Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Rubem Euzébio Ferreira

Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC

Rio de Janeiro
2009

Rubem Euzébio Ferreira

Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Orientadora: Prof^a. Dr^a. Luiza de Macedo Mourelle
Co-orientadora: Prof^a. Dr^a. Nadia Nedjah

Rio de Janeiro
2009

CATALOGAÇÃO NA FONTE
UERJ/REDE SIRIUS/CTC/B

F383 Ferreira, Rubem Euzébio.
Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC/Rubem Euzébio Ferreira. – 2009.
195 f.

Orientadora: Luiza de Macedo Mourelle.
Co-orientadora: Nadia Nedjah.

Dissertação (mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.
Bibliografia: f. 142 – 147.

1. Algoritmos genéticos. 2. Sistemas embutidos. 3. Redes de computador. I. Mourelle, Luiza de Macedo. II. Nedjah, Nadia. III. Universidade do Estado do Rio de Janeiro. IV. Título.

CDU 658.5

Autorizo, apenas para fins acadêmicos e científicos, a reprodução total ou parcial desta dissertação.

Assinatura Data

Rubem Euzébio Ferreira

Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Aprovado em 7 de Agosto de 2009

Comissão Examinadora:

Prof^a. Dr^a. Luiza de Macedo Mourelle (Orientadora)
Faculdade de Engenharia, UERJ

Prof^a. Dr^a. Nadia Nedjah (Co-orientadora)
Faculdade de Engenharia, UERJ

Prof. Dr. Felipe Maia Galvão França
Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ

Prof. Dr. Luiz Satoru Ochi
Instituto de Computação, UFF

Rio de Janeiro
2009

DEDICATÓRIA

Cria em mim Ó Deus um coração puro e renova em mim
um espírito inabalável. (Salmos 51:10)

AGRADECIMENTOS

Ao Deus Único, Vivo e Eterno que me permitiu com sua infinita graça e cuidado, chegar até aqui.

À minha amada esposa Mariana e aos meus filhos Rafael e Rebeca pelo amor, paciência e compreensão.

Aos meus pais Altevi e Girlene pela formação que me deram.

À Universidade do Estado do Rio de Janeiro por me receber novamente como aluno.

Às minhas orientadoras Prof^a. Luiza de Macedo Mourelle e Prof^a. Nadia Nedjah pelos ensinamentos, sugestões, correções e paciência comigo.

Aos funcionários e professores do Programa de Pós-graduação em engenharia eletrônica.

Aos amigos da Dinfo, Marcelo, Suely, Paulo, Ana Beatriz e Jovino que me cobriram durante minha ausência e me incentivaram.

Aos amigos do mestrado, Marcos Paulo e Marcus Vinícius.

Aos amigos da graduação, Luneque, Luis, Fernanda e Gabriel.

E a todos os outros amigos que direta ou indiretamente contribuíram para a conclusão dessa dissertação.

Aos colegas do mestrado, Joaquim, Daniel, Renato, André e Rodrigo pela força.

RESUMO

Ferreira, Rubem Euzébio. *Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC*. 2009. 195f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2009.

Essa dissertação apresenta a implementação de um algoritmo genético paralelo utilizando o modelo de granularidade grossa, também conhecido como modelo das ilhas, para sistemas embutidos multiprocessados. Os sistemas embutidos multiprocessados estão tornando-se cada vez mais complexos, pressionados pela demanda por maior poder computacional requerido pelas aplicações, principalmente de multimídia, Internet e comunicações sem fio, que são executadas nesses sistemas. Algumas das referidas aplicações estão começando a utilizar algoritmos genéticos, que podem ser beneficiados pelas vantagens proporcionadas pelo processamento paralelo disponível em sistemas embutidos multiprocessados. No algoritmo genético paralelo do modelo das ilhas, cada processador do sistema embutido é responsável pela evolução de uma população de forma independente dos demais. A fim de acelerar o processo evolutivo, o operador de migração é executado em intervalos definidos para realizar a migração dos melhores indivíduos entre as ilhas. Diferentes topologias lógicas, tais como anel, vizinhança e *broadcast*, são analisadas na fase de migração de indivíduos. Resultados experimentais são gerados para a otimização de três funções encontradas na literatura.

Palavras-chave: redes intrachip, algoritmos genéticos paralelos, sistemas embutidos.

ABSTRACT

This dissertation presents an implementation of a parallel genetic algorithm using the coarse grained model, also known as the islands model, targeted to MPSoCs systems. MPSoC systems are becoming more and more complex, due to the greater computational power demanded by applications, mainly those that deal with multimedia, Internet and wireless communications, which are executed within these systems. Some of these applications are starting to use genetic algorithms, that can benefit from the parallel processing offered by MPSoC. In the island model for parallel genetic algorithm, each processor is responsible for evolving the corresponding population independently from the others. Aiming at accelerating the evolutionary process, the migration operator is executed periodically in order to migrate the best individuals among islands. Different logic topologies, such as ring, neighborhood and broadcast, are analyzed during the migration step. Experimental results are generated for the optimization of three functions found in the literature.

Keywords: network-on-chip, parallel genetic algorithms, embedded systems.

LISTA DE FIGURAS

1 Estrutura interna de um SoC 21
2 Estrutura interna de um MPSoC 22
3 Chave interconectada com um recurso 23
4 Camadas do modelo OSI 25
5 Pontos de acesso a serviços e entidades 25
6 Nó de rede direta 26
7 Topologias malha, toroide, hipercubo 27
8 Topologias *crossbar* e multiestágio 27
9 Chave Hermes 31
10 Os nove tipos de chaves possíveis 32
11 Chaves RASoC 32
12 Diagrama em blocos do processador ARM 1136 35
13 Diagrama em blocos do processador PowerPC 440 36
14 Diagrama em blocos do processador MIPS32 24Kf 37

15 Rede intrachip Hermes 40
16 Sinais de interfaceamento externo da chave 41
17 Ligação entre as portas leste e oeste de duas chaves vizinhas 41
18 Estrutura interna da lógica de controle 42
19 Tabela de roteamento 44
20 Máquina de estados da lógica de controle 45
21 Estrutura interna da fila 46
22 Fila com duas posições 47
23 Fila com quatro posições 47
24 Máquina de estados de remoção de *flits* da fila 48
25 Processador Plasma 50
26 Espaço de endereçamento do processador Plasma 50
27 Geração do endereço físico 51
28 Registador de estado das interrupções 52
29 Sinais de interfaceamento da interface de rede 53
30 Pacote não segmentado 57
31 Pacote segmentado em flits de 16 bits 58
32 Máquina de estados para o envio de pacotes para a rede intrachip 58
33 Máquina de estados para o recebimento de pacotes da rede intrachip 59
34 Sinais de interfaceamento do controlador de DMA 59
35 Máquina de estados do controlador de DMA 61

36 Estrutura do repositório de tarefas 66
37 Níveis do microkernel do processador mestre 66
38 Estrutura *TaskLocation* 67
39 Estrutura *TaskPackage* 67

40 Estrutura *processors* 67
41 Estrutura *free_pages* 68
42 Microkernel do processador escravo 73
43 Estrutura *Message* 74
44 Estrutura do TCB 74
45 Estrutura *RequestTask* 75
46 Estrutura *RequestMessage* 75
47 Configuração de memória 76
48 Comunicação entre tarefas residentes no mesmo processador 83
49 Comunicação entre tarefas residentes em processadores diferentes 84

50 Exemplo de cromossomo com representação binária 89
51 Exemplo de cromossomo com representação real 89
52 Método de seleção pela roleta 93
53 Cruzamento de um ponto 94
54 Cruzamento de dois pontos 95
55 Cruzamento uniforme 95
56 Mutação binária 97
57 Particionamento de uma tarefa em três subtarefas, com subsequente alocação a três processadores 100
58 Paralelismo de dados 101
59 Paralelismo funcional 102
60 Paralelismo de objetos 102
61 Modelo da paralelização global 104
62 Modelo da granularidade fina 104
63 Modelo da granularidade grossa 105
64 Topologia de migração *ring* 106
65 Topologia de migração *neighborhood* 106
66 Topologia de migração *broadcast* 107

67 Rede de Petri ilustrando a operação do AGPE 109
68 Estrutura do indivíduo 111
69 Estrutura do cromossomo com duas variáveis 112
70 Curvas das funções utilizadas nos processos de otimização 117
71 Alocação das tarefas do AGPE na plataforma HMPS 118
72 Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em anel 121
73 Impacto da taxa e intervalo de migração no *speedup* e eficiência considerando a topologia de migração em anel 124
74 Migração de indivíduo do processador 10 para o 20 utilizando a comunicação em vizinhança 125
75 Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em vizinhança 126
76 Impacto da taxa e intervalo de migração no *speedup* e eficiência, considerando a topologia de migração vizinhança 130
77 Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em broadcast 131
78 Migração de indivíduo do processador 10 para o 21 utilizando a comunicação em broadcast 132

79 Migração de indivíduo do processador 10 para o 20 utilizando a comunicação em broadcast 133
80 Impacto da taxa e intervalo de migração no *speedup* e eficiência considerando a topologia de migração em *broadcast* 134
81 Impacto do número de processadores 136
82 Impacto da escolha da topologia de migração na otimização de $f_1(x)$ 137
83 Impacto da escolha da topologia de migração na otimização de $f_2(x, y)$ 137
84 Impacto da escolha da topologia de migração na otimização de $f_3(x, y)$ 137

LISTA DE TABELAS

1 Sinais de interfaceamento da chave Hermes 40
2 Registradores mapeados em memória do controlador de interrupção 52
3 Máscaras das interrupções 52
4 Sinais de interfaceamento da interface de rede 54
5 Registradores mapeados em memória para a comunicação entre *drivers* e interface de rede 55
6 Descrição dos serviços que um pacote carrega 55
7 Sinais de interfaceamento do controlador de DMA 60
8 Registradores mapeados em memória para a comunicação entre o Microkernel e o controlador de DMA 60
9 Configuração do tamanho do *flit* da chave, da fila da chave, do *flit* da interface de rede, da fila da interface de rede, da rede intrachip e do tamanho de página . 64

10 Serviços das chamadas de sistema 77

11 Exemplo de seleção pelo método da roleta 93

12 Parâmetros do AGPE 110
13 Configuração dos parâmetros da Plataforma HMPS e do AGPE 119
14 Resultados de otimização da função $f_1(x)$ para a comunicação em anel 122
15 Resultados de otimização da função $f_2(x, y)$ para a comunicação em anel 122
16 Resultados de otimização da função $f_3(x, y)$ para a comunicação em anel 123
17 Resultados de otimização da função $f_1(x)$ para a comunicação em vizinhança . . 127
18 Resultados de otimização da função $f_2(x, y)$ para a comunicação com a vizinhança 127
19 Resultados de otimização da função $f_3(x, y)$ para a comunicação com a vizinhança 128
20 Resultados de otimização da função $f_1(x)$ para a comunicação em *broadcast* . . . 129
21 Resultados de otimização da função $f_2(x, y)$ para a comunicação em *broadcast* . 129
22 Resultados de otimização da função $f_3(x, y)$ para a comunicação em *broadcast* . 129
23 Melhores resultados obtidos na otimização da função $f_1(x)$ 134
24 Melhores resultados obtidos da na otimização da função $f_2(x, y)$ 135
25 Melhores resultados obtidos na otimização da função $f_3(x, y)$ 135

26 Arquivos da Plataforma HMPS 152
27 Arquivos utilizados para a compilação da tarefa 153
28 Primitivas da plataforma HMPS 154

LISTA DE ALGORITMOS

1 Função $DRV\_Handler()$ do processador mestre 69
2 Função $TasksAllocation()$ 71
3 Função $WritePipe(msg, t)$ 72
4 Função $Syscall(s, msg, t)$ 77
5 Função $DRV\_Handler()$ dos processadores escravos 79
6 Função $DMA\_Handler()$ 81
7 Função $Scheduler$ 82
8 Fluxograma de um algoritmo genético típico 88
9 Algoritmo genético paralelo para uma ilha 112
10 Função de migração para a comunicação em anel 113
11 Função de migração para a comunicação com a vizinhança 114
12 Função de migração para a comunicação em *broadcast* 114

SUMÁRIO

INTRODUÇÃO 16

1 SISTEMAS EMBUTIDOS MULTIPROCESSADOS **20**
1.1 Sistemas Embutidos 20
1.2 Rede Intrachip 21
1.2.1 O modelo de referência OSI 23
1.2.2 Topologias 25
1.2.3 Métodos de chaveamento 26
1.2.4 Algoritmos de roteamento empregados em redes intrachip 28
1.2.5 Tráfego de pacotes em redes intrachip 30
1.2.6 Arquiteturas de redes intrachip 30
1.2.6.1 Hermes 31
1.2.6.2 SoCIN 31
1.2.6.3 Nostrum 33
1.2.6.4 SoCBUS 33
1.2.6.5 Proteo 33
1.3 Processadores para Sistemas Embutidos 34
1.3.1 ARM 1136JF-S 34
1.3.2 IBM PowerPC 440 34
1.3.3 MIPS32 24Kf 35
1.4 Sistemas Operacionais Embutidos 36
1.4.1 Embedded Linux 37
1.4.2 Windows CE 37
1.4.3 QNX RTOS 37
1.4.4 eCos 38
1.4.5 EPOS 38
1.5 Considerações Finais 38

