K,L] + Rho_J[K+1,L] + Rho_J[K,L+1] +
    Rho_J[K+1,L+1];
    PQKL := P[K,L] + Q[K,L];
    PQK1L := P[K+1,L] + Q[K+1,L];
    PQKL1 := P[K,L+1] + Q[K,L+1];
    PQK1L1 := P[K+1,L+1] + Q[K+1,L+1];
    Y_prime := 2.0 * dTn / AUW;

    U_KL := V[K,L].U - Y_prime*( PQKL * (X[K,L-1].Z -
X[K-1,L].Z) +
PQK1L * (X[K+1,L].Z - X[K,L-1].Z) +
PQKL1 * (X[K-1,L].Z - X[K,L+1].Z) +
PQK1L1 * (X[K,L+1].Z - X[K+1,L].Z) );

    new_u := IF ABS(U_KL) < Vcut THEN 0.0 ELSE U_KL END IF;

    W_KL := V[K,L].W - Y_prime*( PQKL * (X[K,L-1].R -
X[K-1,L].R) +
PQK1L * (X[K+1,L].R - X[K,L-1].R) +

```



```

PQKL1 = (X[K-1,L].R - X[K,L+1].R) +
PQKL1 = (X[K,L+1].R - X[K+1,L].R);

new_u := IF ABS(W_KL) < Vcut THEN 0.0 ELSE W_KL END IF;

new_x := RECORD [ R : X[K,L].R + dTNPW*new_u;
Z : X[K,L].Z + dTNPW*new_u ];
new_v := RECORD [ U : new_u;
W : new_u ];

UV2 := (new_u*new_u) + (new_w*new_w)
RETURNS
ARRAY OF new_x
ARRAY OF new_v
ARRAY OF UV2
END FOR
END LET
END FUNCTION % RZ_Calc

FUNCTION Table_Lookup ( Table : areal;
Elem : REAL;
Start : INTEGER
RETURNS INTEGER )

IF Table[Start] <= Elem
THEN
FOR INITIAL
i := Start
UNTIL (Table[i + 1] > Elem)
REPEAT
i := OLD i + 1
RETURNS
VALUE OF i
END FOR
ELSE
FOR INITIAL
i := Start
UNTIL (Table[i] < Elem) { (i = 1)
REPEAT
i := OLD i - 1
RETURNS
VALUE OF i
END FOR
END IF
END FUNCTION % Table_Lookup

% The rest of the code is a function Main to output results, and three
% functions TwoToOne, RZ_TwoToOne and UW_TwoToOne which fold 2-d array types
% areal, aaRZpair and aaUWpair to areal, aRZpair and aUWpair respectively.

function TwoToOne (s: areal returns areal)

for E1 in S
Row := E1
returns value of catenate Row
end for

end function % TwoToOne

% Extract the real values from the record outputs, because with the pointer
% representation of records, it is not possible to output an array of
% records directly.

function RZ_TwoToOne(s: aaRZpair returns aRZPair)

for e1 in s
Row := E1
returns value of catenate row
end for

end function % RZ_TwoToOne

function UW_TwoToOne(s: aaUWpair returns aUWPair)

for e1 in s
Row := E1
returns value of catenate row
end for

end function % UW_TwoToOne

function Main ( NCycles, ArraySize : integer
returns integer, real, real, real, real, real, real,
real, real, real, real, real, real, real,
aRZpair, aUWpair,
areal, areal, areal, areal, areal, areal )

let

Kmin:= 2; Kmax:= ArraySize; Lmin:= 2; Lmax:= ArraySize;
s := Simple (NCycles, Kmin, Kmax, Lmin, Lmax)

in

s.Cycles_Out,
s.Time_Out,
s.dT_Out,
s.dTC_Out,
s.dTE_Out,
s.ETot_Out,
s.IE_Out,
s.KE_Out,
s.HN_Out,
s.WW_Out,

```

```

s.CE_Out,
s.dCE_Out,
s.dTSPH_Out,
s.dTH_Out,
RZ_TwoToOne(s.RZ_Out), % Kmin..Kmax X Lmin..Lmax
UV_TwoToOne(s.UV_Out), % Kmin..Kmax X Lmin..Lmax
TwoToOne(s.Rho_Out), % Kmin+1..Kmax X Lmin+1..Lmax
TwoToOne(s.E_Out), % Kmin+1..Kmax X Lmin+1..Lmax
TwoToOne(s.P_Out), % Kmin+1..Kmax X Lmin+1..Lmax
TwoToOne(s.Q_Out), % Kmin+1..Kmax X Lmin+1..Lmax
TwoToOne(s.AJ_Out), % Kmin+1..Kmax X Lmin+1..Lmax
TwoToOne(s.Theta_Out) % Kmin+1..Kmax X Lmin+1..Lmax

end let

end function % Main

```