2 PLATAFORMA HMPS DE REDE INTRACHIP **39**
2.1 Rede Intrachip Hermes 39
2.1.1 A chave Hermes 40
2.1.1.1 Lógica de controle 42
2.1.1.2 Fila 45
2.1.2 Conexões entre as chaves e os recursos 48
2.1.3 Interconexões entre as chaves da rede intrachip 48
2.2 O Processador Plasma 49
2.2.1 Paginador de memória 49
2.2.2 Controlador de interrupção 51
2.2.3 Interface de rede 53
2.2.3.1 Envio de pacotes para a rede intrachip 57
2.2.3.2 Recepção de pacotes da rede intrachip 58

2.2.4 Controlador de DMA . . . 59
2.3 Melhorias no Modelo da Plataforma . . . 61
2.3.1 Parametrização do tamanho do flit da chave . . . 62
2.3.2 Parametrização do tamanho da rede intrachip . . . 62
2.3.3 Parametrização do tamanho de página de memória . . . 63
2.3.4 Configuração do sistema embutido multiprocessado HMPS . . . 63
2.4 Considerações Finais . . . 64

3 O MICROKERNEL . . . **65**
3.1 O Microkernel do Processador Mestre . . . 65
3.1.1 Repositório de tarefas . . . 65
3.1.2 Estrutura do microkernel do processador mestre . . . 66
3.1.3 Estruturas de dados do processador mestre . . . 66
3.1.4 Inicialização do microkernel do processador mestre . . . 67
3.1.5 Tratamento de interrupções . . . 68
3.1.6 Drivers de comunicação . . . 68
3.1.7 Alocação estática . . . 70
3.1.8 Alocação dinâmica . . . 71
3.2 O Microkernel dos Processadores Escravos . . . 72
3.2.1 Estrutura do microkernel dos processadores escravos . . . 73
3.2.2 Estruturas de dados dos processadores escravos . . . 74
3.2.3 Inicialização do microkernel dos processadores escravos . . . 75
3.2.4 Tratamento de interrupções . . . 76
3.2.5 Chamadas de sistema . . . 76
3.2.6 Drivers de comunicação . . . 78
3.2.7 Escalonamento de tarefas . . . 81
3.2.8 Comunicação entre tarefas . . . 82
3.3 Melhorias Realizadas no Microkernel da Plataforma . . . 84
3.3.1 Parametrização do tamanho do flit . . . 84
3.3.2 Parametrização do tamanho de página de memória . . . 85
3.4 Considerações Finais . . . 85

4 ALGORITMOS GENÉTICOS . . . **86**
4.1 Conceitos Algoritmos Genéticos . . . 86
4.1.1 Representação dos parâmetros . . . 88
4.1.2 Inicialização da população . . . 90
4.1.3 Avaliação . . . 90
4.1.3.1 Ordenamento linear . . . 91
4.1.3.2 Ordenamento exponencial . . . 91
4.1.4 Seleção . . . 91
4.1.4.1 Roleta viciada . . . 92
4.1.4.2 Torneio . . . 93
4.1.4.3 Amostragem estocástica universal . . . 93
4.1.4.4 Elitismo . . . 93
4.1.5 Operadores genéticos . . . 94
4.1.5.1 Cruzamento . . . 94
4.1.5.2 Mutação . . . 97
4.1.6 Parâmetros utilizados pelos algoritmos genéticos . . . 98
4.2 Algoritmos Genéticos Paralelos . . . 99
4.2.1 Tipos de paralelismo . . . 100

4.2.1.1 Paralelismo de dados . . . 101
4.2.1.2 Paralelismo funcional . . . 101
4.2.1.3 Paralelismo de objetos . . . 101
4.2.2 Plataformas para processamento paralelo . . . 101
4.2.2.1 Modelo de memória compartilhada . . . 102
4.2.2.2 Modelo de troca de mensagens . . . 103
4.2.2.3 Modelo de threads . . . 103
4.2.3 Modelos de algoritmos genéticos paralelos . . . 103
4.2.3.1 Modelo de paralelização global . . . 104
4.2.3.2 Granularidade fina . . . 104
4.2.3.3 Granularidade grossa . . . 105
4.3 Considerações Finais . . . 107

5 ALGORITMO GENÉTICO PARALELO PARA SISTEMA EMBUTIDO MULTIPROCESSADO . . . 108
5.1 Algoritmo Genético Paralelo Embutido . . . 108
5.2 Algoritmo Genético de uma Ilha . . . 110
5.2.1 Codificação dos indivíduos . . . 111
5.2.2 Migração . . . 112
5.3 Resultados Experimentais . . . 115
5.3.1 Ambiente de desenvolvimento . . . 115
5.3.2 Configurações de simulação . . . 116
5.3.2.1 Funções objetivo . . . 116
5.3.2.2 Configuração da plataforma e do AGPE . . . 117
5.3.2.3 Métricas de desempenho do AGPE . . . 118
5.3.3 Resultados de simulação . . . 119
5.3.3.1 Comunicação em anel . . . 119
5.3.3.2 Comunicação com vizinhança . . . 124
5.3.3.3 Comunicação em broadcast . . . 128
5.3.4 Discussão dos resultados . . . 130
5.4 Considerações Finais . . . 138

6 CONCLUSÕES E TRABALHOS FUTUROS . . . 139
6.1 Conclusões . . . 139
6.2 Trabalhos Futuros . . . 140

REFERÊNCIAS . . . 142

APÊNDICE A – Configuração da Plataforma . . . 148

APÊNDICE B – Instruções de Uso da Plataforma . . . 151

APÊNDICE C – Modelo VHDL da Chave . . . 167

APÊNDICE D – Modelo VHDL da Rede Intrachip . . . 186

APÊNDICE E – Modelo VHDL do Sistema HMPS . . . 190

APÊNDICE F – Passos para Construção do Compilador Cruzado . . . 194

INTRODUÇÃO

A DEMANDA crescente de dispositivos eletrônicos que requerem cada vez mais poder de processamento, baixo consumo, redução de espaço, redução de custo e redução de tempo de projeto, tem levado frequentemente ao projeto e fabricação de um dispositivo inteiro em um único *chip*. Esses sistemas são chamados de sistemas embutidos ou SoC (*System-on-Chip*). Um número cada vez maior de produtos, como câmeras e filmadoras digitais, telefones celulares, vídeo-games e computadores portáteis, são construídos em um único *chip* (NILSSON, 2002). Esses dispositivos vem se tornando cada vez mais complexos para atender à demanda de aplicações de multimídia, Internet e comunicações sem fio, requerendo um poder computacional maior e apoiados pelos benefícios que os avanços na tecnologia de integração tem proporcionado. Por outro lado, o aumento da complexidade desses dispositivos também tende a comprometer o tempo mínimo desejado de desenvolvimento do projeto e a ocorrência de erros torna-se inaceitável. Tem sido um grande desafio atender a todos esses requisitos.

Os sistemas embutidos podem ser constituídos de vários módulos independentes que operam em paralelo e trocam informações, utilizando um barramento comum de comunicação. Para atender às pressões de mercado, reduzir o custo de projeto e acelerar o desenvolvimento dos sistemas embutidos, tem-se utilizado cada vez mais de módulos reutilizáveis com funções específicas, os chamados blocos IP (*Intelectual Property*). Dentre os blocos IP disponíveis atualmente encontram-se processadores, memórias, barramentos, dispositivos de entrada e saída, e outras funções digitais.

O aumento da complexidade dos sistemas embutidos, pressionado pelo aumento de poder computacional requerido pelas aplicações de aplicações de multimídia, Internet e comunicações sem fio mencionadas anteriormente, tem levado ao projeto e construção de dispositivos multiprocessados. Quando um sistema embutido possui mais de um processador, é chamado de Sistema Embutido Multiprocessado ou MPSoC (*Multi-Processor System-on-Chip*). Algumas aplicações estão começando a utilizar algoritmos genéticos (RUIZ; ANTONIO, 2003)(ZHANG; LEUNG, 1999) e podem ser beneficiadas pelas vantagens proporcionadas pelo processamento paralelo de sistemas embutidos multiprocessados.

Como mencionado anteriormente, a comunicação entre os blocos IP em sistema embutido é realizada através de um barramento comum. Entretanto em sistemas embutidos multiprocessados, esse tipo de interconexão compromete o desempenho do sistema, visto que vários processadores e outros blocos IP compartilham o mesmo barramento. Sendo assim, essa comunicação é passa a ser implementada através de uma rede intrachip.

A rede intrachip, utilizada em um sistema embutido multiprocessado, é constituída de vários blocos IP interconectados por meio de chaves (ZEFERINO, 2003) (MELLO, 2003). Esses dispositivos tem como função interconectar os elementos de uma rede local, permitir a retransmissão de mensagens de qualquer bloco IP para outro e tomar decisões relacionadas ao caminho que essas mensagens devem seguir, sendo o principal componente da rede. As redes de comunicação empregadas em sistemas embutidos não são obrigadas a empregar todos os conceitos e técnicas de redes de computadores tradicionais. Em relação ao modelo de referência OSI (*Open System Interconnection*) (ZIMMERMANN, 1980), verifica-se que o modo como cada plataforma de sistema embutido multiprocessado implementa suas camadas é diferente. Além disso, os projetistas de cada arquitetura têm ideias diferentes sobre que camadas devem ser implementadas. De um modo geral, as camadas física, enlace e rede são implementadas pelas chaves, as camadas de transporte, sessão e apresentação pelo sistema operacional, e a camada de aplicação pelo *software* que iremos executar.

Sistemas operacionais (TANENBAUM, 1997) são programas responsáveis por intermediar os recursos existentes em um computador, tais como processadores, memórias, dispositivos de E/S, além de fornecer a base para o desenvolvimento de programas de aplicação. As aplicações são compostas por tarefas, sendo uma tarefa um conjunto de instruções e dados com informações necessárias à sua correta execução em um processador. Além disso, os sistemas operacionais podem ser vistos como uma camada de *software* que provê um ambiente com uma interface mais simples e conveniente para o usuário.

Como um subgrupo de sistemas operacionais, encontram-se os sistemas operacionais embutidos (WOSZEZENKI, 2007). Estes vêm tornando-se bastante populares, visto que implementam apenas as funcionalidades necessárias à aplicação que será executada. O tamanho do sistema operacional embutido tende a ser menor do que o de um sistema operacional, reduzindo o *kernel* a um *microkernel*, que é o núcleo do sistema operacional. Essa redução é desejável para aplicações embutidas, como, por exemplo, aplicações para telefones celulares. O tamanho do sistema operacional deve ser levado em conta, uma vez que a quantidade de memória disponível em um sistema embutido multiprocessado é restrita.

Os algoritmos genéticos (LACERDA; CARVALHO, 1999) são métodos de otimização e busca inspirados nos mecanismos de evolução de populações de seres vivos. Otimização é a busca da melhor solução para um dado problema. Consiste em tentar várias soluções e utilizar a informação obtida nesse processo de forma a encontrar soluções cada vez melhores. Algoritmos genéticos são implementados como uma simulação de computador em que uma população, de representações abstratas de solução, é selecionada em busca de soluções melhores. A evolução geralmente se inicia a partir de um conjunto de soluções criado aleatoriamente e é realizada através de gerações. A cada geração, a adaptação de cada solução na população é avaliada, alguns indivíduos são selecionados para a próxima geração, e recombinados ou alterados para formar uma nova população. A nova população então é utilizada como entrada para a próxima iteração do algoritmo.

Os algoritmos genéticos se constituem em uma técnica muito robusta para resolver computacionalmente problemas complexos. No entanto, os algoritmos genéticos necessitam de um poder computacional muito grande quando as dimensões do problema aumentam. Portanto, versões distribuídas de algoritmos genéticos se tornam atrativas. Dos métodos de algoritmos genéticos paralelos existentes, foi selecionado o método da granularidade grossa (CANTU-PAZ, 1995) por ser mais adequado para execução em sistemas multiprocessados (LIN et al., 1995).

O objetivo dessa dissertação é desenvolver um algoritmo genético paralelo para ser executado em um sistema embutido multiprocessado, levando em conta os recursos limitados disponíveis por esse tipo plataforma, em comparação com um sistema de computação paralelo convencional, como o *cluster* Beowulf (ALLAN; ANDREWS; GUEST, 2009). Para tal, foram feitas modificações em um sistema embutido multiprocessado HMPS (*Hermes MultiProcessor System-on-Chip*), de domínio público, para que o mesmo pudesse executar o algoritmo genético paralelo. Para o desenvolvimento do algoritmo genético paralelo foi necessário um compilador cruzado, que consiste de um compilador capaz de criar código executável para uma plataforma diferente daquela onde o compilador é executado. Os compiladores cruzados são utilizados para gerar código executável para sistemas embutidos ou múltiplas plataformas. Normalmente, essas plataformas possuem recursos limitados de memória que não permitem abrigar seus próprios compiladores. O objetivo fundamental da utilização de compiladores cruzados é separar o ambiente de desenvolvimento do *software*, nno caso o computador, da plataforma onde o mesmo será executado, no caso o sistema embutido multiprocessado. Essa dissertação está estruturada em seis capítulos e sete apêndices, cujos conteúdos são descritos a seguir.

O Capítulo 1 apresenta o conceito de sistemas embutidos multiprocessados e os respec-

tivos componentes. O conceito de rede intrachip, suas topologias, métodos de chaveamento, algoritmos de roteamento, tráfego de pacotes, processadores utilizados nos sistemas embutidos e os sistemas operacionais embutidos também são apresentados.

O Capítulo 2 descreve a plataforma HMPS utilizada nesse trabalho, constituída da rede intrachip Hermes e do processador Plasma. As características de cada um são apresentadas. As modificações realizadas na rede intrachip e no processador, para permitir o desenvolvimento do algoritmo genético paralelo, são também introduzidas.

O Capítulo 3 apresenta a infra-estrutura de *software* da plataforma. Essa infra-estrutura é composta pelo *micokernel* que executa em cada processador do sistema embutido multiprocessado. São descritas as estruturas de dados utilizadas, a inicialização do sistema e os serviços disponíveis: tratamento de interrupções, escalonamento, comunicação entre tarefas, chamadas de sistema e *drivers* de comunicação. Também são descritas as modificações realizadas no *microkernel*, necessárias para a implementação do algoritmo genético paralelo.

O Capítulo 4 apresenta os principais conceitos relacionados com algoritmos genéticos. A inicialização, seleção, avaliação e os operadores genéticos são descritos com detalhes. Por último, são apresentados os modelos de algoritmos genéticos paralelos desenvolvidos.

O Capítulo 5 apresenta a implementação do Algoritmo Genético Paralelo Embutido (AGPE) na plataforma MPSoC. Os resultados obtidos, através de simulações, são introduzidos, como forma de validar o seu funcionamento.

O Capítulo 6 consiste de uma síntese do trabalho desenvolvido, apresentando algumas conclusões e propostas para trabalhos futuros.

O Apêndice A apresenta os parâmetros de configuração da plataforma. O Apêndice B contém as instruções para utilização da plataforma. Os Apêndices C, D e E contêm os modelos da chave, da rede intrachip e do sistema HMPS, respectivamente, especificados na linguagem de descrição de *hardware* VHDL. O Apêndice F contém os passos utilizados para gerar o compilador cruzado. O Apêndice G contém o código das funções do AGPE.

Capítulo 1

SISTEMAS EMBUTIDOS MULTIPROCESSADOS

NESSE capítulo, são apresentados os principais conceitos relacionados aos sistemas embutidos baseados em multiprocessadores. Na Seção 1.1 é apresentada uma visão geral de sistemas embutidos multiprocessados. Na Seção 1.2 são apresentados os conceitos de redes intrachip. Na Seção 1.3 são apresentados os processadores utilizados nos sistemas embutidos e na Seção 1.4 são apresentados os sistemas operacionais executados nesses ambientes.

1.1 Sistemas Embutidos

Sistema Embutido ou *System-on-Chip* (SoC), mostrado na Figura 1, é um sistema implementado em um único encapsulamento, dedicado para um propósito específico. Os sistemas embutidos são constituídos de microprocessadores, memórias, dispositivos de entrada-e-saída, barramentos e outras funções digitais. Muitos dos dispositivos eletrônicos modernos, como máquinas fotográficas digitais, filmadoras e outros, são, na verdade, sistemas embutidos.

Para reduzir o tempo consumido para o desenvolvimento e os custos do projeto de um sistema embutido, é importante que os componentes integrados nesse sistema sejam reutilizáveis. Desta forma, as metodologias de projeto devem ser baseadas na reutilização de componentes pré-projetados. Esses componentes reutilizáveis operam em paralelo, trocam informações utilizando um barramento comum e são denominados blocos de propriedade intelectual (*Intelectual Property Blocks - IP*), podendo ser desenvolvidos pela empresa responsável pelo projeto do sistema embutido ou adquiridos a partir de terceiros.

A demanda por dispositivos com cada vez mais capacidade de processamento vem aumentando, pressionada pelas aplicações atuais de multimídia, Internet e comunicação sem fio. Devido a este fato, os sistemas embutidos tornaram-se mais complexos e o paralelismo veio

como uma solução para melhorar o desempenho desses sistemas. Uma forma de paralelismo é o multiprocessamento.

O multiprocessamento pode ser caracterizado pela existência de vários processadores independentes em um sistema de computação. Quando um sistema embutido possui mais de um processador, é chamado de Sistema Embutido Multiprocessado (*Multi-Processor System on Chip* - MPSoC), mostrado na Figura 2. Enquanto nos sistemas embutidos a comunicação entre os blocos de IP é basicamente realizada através de um barramento comum, em um Sistema Embutido Multiprocessado esse tipo de interconexão compromete o desempenho desejado (MELLO, 2003), sendo então realizada através de uma rede intrachip.

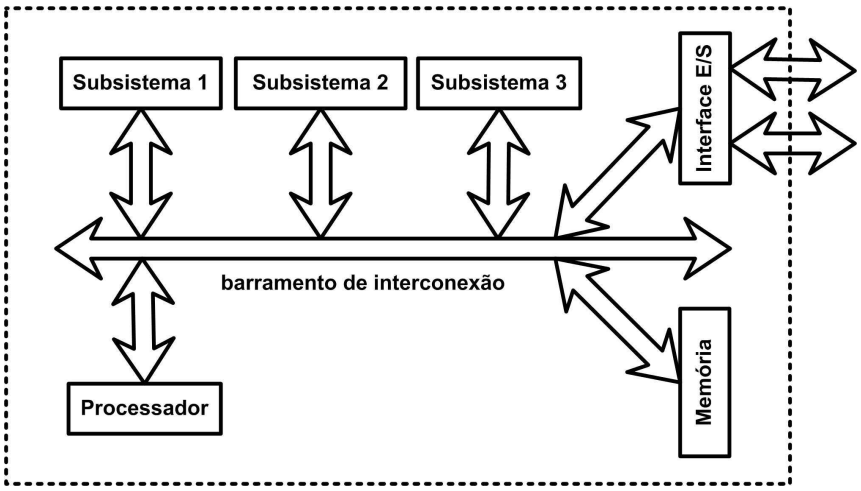


Figura 1: Estrutura interna de um SoC

O *software* utilizado para controle de sistemas embutidos pode variar de um simples *firmware*, gravado em uma memória ROM (Read Only Memory), até um sistema operacional embutido.

1.2 Rede Intrachip

Rede Intrachip ou *Network on Chip* (NoC) é uma plataforma utilizada para interconectar os subsistemas, também chamados de recursos, de um sistema embutido multiprocessado. O recurso pode ser um processador, uma memória, um dispositivo de entrada-e-saída ou um outro dispositivo dedicado de *hardware*. O componente mais simples utilizado para realizar essa interconexão é a chave, mostrada na Figura 3. As informações trocadas entre os recursos são transferidas na forma de mensagens, as quais podem ser divididas em unidades menores chamadas pacotes (ZEFERINO, 2003). A chave permite a retransmissão de mensagens de qualquer

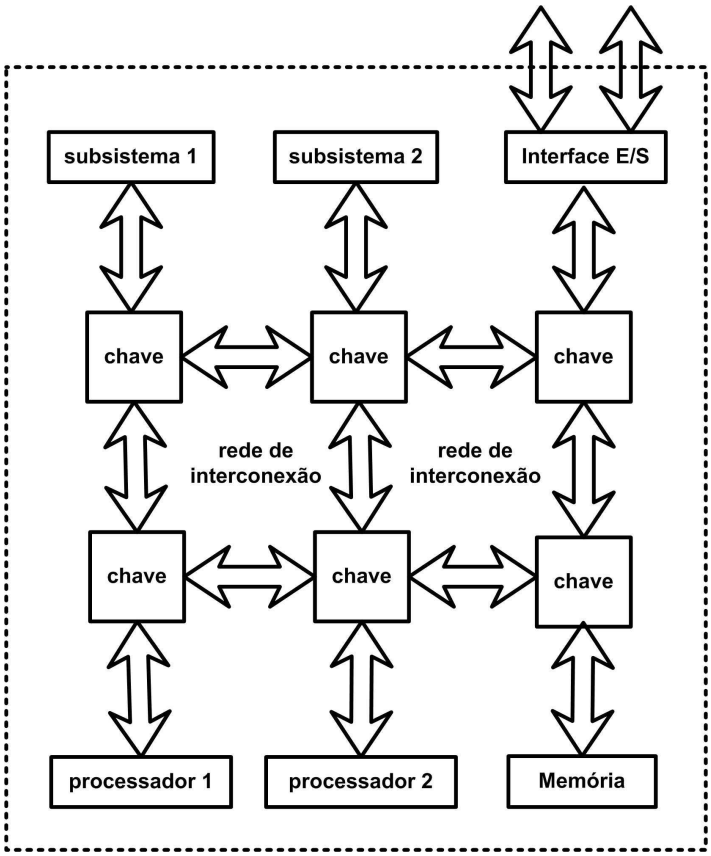


Figura 2: Estrutura interna de um MPSoC

módulo para outro e tomam decisões relacionadas ao caminho que essas mensagens devem seguir, sendo o principal componente da rede intrachip. Cada chave possui um conjunto de portas bidirecionais para a interconexão com um recurso e com as chaves vizinhas. As redes de comunicação empregadas em sistemas embutidos multiprocessados utilizam conceitos originados na área de redes de computadores, comunicação de dados e sistemas distribuídos, tais como: organização da transferência de dados em camadas de protocolos, topologias de conexão, técnicas de comunicação e técnicas de roteamento.

O recurso é constituído por um dispositivo capaz de processar informação, podendo ser tanto um processador quanto uma memória local, ou um dispositivo criado para realizar uma tarefa específica, como a decodificação MPEG (WOSZEZENKI, 2007), ou uma memória compartilhada (GIRãO, 2007), ou outro dispositivo de entrada-e-saída, como uma porta serial que permite a comunicação de um computador de propósito geral com o sistema embutido multiprocessado.

A chave é constituída por um núcleo de chaveamento *crossbar*, uma lógica de controle para roteamento e arbitragem, além de portas de comunicação para interconexão com outras

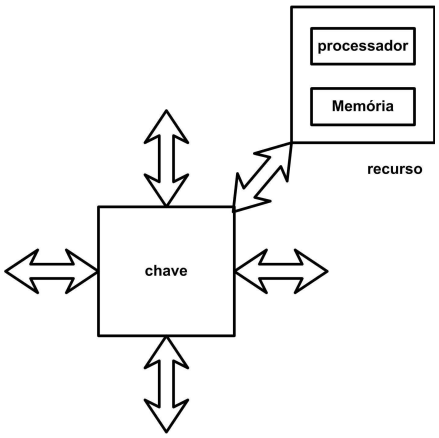


Figura 3: Chave interconectada com um recurso

chaves e com os recursos do sistema embutido multiprocessado. Essas portas de comunicação podem possuir uma memória para armazenamento temporário de dados (*buffer*), como também controladores de enlace para a implementação do protocolo físico de comunicação.

As informações trocadas entre os recursos são transferidas na forma de mensagens (ZEFERINO, 2003), que possuem, em geral, três partes: um cabeçalho (*header*), uma carga útil de dados (*payload*) e um terminador (*trailer*). O cabeçalho inclui informações de roteamento e de controle utilizadas pelas chaves para propagar a mensagem em direção ao seu destino. O terminador, por sua vez, inclui informações utilizadas para a detecção de erros e para a sinalização do fim da mensagem.

Normalmente, as mensagens são quebradas em pacotes para transmissão. Um pacote é a menor unidade de informação que contém detalhes sobre o roteamento e sequenciamento dos dados, mantendo uma estrutura semelhante à de uma mensagem, com um cabeçalho, uma carga útil de dados e um terminador. Um pacote é constituído por uma sequência de palavras, denominadas *flits* (FLow control unITS ou unidades de controle de fluxo) (ZEFERINO, 2003) (WOSZEZENKI, 2007), cuja largura é igual à largura física do canal.

1.2.1 O modelo de referência OSI

O modelo de referência OSI (ZIMMERMANN, 1980) (Open Systems Interconnection - Interconexão de Sistemas Abertos) é um padrão internacional empregado como base para muitos sistemas de comunicação, à semelhança do que ocorre na rede intrachip. A arquitetura de um sistema de comunicação que segue esse modelo é formada por camadas, conforme pode ser visto na Figura 4, interfaces e protocolos. Cada camada oferece serviços para a camada superior, utilizando serviços existentes na própria camada e nas camadas inferiores. Cada camada é

feita de entidades que podem ser de *hardware* ou *software*. Uma entidade se comunica com outra entidade da mesma camada, mas em outra localização da rede. A interface, vista na Figura 5, entre uma camada inferior e superior é chamada de Ponto de Acesso a um Serviço *Service Access Point* (SAP). Os protocolos são um conjunto de regras criadas para permitir a comunicação entre as entidades. A principal vantagem de utilizar um modelo em camadas é a possibilidade de modificar a implementação de uma camada sem afetar as outras. As camadas do modelo de referência OSI são chamadas de pilha de protocolos e são apresentadas a seguir.

- Camada física: realiza a transferência confiável em *bits* através de um meio físico. Essa camada lida com as características mecânicas, elétricas e funcionais para o acesso do meio físico.

- Camada de enlace de dados: realiza a transferência confiável de dados em quadros (grupos de *bits*) através da camada física. Essa camada é responsável pela sincronização dos dados, controle de fluxo de dados e controle de erro.

- Camada de rede: realiza a transferência confiável de dados em pacotes (grupos de quadros). Essa camada é responsável por empacotar mensagens, estabelecer conexões, rotear pacotes, contabilizar os pacotes transferidos, controlar o congestionamento da rede, manter conexões e terminar conexões.

- Camada de transporte: estabelece uma conexão fim-a-fim entre a origem e o destino da mensagem realizando a transferência confiável de dados de forma transparente. Essa camada é responsável pelo controle de fluxo de pacotes, segmentação de pacotes e remontagem de pacotes.

- Camada de sessão: fornece a estrutura de controle para a comunicação entre aplicações. Essa camada é responsável por estabelecer, gerenciar, e terminar conexões (sessões) entre as aplicações.

- Camada de apresentação: realiza a conversão do formato dos dados recebidos da camada de aplicação em um formato comum a ser utilizado na transmissão desses dados, ou seja, um formato entendido pelo protocolo utilizado. Essa camada é responsável por converter formato de dados, converter dados e criptografar dados.

- Camada de aplicação: realiza a interface entre o protocolo de comunicação e o aplicativo que pediu ou receberá a informação através da rede.

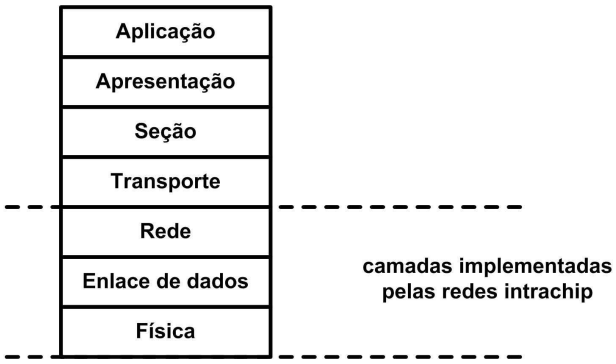


Figura 4: Camadas do modelo OSI

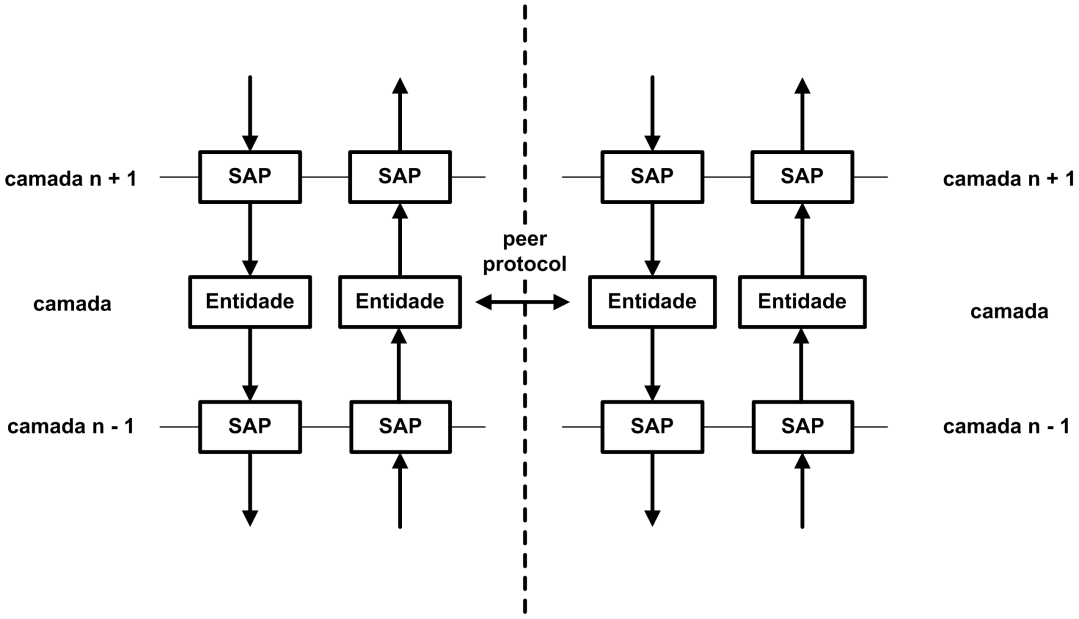


Figura 5: Pontos de acesso a serviços e entidades

A implementação de um modelo em camadas pode variar, mas, de um modo geral, a ideia é simplificar as funções envolvidas em um sistema de comunicação (AMARAL, 2008). Normalmente, as redes intrachip implementam apenas funções das camadas física, enlace de dados e rede.

1.2.2 Topologias

As topologias de redes intrachip podem ser agrupadas em duas classes principais: as redes diretas (ZEFERINO, 2003) (MELLO, 2003) e as redes indiretas (ZEFERINO, 2003) (MELLO, 2003).

As redes diretas são caracterizadas pelo recurso conectado diretamente à chave e esse par pode ser visto como um elemento único do sistema embutido multiprocessado, sendo refe-

renciado geralmente pelo termo nó, como mostrado na Figura 6. As topologias de redes diretas mais utilizadas, mostradas na Figura 7, são: *mesh*, toroide e o hipercubo.

As redes indiretas são caracterizadas pelo recurso conectado a uma interface para uma rede de chaves, não formando um elemento único como nas redes diretas. Cada chave possui um conjunto de portas para a interconexão com outras chaves ou recursos. Somente algumas chaves possuem interconexão com recursos e apenas esses podem servir de origem ou destino de uma mensagem. As topologias de redes indiretas mais utilizadas são *crossbar* e multiestágio. Para conexão indireta de N nós de processamento, a topologia *crossbar* é a ideal, pois consiste de um único roteador com uma chave $N \times N$ capaz de ligar qualquer entrada a qualquer saída. A Figura 8 mostra uma rede *crossbar* constituída de um roteador 4×4 (quatro portas de entrada e quatro portas de saída) e uma rede multiestágio 8×8 bidirecional.

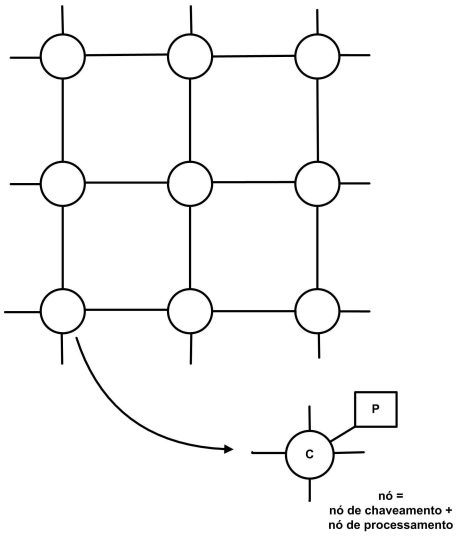


Figura 6: Nó de rede direta

1.2.3 Métodos de chaveamento

Em uma rede intrachip, o chaveamento define a forma como os dados são transferidos entre a chave de origem e a chave de destino. Os dois métodos mais utilizados são chaveamento de circuitos e chaveamento de pacotes.

No chaveamento de circuitos, um caminho é estabelecido antes do envio da mensagem. Quando um circuito entre a origem e o destino foi estabelecido, a mensagem pode ser enviada e qualquer requisição de comunicação no canal alocado será recusada. A vantagem desse método é que não são necessárias filas nas chaves intermediárias, uma vez que quando a comunicação é estabelecida a mensagem não é bloqueada. A desvantagem é que esse método causa a perda

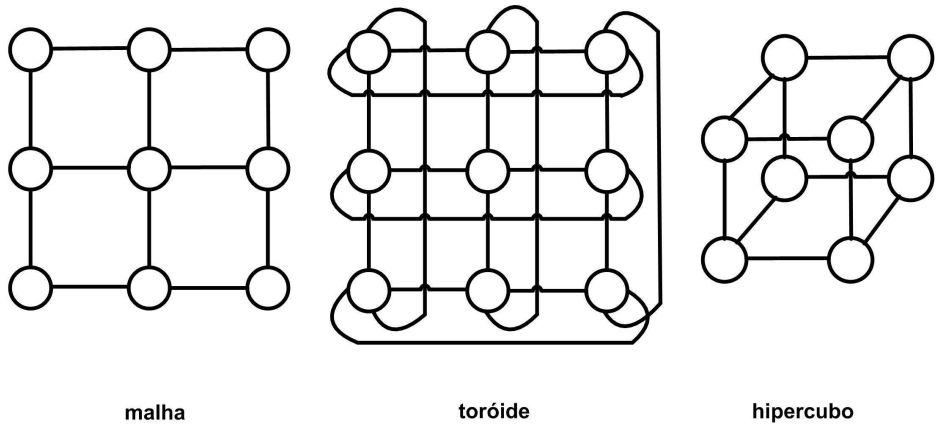


Figura 7: Topologias malha, toroide, hipercubo

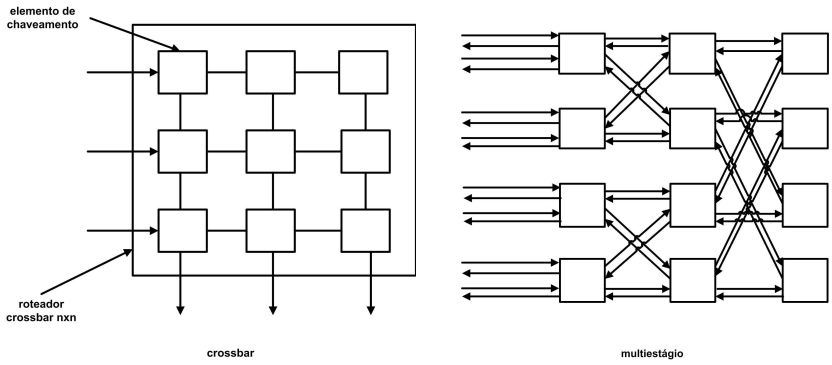


Figura 8: Topologias *crossbar* e multiestágio

de desempenho do sistema como um todo, devido ao fato do caminho da mensagem entre a chave de origem e a chave de destino ficar reservado durante a transmissão de dados.

No chaveamento de pacotes, a mensagem é dividida em vários pacotes que são transmitidos pela rede. Cada pacote possui um cabeçalho que é verificado na chegada de cada chave intermediária. A chave intermediária, com base no cabeçalho do pacote, decide para qual porta de saída ela deve enviar o pacote. A vantagem desse método é que o caminho permanece ocupado apenas quando o pacote está sendo transferido. A desvantagem é que torna-se necessária a utilização de filas para o armazenamento temporário dos pacotes. Os principais métodos de chaveamento de pacotes são *Store-And-Forward*, *Virtual-Cut-Through*, *Wormhole* e *Deflection Routing*.

No método *Store-And-Forward*, o pacote inteiro é armazenado, para só então ser enviado pela rede. Isto implica na necessidade de uma fila capaz de armazenar o pacote inteiro, acarretando uma alta latência em cada chave intermediária.

No método *Virtual-Cut-Through*, que é um aperfeiçoamento do método *Store-And-Forward*, o pacote inteiro só é armazenado se a chave de destino estiver ocupada. A vantagem desse método em relação ao *Store-And-Forward* é que é possível reduzir a latência quando a chave seguinte não estiver ocupada.

No método *Wormhole*, o pacote é dividido em *flits*, que são transmitidos entre as chaves intermediárias até o destino. Esse método funciona como um *pipeline*, onde os *flits* do cabeçalho, que contém a informação de destino, se movem pela rede e todos os *flits* da carga útil de dados (*payload*) os seguem (TOTA; CASU; MACCHIARULO, 2006)(KARAIVAZOGLOU; SPIRAKIS; TRIANTAFILOU, 1996). Quando os *flits* do cabeçalho são bloqueados, os *flits* da carga útil de dados ficam armazenados nas filas das chaves intermediárias. A vantagem desse método é que a latência não depende da distância, como nos métodos anteriores, mas do tráfego entre as chaves de origem e destino. Outra vantagem é que o tamanho das filas das chaves intermediárias pode ser reduzido, já que não precisam armazenar o pacote inteiro. A desvantagem é a contenção de recursos causada pelo bloqueio do pacote.

No método *Deflection Routing*, também conhecido como *Hot Potato*, cada pacote que chega em uma chave deve ser enviado para a próxima no próximo ciclo de *clock* (TOTA; CASU; MACCHIARULO, 2006)(KARAIVAZOGLOU; SPIRAKIS; TRIANTAFILOU, 1996)(NILSSON, 2002). A vantagem desse método é que não existe a necessidade de filas na chave. Outra vantagem é que a chave ocupa menos espaço no *chip* e consome menos energia. Mais uma vantagem é que não existe o problema de bloqueio do pacote, como ocorre no *wormhole* (TOTA; CASU; MACCHIARULO, 2006). A desvantagem é que esse método não garante a entrega ordenada dos *flits* de um pacote.

1.2.4 Algoritmos de roteamento empregados em redes intrachip

Em uma rede intrachip, o roteamento define a forma pela qual os dados são transferidos de uma porta de entrada da chave para outra de saída. A seguir, são apresentados vários algoritmos de roteamento existentes na literatura, classificados segundo os critérios: quanto ao local de decisão de roteamento, quanto ao momento de realização do roteamento, quanto ao número de destinatários, quanto à implementação, quanto ao número de caminhos possíveis e quanto ao caminho percorrido.

Quanto ao local onde as decisões de roteamento são tomadas, o algoritmo pode ser origem, distribuído ou centralizado. No roteamento origem, o caminho de cada pacote de uma mensagem é decidido na chave de origem antes do mesmo ser enviado na rede. As desvantagens

desta abordagem são (MELLO, 2003) que o cabeçalho do pacote deve conter todas as informações de roteamento e não é tolerante a falhas, ou seja, se um caminho se encontra defeituoso, todas as mensagens que deveriam passar por esse caminho serão bloqueadas. No roteamento distribuído, o caminho de cada pacote de uma mensagem é decidido em cada chave onde o mesmo chega. Já no roteamento centralizado, o caminho de cada pacote de uma mensagem é decidido por um controlador central na rede.

Quanto ao momento de realização do roteamento, o algoritmo pode ser estático, se o caminho de cada pacote de uma mensagem for decidido durante a compilação de uma aplicação, ou dinâmico, se o caminho de cada pacote de uma mensagem for decidido durante a execução de uma aplicação.

Quanto ao número de destinatários, o roteamento pode ser *unicast*, se o caminho de cada pacote de uma mensagem possuir um único destino, ou *multicast*, se o caminho de cada pacote de uma mensagem possuir múltiplos destinos.

Quanto à implementação, o roteamento pode ser baseado em tabela, se o caminho de cada pacote de uma mensagem for decidido a partir da consulta a uma tabela armazenada em memória, ou baseado em máquina de estados, se o caminho de cada pacote de uma mensagem for decidido a partir da execução de um algoritmo implementado em *hardware* ou *software*.

Quanto ao número de caminhos possíveis, o roteamento pode ser determinístico, se cada pacote de uma mensagem seguir sempre o mesmo caminho entre a origem e o destino, ou adaptativo, se o caminho de cada pacote de uma mensagem for definido em função do tráfego na rede. A vantagem dessa abordagem é que o pacote possui mais de uma alternativa para chegar ao destino (MELLO, 2003). A desvantagem é a possibilidade de *deadlock* (ver Seção 1.2.5) e entrega de pacotes fora da ordem (MELLO, 2003). Esses algoritmos podem ainda ser classificados quanto aos critérios: progressividade, minimalidade e número de caminhos.

Quanto à progressividade, o roteamento pode ser progressivo, se os cabeçalhos dos pacotes de cada mensagem sempre avançarem pela rede, reservando um novo caminho a cada chave por onde passarem, ou regressivo, se os cabeçalhos dos pacotes de cada mensagem retornarem pela rede, liberando caminhos anteriormente reservados.

Quanto à minimalidade, o roteamento pode ser não mínimo, se cada pacote de uma mensagem pode escolher qualquer caminho entre a origem e o destino. A vantagem dessa abordagem é que os pacotes da mensagem podem evitar caminhos bloqueados. A desvantagem é que isso pode causar *livelock* (ver Seção 1.2.5). O roteamento é dito mínimo quando os pacotes de uma mensagem são roteados por um dos menores caminhos entre a origem e o

destino. A vantagem dessa abordagem é a garantia de que, a cada chave por onde passam, os pacotes da mensagem se aproximam mais do destino. Outra vantagem é que essa abordagem evita o *livelock* do caminho não mínimo. A desvantagem é que esse algoritmo fica aguardando o caminho escolhido até este ser liberado.

Quanto ao número de caminhos, o roteamento pode ser completo, se cada pacote de cada mensagem puder utilizar todos os caminhos possíveis para chegar ao destino, ou parcial, se cada pacote de cada mensagem utilizar apenas um subconjunto dos caminhos possíveis para chegar ao destino.

1.2.5 Tráfego de pacotes em redes intrachip

Uma rede intrachip tem a função principal de oferecer o suporte físico necessário à comunicação entre os seus recursos. A rede transporta pacotes através das chaves e das interconexões entre elas. Uma comunicação é realizada com sucesso quando uma mensagem enviada pelo recurso de origem é devidamente recebida pelo recurso de destino. Entretanto, existem três situações que podem impedir que os pacotes de uma mensagem não cheguem ao seu destino: *starvation*, *livelock* e *deadlock*.

A situação de *starvation* ocorre em uma chave, quando um pacote de uma fila de entrada requisita uma porta de saída e é bloqueado porque essa porta está sempre alocada para outro pacote. Essa situação pode ser evitada por um mecanismo adequado de arbitragem de filas.

A situação de *livelock* ocorre quando um pacote circula permanentemente pela rede porque os caminhos necessários para que ele chegue ao seu destino estão sempre ocupados. Esse problema ocorre normalmente em algoritmos de roteamento adaptativos não mínimos. Isto pode ser evitado com estratégias de roteamento adaptativo que restrinjam o número de desvios que o pacote pode realizar.

A situação de *deadlock* é a mais difícil de resolver e ocorre quando existe uma dependência cíclica entre nós ou chaves requisitando acesso a um conjunto de recursos, de forma que nenhum possa obter progresso algum, independente da sequencia de eventos que ocorra.

1.2.6 Arquiteturas de redes intrachip

Diversas arquiteturas de redes intrachip tem sido propostas na literatura, sendo algumas delas apresentadas a seguir.

1.2.6.1 Hermes

A rede intrachip Hermes (MORAES et al., 2004), desenvolvida pela Faculdade de Informática da Pontífícia Universidade Católica do Rio Grande do Sul (FACIN/PUC-RS), utiliza a chave Hermes, mostrada na Figura 9, que possui cinco portas bidirecionais (norte, sul, leste, oeste e local), cada uma contendo uma fila de tamanho parametrizável, utilizada para a interconexão com outras chaves ou blocos IP. A chave Hermes possui um controle que implementa a lógica de arbitragem e o algoritmo de roteamento.

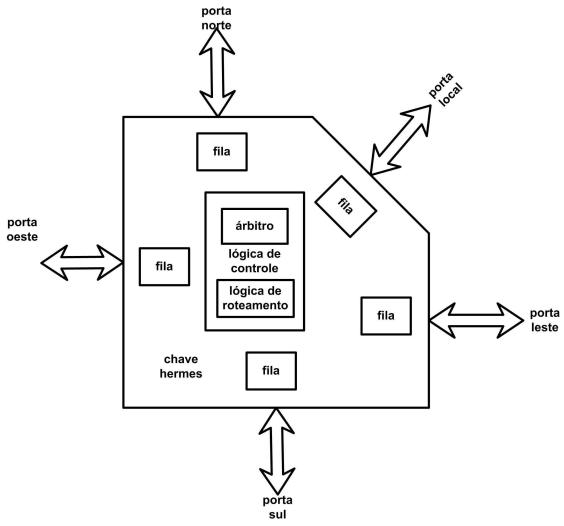


Figura 9: Chave Hermes

A técnica de chaveamento empregada é de pacotes, utilizando o método *wormhole* e o algoritmo de roteamento distribuído, adaptativo e mínimo. A topologia empregada é a malha. Na implementação, o número de portas da chave depende da localização da mesma na rede. Isso implica em até 9 modelos diferentes de chave, conforme mostrado na Figura 10.

1.2.6.2 SoCIN

A rede intrachip SoCIN (*System On Chip Interconnection Network*) (ZEFERINO, 2003), desenvolvida no Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS), utiliza a chave RASoC (*Router Architecture for Systems on Chip*), mostrada na Figura 11. Esta rede também possui cinco portas bidirecionais (norte, sul, leste, oeste e local), cada uma contendo uma fila de tamanho parametrizável, utilizada para a interconexão com outras chaves ou blocos IP. A chave RASoC possui uma chave *crossbar* 5×5 parcial que implementa 20 das 25 conexões que poderiam ser realizadas em uma chave *crossbar* com essas dimensões. Isso se deve ao fato de que não é permitido a um canal de entrada de uma porta ser conectado ao

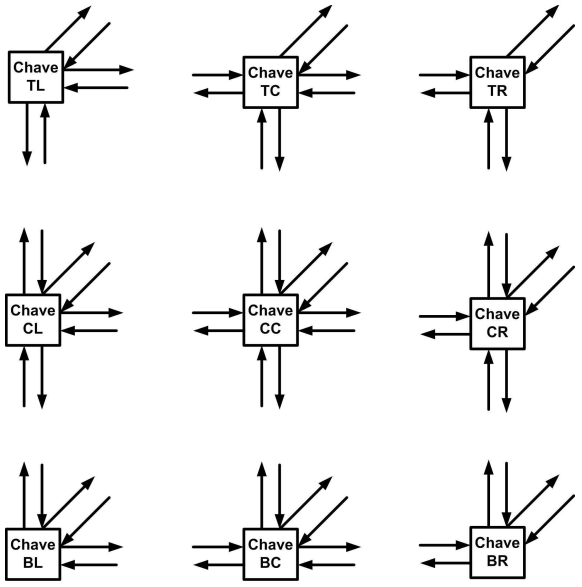


Figura 10: Os nove tipos de chaves possíveis

canal de saída associado à mesma porta. Em outras palavras, um pacote que chega ao canal de entrada da porta oeste de um roteador não pode ser encaminhado ao canal de saída dessa porta. Nesse caso, os únicos canais de saída possíveis de serem utilizados por esse pacote são aqueles associados às portas local, norte, leste e sul.

A topologia empregada pela rede intrachip SoCIN é a malha. A técnica de chaveamento empregada é de pacotes, utilizando o método *wormhole* e o algoritmo de roteamento origem e determinístico.

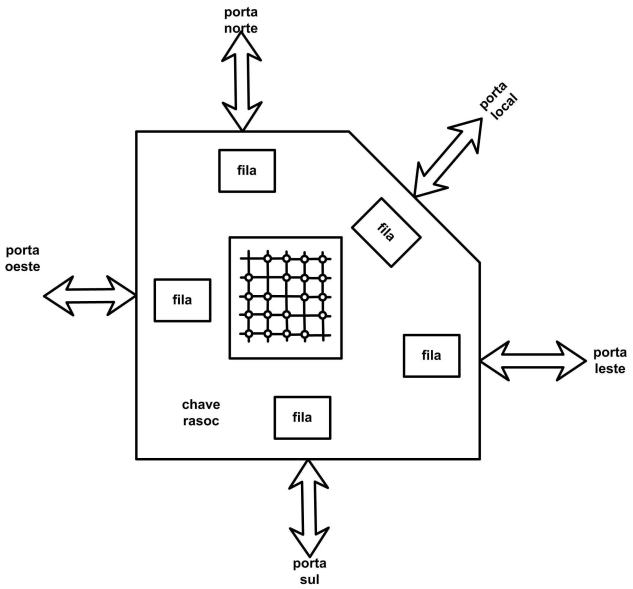


Figura 11: Chaves RASoC

1.2.6.3 Nostrum

A rede intrachip Nostrum (MILLBERG et al., 2004), desenvolvida no KTH (Kungliga Tekniska Högskolan - Instituto de Tecnologia Real, Suécia) utiliza a chave Nostrum, que também possui cinco portas bidirecionais (norte, sul, leste, oeste e local) de largura de barramento parametrizável podendo atingir a largura de 128 *bits*, sem filas, utilizada para a interconexão com outras chaves ou blocos IP.

A topologia empregada pela rede intrachip Nostrum é a malha. A técnica de chaveamento empregada é de pacotes, utilizando o método *deflection routing* e o algoritmo de roteamento distribuído, adaptativo e não mínimo.

1.2.6.4 SoCBUS

A rede intrachip SoCBUS (WIKLUND, 2005), desenvolvida na Universidade Linköping, Suécia, utiliza a chave SoCBUS, que possui várias portas bidirecionais (não há um limite no número de portas em uma chave SoCBUS (WIKLUND, 2005)), utilizadas para a interconexão com outras chaves ou blocos IP. Cada bloco IP é conectado com a chave por meio de uma interface de rede (*wrapper*).

A topologia empregada pela rede intrachip SoCBUS é a malha. A técnica de chaveamento empregada é um modelo híbrido de circuito-pacote, conhecido como circuito de pacote conectado (*Packet Connected Circuit* – PCC) e o algoritmo de roteamento utilizado é distribuído, adaptativo e mínimo.

1.2.6.5 Proteo

A rede intrachip Proteo (SIGUENZA-TORTOSA, 2002) está sendo desenvolvida na TUT (Tampere University of Technology - Universidade de Tecnologia de Tampere, Finlândia), sendo uma proposta para arquitetura de rede. Nesse projeto, o foco consiste em pesquisar novos protocolos, arquiteturas e implementações de blocos IP, deixando de lado as ferramentas de *software*.

A topologia empregada pela rede intrachip Proteo é anel, com várias sub-redes com diferentes topologias, formatos de pacotes e desempenho, conectadas aos *hubs* da rede em anel. Atualmente, estão sendo utilizadas sub-redes com topologias em estrela ou barramento.

As redes intrachip Hermes, SoCIN, Nostrum e SoCBUS podem utilizar outras topologias, tais como a toroide e a hipercubo. Porém, implementações nessas topologias implicam em mudanças nas conexões das chaves e no algoritmo de roteamento.

1.3 Processadores para Sistemas Embutidos

Sistemas embutidos são projetados tendo por base processadores de propósito geral e coprocessadores de propósito específico, como, por exemplo, processadores de áudio, vídeo e sinais digitais. Processadores de propósito geral para sistemas embutidos possuem aspectos diferentes dos processadores encontrados nos computadores pessoais. Um computador pessoal deve suportar uma ampla variedade de aplicações, tais como processadores de texto, planilhas eletrônicas, apresentações, ferramentas de projeto assistido CAD (*Computer Aided Design*), jogos, multimídia e o sistema operacional em si. Por outro lado, um sistema embutido deve suportar um conjunto dedicado de aplicações.

A arquitetura do conjunto de instruções (*Instruction Set Architecture* - ISA) de um processador de alto desempenho tende a ser bem mais complexa do que a de um processador de propósito geral para sistemas embutidos. Diversos processadores são utilizados por sistemas embutidos. Alguns dos mais conhecidos são apresentados a seguir.

1.3.1 ARM 1136JF-S

O processador ARM 1136JF-S (ARM, 2008) é um processador RISC (*Reduced Instruction Set Computer*) de 32 *bits*, encontrado em quase todas as áreas da eletrônica de consumo, desde dispositivos portáteis, como telefones celulares, PDAs, iPods, MP3 e MP4 *players*, calculadoras, até periféricos de computadores, como discos rígidos e roteadores. O ARM 1136JF-S pode suportar até 16 coprocessadores (0 - 15), sendo que o coprocessador 15 é reservado para a unidade de gerenciamento de memória (*Memory Management Unit* – MMU). A arquitetura do conjunto de instruções do ARM 1136JF-S inclui *Thumb*, um conjunto de instruções de 16 *bits* para código compacto; DSP, um conjunto de extensões aritméticas para processamento digital de sinais e aplicações de multimídia ; e *Jazelle*, uma extensão que permite a execução direta de *byecode* Java. Um diagrama em blocos do ARM 1136JF-S é mostrado na Figura 12.

1.3.2 IBM PowerPC 440

O Processador PowerPC 440 (IBM, 2008) é um processador RISC de 32 *bits*, concebido para uma variedade de aplicações, tais como microcontroladores, sistemas embutidos, até supercomputadores. Possui uma arquitetura com *pipeline* superescalar de sete estágios, com suporte a duas instruções por ciclo. A memória *cache* de dados é separada da *cache* de instruções. Há 32 registradores de propósito geral, unidade de gerenciamento de memória, interface para o barramento *CoreConnect*, interface para *cache* L2 com até 256 KB e interface para um processador

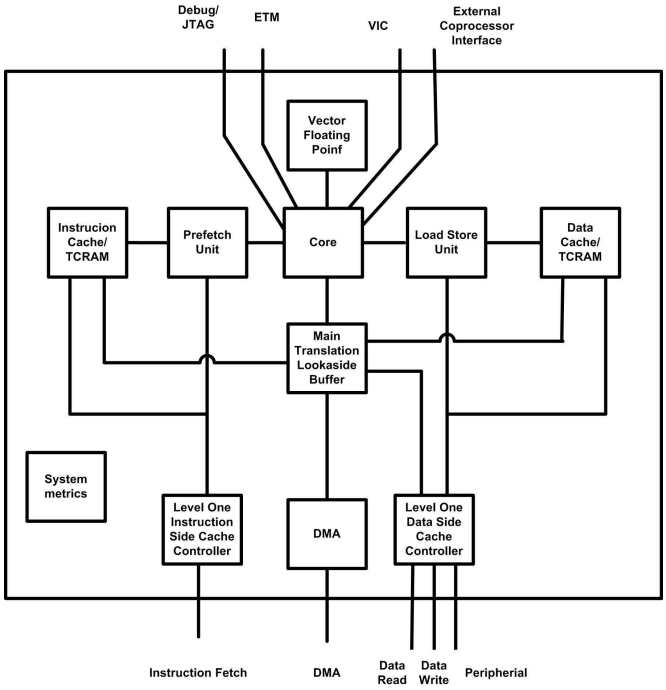


Figura 12: Diagrama em blocos do processador ARM 1136

auxiliar (*Auxiliar Processor Unit* – APU). Esse processador auxiliar pode ser uma unidade de ponto flutuante (*Floating Point Unit* – FPU), um processador de sinais digitais (*Digital Signal Processor* – DSP) ou um outro processador auxiliar. O diagrama em blocos do PowerPC 440 é mostrado na Figura 13.

1.3.3 MIPS32 24Kf

O MIPS32 24Kf (MIPS, 2008) é um processador RISC de 32 *bits* sintetizável para aplicações embutidas, com uma arquitetura *pipeline* de oito estágios. Apresenta uma unidade de ponto flutuante que suporta instruções de precisão simples e dupla, além de uma unidade de multiplicação/divisão (*Multiple/Divide Unit* – MDU) de alto desempenho. As memórias *cache* de dados e de instruções podem ser configuradas para operar com 0, 16, 32 e 64KB. A unidade de interface com o barramento (*Bus Interface Unit* – BIU) implementa o padrão de protocolo de núcleo aberto (*Open Core Protocol* – OCP)). Interfaces opcionais suportam blocos externos, como coprocessadores. O módulo EJTAG (*Enhanced JTAG*) fornece suporte para depuração. O diagrama em blocos do MIPS32 24Kf é mostrado na Figura 14.

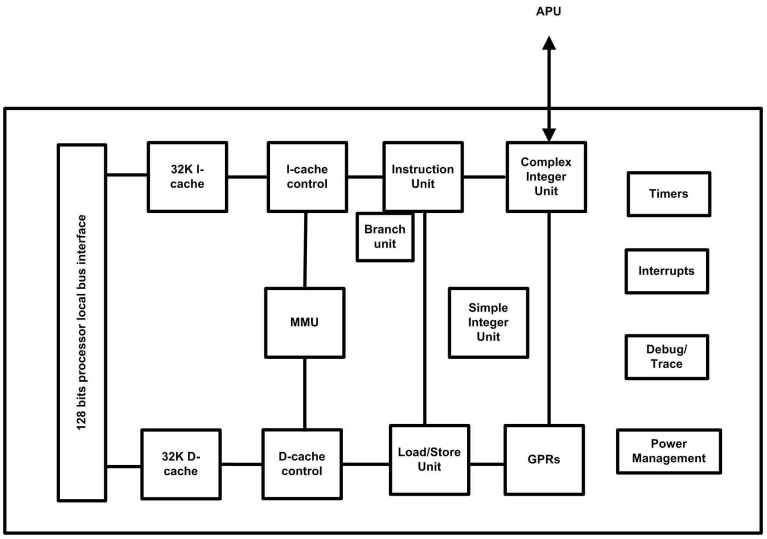


Figura 13: Diagrama em blocos do processador PowerPC 440

1.4 Sistemas Operacionais Embutidos

Sistemas operacionais são programas responsáveis por controlar os recursos existentes em um computador (processadores, memórias, dispositivos de E/S), servindo de interface entre o mesmo e o usuário, além de fornecer a base para o desenvolvimento de aplicações. As aplicações são compostas de tarefas, sendo uma tarefa um conjunto de instruções e dados com informações necessárias para a sua correta execução pelo processador.

Os sistemas operacionais dispõem de diversos tipos de serviços. Entretanto, levando em consideração a maioria das implementações de sistemas operacionais existentes atualmente, pode-se dizer que os principais serviços implementados no núcleo ou *kernel* de um sistema operacional são: escalonamento de tarefas, troca de contexto, comunicação entre tarefas, tratamento de interrupções, gerenciamento de memória e gerenciamento de sistemas de arquivos.

Sistemas operacionais embutidos são considerados um subgrupo de sistemas operacionais e implementam somente as funcionalidades necessárias pela a aplicação que será executada. O tamanho de um sistema operacional embutido é muito menor que o de um sistema operacional convencional, reduzindo o *kernel* a um *microkernel*, o que é desejável para sistemas embutidos, tendo em vista o tamanho limitado da memória RAM desses sistemas. Os principais sistemas operacionais embutidos atualmente são: *Embedded Linux* (TORVALDS, 2008), *Windows CE* (MICROSOFT, 2008), QNX RTOS (QNX, 2008), eCos (ECOSCENTRIC, 2008) e o EPOS (UFSC, 2008).

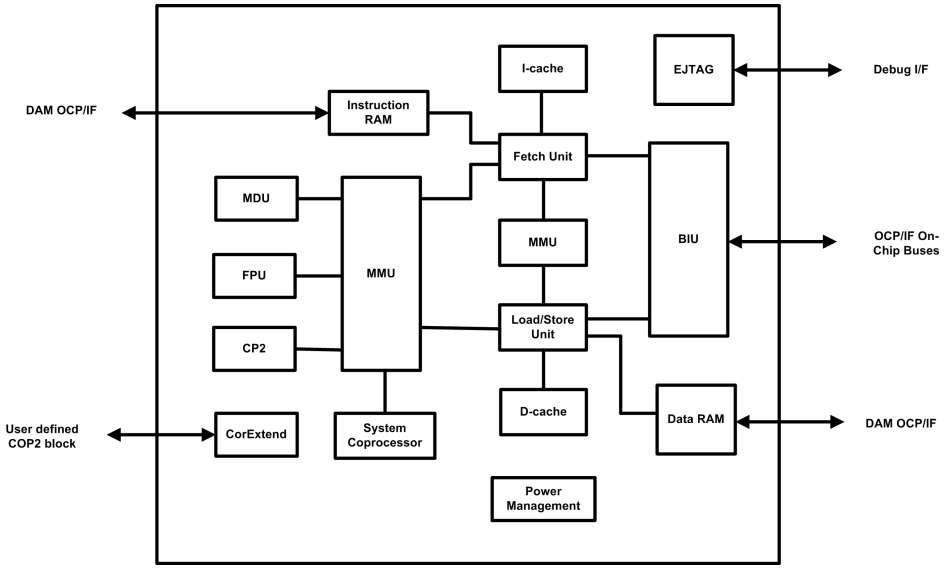


Figura 14: Diagrama em blocos do processador MIPS32 24Kf

1.4.1 Embedded Linux

O *Embedded Linux* é uma versão reduzida do Linux utilizada em vários sistemas embutidos como telefones celulares, PDAs, MP3 e MP4 *players*, chaves, roteadores, eletrônica automotiva, automação industrial, equipamentos de navegação e instrumentos médicos.

1.4.2 Windows CE

O *Windows CE* é uma versão do *Windows* que utiliza um subconjunto da Win32 API (*Application Programming Interface*) adequada para a maioria das aplicações embutidas, sendo portado para um vasto número de dispositivos industriais, de negócios e eletrônica de consumo, como controladores lógicos programáveis, leitores de códigos de barras, câmeras digitais e *Handheld* PC.

1.4.3 QNX RTOS

O QNX RTOS (*Real Time Operating System*) é um sistema operacional de alta confiabilidade, desenvolvido para aplicações embutidas, principalmente eletrônica de consumo, telecomunicações, sistemas automotivos e instrumentação médica, que necessitam de desempenho elevado, funcionalidade sofisticada e escalabilidade maciça. Esse sistema operacional é pequeno, escalável, extensível e rápido.

1.4.4 eCos

O eCos é um sistema operacional que utiliza diversas ferramentas de configuração, construção, compiladores e simuladores do projeto GNU (GNU, 2009c). Foi concebido para aplicações embutidas dedicadas e portado para diversas arquiteturas de microprocessadores e microcontroladores de 16, 32 e 64 *bits*.

1.4.5 EPOS

O EPOS é um sistema operacional orientado à aplicação, ou seja, adapta-se automaticamente aos requisitos da aplicação elaborada pelo usuário. Foi concebido para aplicações embutidas dedicadas e portado para diversas arquiteturas de microprocessadores.

1.5 Considerações Finais

Este capítulo apresentou uma breve introdução de sistemas embutidos multiprocessados. Os principais conceitos relacionados a redes intrachip, os processadores utilizados em sistemas embutidos e os sistemas operacionais empregados nos mesmos foram apresentados. No capítulo seguinte, será descrita a infra-estrutura de *hardware* do sistema embutido multiprocessado utilizado neste trabalho e as mudanças realizadas no respectivo modelo.

Capítulo 2

PLATAFORMA HMPS DE REDE INTRACHIP

ESSE capítulo apresenta a infra-estrutura de *hardware* da plataforma HMPS (Hermes MultiProcessor System on chip - Sistema Embutido Multiprocessado Hermes), onde será executado o algoritmo genético paralelo. Esta infra-estrutura consiste basicamente da rede intrachip Hermes (MORAES et al., 2004) e do processador Plasma (RHOADS, 2006). A rede intrachip e o processador Plasma são componentes não desenvolvidos no presente trabalho, mas cujos modelos são de domínio público. Na Seção 2.1, é apresentada a estrutura interna da chave utilizada pela rede intrachip Hermes, juntamente com o seu funcionamento. Em seguida, na Seção 2.2, é apresentado o processador Plasma, juntamente com os seus componentes principais. Na Seção 2.3, são apresentadas as mudanças no modelo da plataforma realizadas neste trabalho.

2.1 Rede Intrachip Hermes

Para a interconexão dos processadores da plataforma HMPS e o roteamento de pacotes é utilizada a rede intrachip Hermes, mostrada na Figura 15. A plataforma HMPS foi desenvolvida pelo grupo de pesquisa GAPH (GAPH, 2006). A rede intrachip Hermes, que utiliza a chave de mesmo nome, emprega a técnica de comunicação de dados denominada chaveamento de pacotes, descrita na Seção 1.2.3 (do Capítulo 1). O método de chaveamento empregado, para definir como os pacotes devem se mover através das chaves, é o *wormhole*, também introduzido na Seção 1.2.3 (do Capítulo 1).

A rede intrachip Hermes utiliza uma topologia em malha, definida na Seção 1.2.2 (de Capítulo 1), onde o recurso corresponde ao processador Plasma e o número alocado à chave representa o endereço da mesma, correspondendo à posição XY na rede. Cada processador Plasma possui uma memória local, não acessível pelos outros processadores (MELLO et al., 2005).

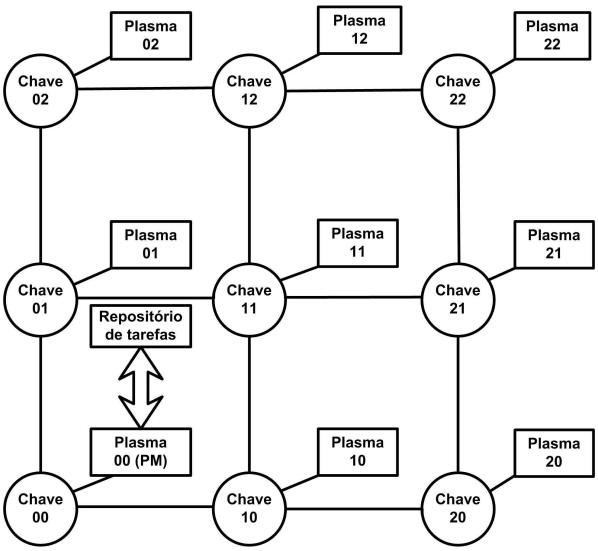


Figura 15: Rede intrachip Hermes

2.1.1 A chave Hermes

A chave Hermes contém uma lógica de controle de roteamento e 5 portas bidirecionais, designadas Leste, Oeste, Norte, Sul e Local, como pode ser visto na Figura 16. A porta Local estabelece a comunicação entre a chave e o processador Plasma. As demais portas ligam a chave às chaves vizinhas. A Tabela 1 descreve os sinais de interfaceamento externo da chave. Um exemplo de conexão entre duas chaves Hermes vizinhas pode ser visto na Figura 17.

Tabela 1: Sinais de interfaceamento da chave Hermes

Sinal	Tipo	# de bits	Descrição
clock	Entrada	1	Sinal de *clock* da chave
reset	Entrada	1	Sinal de *reset* da chave
clock_tx	Saída	1	*Clock* da porta de saída que sincroniza a transmissão de dados
data_out	Saída	16 ou 32	Saída de dados
Tx	Saída	1	Informa que a chave tem dado para enviar
credit_in	Entrada	1	Informa que a chave pode enviar dados
clock_Rx	Entrada	1	*clock* da porta de entrada que sincroniza a recepção de dados
data_in	Entrada	16 ou 32	Entrada de dados
Rx	Entrada	1	Informa que a chave tem dado para receber
credit_out	Saída	1	Informa que a chave pode receber dados

A lógica de controle engloba o árbitro e a lógica de roteamento, segundo a estrutura interna da chave apresentada na Figura 9 do Capítulo 1. Cada porta corresponde a um canal físico. O método de chaveamento *wormhole* permite que cada canal físico seja multiplexado em

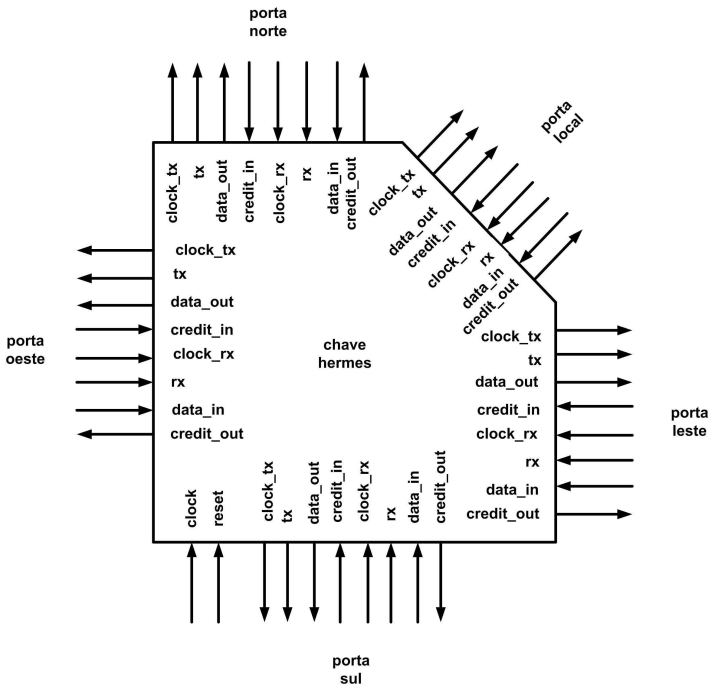


Figura 16: Sinais de interfaceamento externo da chave

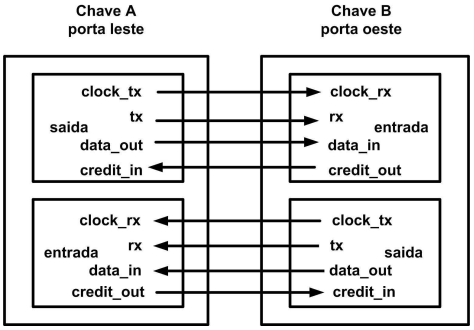


Figura 17: Ligação entre as portas leste e oeste de duas chaves vizinhas

N canais virtuais. Embora esse recurso aumente o desempenho do chaveamento (RIJPKEMA; GOOSSENS; WIELAGE, 2001), os projetistas da chave Hermes optaram por utilizar um único canal lógico para cada canal físico, objetivando reduzir a complexidade e o custo do mesmo.

A Figura 18 exibe os principais módulos que compõem a chave. Cada porta possui uma fila para armazenamento temporário de *flits*. Cada uma das filas da chave (L, O, N, S e Local), ao receber um novo pacote, requisita roteamento ao árbitro ativando o sinal `h`. O árbitro seleciona a requisição de maior prioridade, quando existem requisições simultâneas, e encaminha o pedido de roteamento para a lógica de roteamento ativando o sinal `req_rot`. A lógica de roteamento verifica se é possível atender à solicitação. Sendo possível, a conexão é estabelecida e o árbitro é informado pela ativação do sinal `ack_rot`. Por sua vez, o árbitro

ativa o sinal `ack_h`, informando para a fila que o mesmo pode enviar os *flits* armazenados. Depois que todos os *flits* do pacote forem enviados, a fila ativa o sinal `free`, encerrando a conexão.

2.1.1.1 Lógica de controle

Conforme apresentado na Figura 18, a lógica de controle é constituída de dois módulos: árbitro e lógica de roteamento. Quando uma ou mais portas da chave recebe o *flit* de cabeçalho (*header*) de um pacote (o primeiro *flit*), o árbitro é acionado e, se a requisição de roteamento é atendida, a lógica de roteamento é acionada para conectar o *flit* da porta de entrada selecionada pelo árbitro à porta de saída correta. Cada chave possui um endereço único na rede. Para simplificar o roteamento na rede, esse endereço é expresso de acordo com as coordenadas XY, onde X representa a posição horizontal e Y a posição vertical.

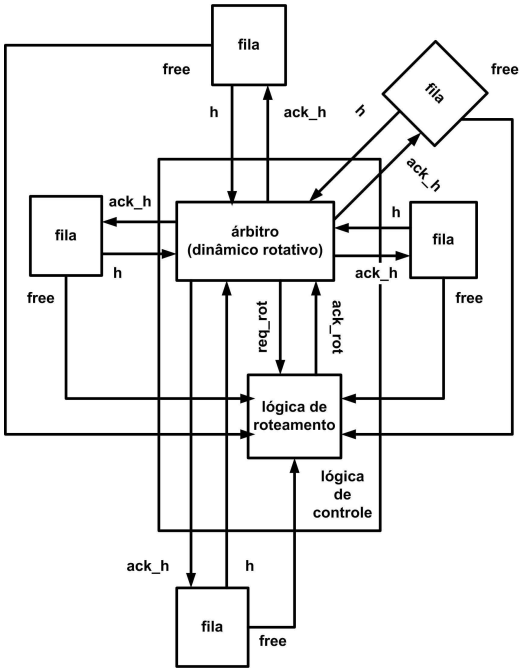


Figura 18: Estrutura interna da lógica de controle

Uma chave pode ser requisitada para estabelecer até 5 conexões simultaneamente. A lógica do árbitro é utilizada para garantir acesso a uma porta de saída quando uma ou mais portas de entrada simultaneamente requerem uma conexão. Um esquema de prioridades dinâmicas rotativas é utilizado. A prioridade de uma porta de entrada é variável e depende da última porta que teve uma requisição de roteamento atendida. Por exemplo, se a porta de entrada Local (índice 4) foi a última a ter requisição de roteamento atendida, a porta de entrada Leste (índice 0) terá a maior prioridade seguida das portas de entrada Oeste (índice

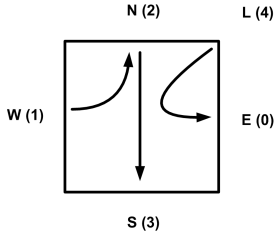
1), Norte (índice 2), Sul (índice 3) e Local (índice 4), que recebem prioridades decrescentes. Esse método garante que todas as requisições de entrada serão atendidas, evitando que ocorra *starvation*.

Após atender uma requisição de roteamento, o árbitro aguarda 4 ciclos de relógio para voltar a atender novas requisições. Esse tempo é utilizado para acionar a lógica de roteamento. Se esta não consegue estabelecer uma conexão, a porta de entrada volta a requisitar roteamento ao árbitro, porém com a menor prioridade.

A lógica de roteamento utiliza um algoritmo de roteamento, denominado XY adaptativo (MELLO, 2003), para determinar por qual porta de saída o pacote deve ser enviado. Esse algoritmo compara o endereço X_lY_l, onde X_l é o endereço horizontal e Y_l o endereço vertical, da chave atual com o endereço X_dY_d, onde X_d é o endereço horizontal e Y_d o endereço vertical, da chave destino do pacote, armazenado no *flit* de cabeçalho. Os *flits* devem ser roteados para a porta Local quando o endereço X_lY_ℓ da chave atual é igual ao endereço X_dY_d do pacote. Se esse não for o caso, o endereço (horizontal) X_d é primeiro comparado ao endereço X_ℓ. Os *flits* serão roteados para a porta Leste quando $X_l < X_d$, para Oeste quando $X_l > X_d$ e, se $X_l = X_d$, o *flit* de cabeçalho já está alinhado horizontalmente. Se esta última condição é a verdadeira, o endereço (vertical) Y_d é comparado ao endereço Y_l. Os *flits* serão roteados para a porta Norte quando $Y_l < Y_d$, para Sul quando $Y_l > Y_d$. Se a porta escolhida estiver ocupada, o *flit* de cabeçalho, bem como todos os *flits* subsequentes do pacote em todas as portas intermediárias, serão bloqueados até que a porta de saída escolhida seja liberada.

Quando o algoritmo de roteamento encontra uma porta livre, a conexão entre a porta de entrada e a porta de saída é estabelecida. Para tal, é utilizada uma tabela de roteamento, consistindo de três vetores: *in*, *out* e *free*, conforme mostra a Figura 19. O vetor *in* conecta uma porta de entrada com uma porta de saída e é preenchido com a porta de saída. O vetor *out* conecta uma porta de saída com uma porta de entrada e é preenchido com a porta de entrada. O vetor *free* serve para alterar o estado da porta de saída, que, no momento, encontra-se livre (1), passando para o estado de ocupado (0). Considere a porta Norte da Figura 19. A porta de saída Norte está ocupada (`free = 0`) e conectada à entrada da porta Oeste (`out = 1`). A porta de entrada Norte está conectada à porta de saída Sul (`in = 3`). A estrutura da tabela de roteamento, mostrada na Figura 19, contém informação redundante das conexões, mas essa organização é útil para melhorar a eficiência do algoritmo de roteamento.

Depois que todos os *flits* do pacote forem transmitidos, a conexão deve ser encerrada. Para isto, a chave possui cinco contadores, um para cada porta de entrada. Esses contadores



	0 - E	1 - W	2 - N	3 - S	4 - L
Free	0	1	0	0	1
In	-	2	3	-	0
Out	4	-	1	2	-

Figura 19: Tabela de roteamento

estão implementados dentro de uma fila e são inicializados quando o segundo *flit* do pacote, que contém o número de *flits* restantes do mesmo, chega à porta de entrada da conexão. Esses contadores são decrementados a cada *flit* enviado com sucesso. Quando o valor do contador chega a zero, a conexão é encerrada e o índice da porta de saída no vetor `free` é liberado.

A máquina de estados da lógica de controle é apresentada na Figura 20 com o seu funcionamento detalhado a seguir:

- O estado S_0 é o estado de inicialização da máquina de estados. Este estado somente é atingido quando o sinal `reset` é ativado.

- O estado S_1 é o estado de espera por requisição de roteamento. Quando o árbitro recebe uma ou mais requisições, o sinal `ask` é ativado, fazendo a máquina de estados avançar para o estado S_2.

- No estado S_2, a porta de entrada que solicitou roteamento é selecionada. Se houver mais de uma, aquela com maior prioridade é a selecionada. Então, a máquina de estados avança para o estado S_3.

- No estado S_3 é realizado o algoritmo de roteamento XY. Se o endereço X_lY_l da chave é igual ao endereço X_dY_d do pacote, a máquina de estados avança para o estado S_4. Caso contrário, se o endereço X_l da chave é diferente do endereço X_d do pacote, a máquina de estados avança para o estado S_5. Caso contrário, se o endereço X_l da chave é igual ao endereço X_d do pacote e o endereço Y_l da chave é diferente do endereço Y_d do pacote, a máquina de estados avança para o estado S_6. Caso contrário, se nenhuma das condições anteriores for satisfeita, a máquina de estados volta para o estado S_1 e os *flits* do pacote são bloqueados até que esse pacote possa ser roteado novamente.

- No estado S_4, é estabelecida a conexão da porta de entrada com a porta Local. Então, a máquina de estados avança para o estado S_7.

- No estado S_5, se o endereço X_l da chave é menor que o endereço X_d do pacote, é estabelecida a conexão da porta de entrada com a porta Leste. Caso contrário, é estabelecida a conexão da porta de entrada com a porta Oeste. Então, a máquina de estados avança para o estado S_7.

- No estado S_6, se o endereço Y_l da chave é menor que o endereço Y_d do pacote, é estabelecida a conexão da porta de entrada com a porta Norte. Caso contrário, é estabelecida a conexão da porta de entrada com a porta Sul. Então, a máquina de estados avança para o estado S_7.

- No estado S_7 a porta selecionada para roteamento desativa o sinal `h`. Então, a máquina de estados volta para o estado S_1.

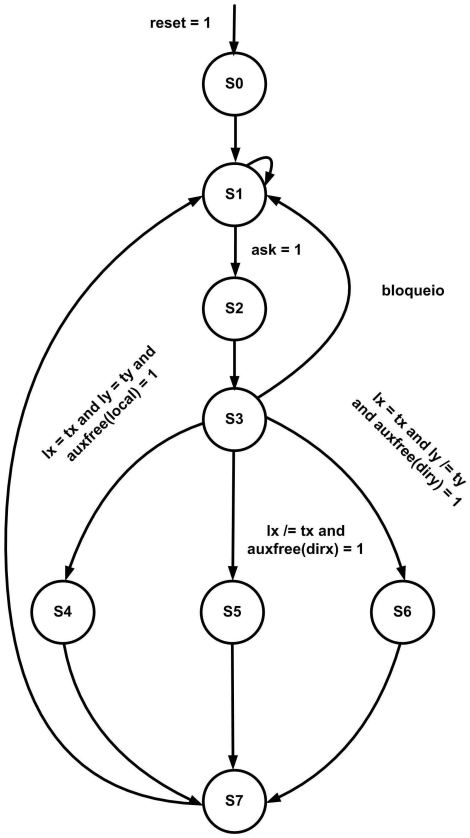


Figura 20: Máquina de estados da lógica de controle

2.1.1.2 Fila

A estrutura interna de uma fila, associada a cada porta da chave, é mostrada na Figura 21. A fila engloba a lógica que controla a inserção de *flits*, a máquina de estados que controla a remoção de *flits* e o contador de *flits* do pacote.

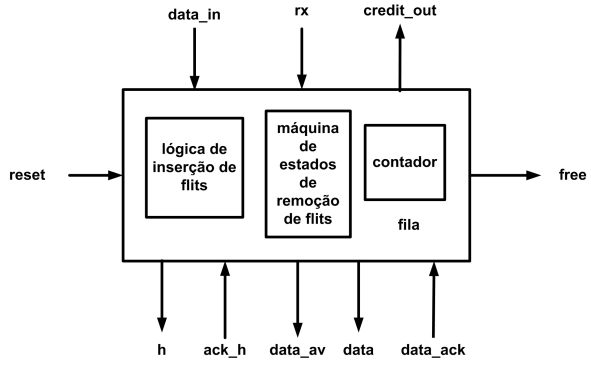


Figura 21: Estrutura interna da fila

Quando o algoritmo de roteamento resulta no bloqueio dos *flits* de um pacote, ocorre uma perda de desempenho em toda a rede de interconexão, porque os *flits* são bloqueados não somente na chave atual, mas em todas as chaves intermediárias. Por exemplo, se as chaves 00 e 01 transmitem ao mesmo tempo um pacote de 8 *flits* destinado à chave 21, o pacote que atingir primeiro o destino tem seu roteamento autorizado e a conexão estabelecida, enquanto o outro deverá ser bloqueado e aguardar até que a conexão seja finalizada. Como a chave 01 está fisicamente mais próxima da chave 21, seu pacote será entregue primeiro e o pacote da chave 00 terá seus *flits* bloqueados nas chaves intermediárias, como é apresentado na Figura 22.

É importante observar que quanto maior for o tamanho da fila, menor será o número de chaves intermediárias afetadas. Como pode ser observado na Figura 23, o pacote com 8 *flits* bloqueia 4 chaves intermediárias, quando a fila possui 2 posições, enquanto o mesmo pacote bloqueia apenas 2 chaves intermediárias, quando a fila possui 4 posições.

As filas funcionam como FIFOs (*First In-First Out*) circulares. Cada fila possui dois ponteiros: `first` e `last`. O ponteiro `first` aponta para a posição da fila onde se encontra o *flit* a ser consumido. O ponteiro `last` aponta para a posição da fila onde deve ser inserido o próximo *flit*.

No momento em que o sinal `reset` é ativado, o parâmetro `tem_espaco` é inicializado com o valor 1 e o ponteiro `last` é com o valor 0. Então, a fila espera pela recepção de *flits*. Quando o sinal `Rx` é ativado, indicando que existe um *flit* na porta de entrada, é verificado se existe espaço na fila para armazená-lo. Se existir espaço na fila, o sinal `credit_out` é ativado, o *flit* recebido é armazenado na posição apontada pelo ponteiro `last` e o mesmo é incrementado. Quando `last` atingir o tamanho da fila, ele é reinicializado com 0.

A máquina de estados para remoção de *flits* da fila é apresentado na Figura 24 e o seu

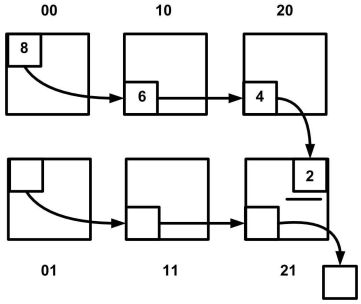


Figura 22: Fila com duas posições

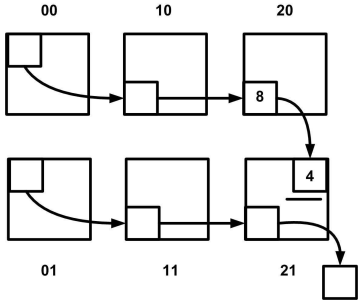


Figura 23: Fila com quatro posições

funcionamento é detalhado a seguir:

- No estado S_0, os sinais `counter_flit`, contador de *flits* do corpo do pacote, `h`, indica requisição de roteamento, e `data_av`, indica a existência de *flit* a ser transmitido, são inicializados com 0. Se existir algum *flit* na fila, ou seja, os ponteiros `first` e `last` apontarem para posições diferentes, a máquina de estados avança para o estado S_1.

- No estado S_1, é requisitado o roteamento (`h` = 1), uma vez que o *flit* da posição apontada pelo ponteiro `first`, quando a máquina encontra-se nesse estado, é sempre o cabeçalho do pacote. A máquina permanece nesse estado até que receba a confirmação do roteamento (`ack_h` = 1). Então, o sinal `h` recebe o valor 0 e a máquina de estados avança para S_2.

- No estado S_2, é indicado que existe um *flit* a ser transmitido (`data_av` = 1). A máquina de estados permanece nesse estado até receber a confirmação da transmissão (`data_ack` = 1). Então, o ponteiro `first` aponta para o segundo *flit* do pacote e a máquina de estados avança para o estado S_3.

- No estado S_3, é indicado que existe um *flit* a ser transmitido (`data_av` = 1). Quando é recebida a confirmação da transmissão (`data_ack` = 1), é verificado o valor do sinal `counter_flit`. Se `counter_flit` é igual a 0, ele é, então, inicializado com o valor do

flit, que corresponde ao número de *flits* do corpo do pacote. Caso `counter_flit` seja diferente de 0 e de 1, ele é decrementado e a máquina de estados permanece nesse estado, enviando o próximo *flit* do pacote. Caso `counter_flit` seja igual a 1, a máquina de estados avança para o estado S_0.

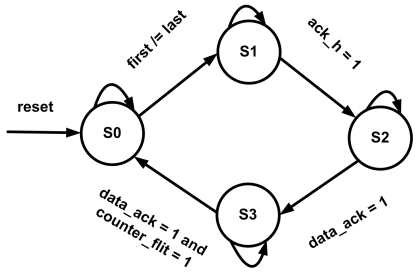


Figura 24: Máquina de estados de remoção de *flits* da fila

2.1.2 Conexões entre as chaves e os recursos

Os recursos são conectados às chaves, sendo que a porta de conexão do recurso e a porta de conexão correspondente da chave, ou seja a sua porta Local, devem ter o mesmo tamanho. Conforme visto na Seção 2.1.1, o tamanho das portas da chave é igual ao tamanho do *flit*. Ou seja, aumentar ou diminuir o tamanho do *flit* equivale a aumentar ou diminuir o tamanho das portas da chave. Isto implica em mudanças no modelo da chave, da rede intrachip, do processador e, até mesmo, no código fonte do microkernel. Para que a chave possa ser conectada a recursos com tamanho de porta variável, optou-se por desenvolver um mecanismo que tornasse o tamanho do *flit* da chave facilmente parametrizável. Esse mecanismo será descrito na Seção 2.3.1 e 3.3.1 (do Capítulo 3).

2.1.3 Interconexões entre as chaves da rede intrachip

As chaves são interconectadas manualmente, sendo difícil a modificação do tamanho da rede intrachip. Aumentar ou diminuir o tamanho da rede intrachip é bastante trabalhoso, sujeito a erros e envolve mudanças no modelo da chave, da rede intrachip e do processador. Por esse motivo, para que o algoritmo genético paralelo pudesse ser executado em uma rede intrachip de tamanho maior que 3×2 (tamanho original da plataforma), optou-se por desenvolver um mecanismo que tornasse o tamanho da rede intrachip facilmente parametrizável. Esse mecanismo será descrito na Seção 2.3.2

2.2 O Processador Plasma

O Plasma é um processador RISC de 32 *bits* capaz de executar a maioria das instruções da arquitetura MIPS I (PATTERSON; HENNESSY, 1998). Seu modelo VHDL é aberto e está disponível no sitio do *OpenCores* (OPENCORES..., 2006). O Plasma possui um *pipeline* de instruções de três estágios: busca, decodificação e execução. A organização de memória do Plasma é Von Neumann e não Harvard, como definido originalmente na arquitetura MIPS I. O Plasma oferece suporte ao compilador C (GCC) e tratamento de interrupções. Para este trabalho, foi utilizada a versão do Plasma modificada pelo grupo GAPH.

Essa versão modificada do Plasma, apresentada na Figura 25, contém 5 componentes principais. O MLite_CPU (*Mips Lite Central Processor Unit*) é o processador Plasma propriamente dito. O Paginador de Memória divide o espaço de endereçamento da memória do Plasma em páginas, permitindo a execução de várias tarefas por um único processador. O Controlador de Interrupção faz a gerência das interrupções de *hardware*. A Interface de Rede faz a interface entre o processador e a rede. O Controlador de Acesso Direto à Memória DMA (*Direct Memory Access*) transfere para a memória dos processadores escravos o código objeto das tarefas localizadas no processador mestre. Além desses, há uma memória RAM para dados e programas, interface serial UART (*Universal Asynchronous Receiver/Transmitter*), contador de tempo para interrupção (counter_reg) e contador de tempo do sistema (tick_counter). Este último tem como função acumular os ciclos de *clock* durante a execução do sistema e pode ser lido pelo microkernel ou pela aplicação através de uma chamada de sistema, conforme explicado na Seção 3.2.5 (do Capítulo 3).

2.2.1 Paginador de memória

A arquitetura MIPS define suporte para até 4 coprocessadores (SWEETMAN, 2006) CP0, CP1, CP2 e CP3. CP0 é o chamado coprocessador de controle do sistema, responsável pela paginação da memória, tratamento de interrupções e tratamento de exceções. CP1 é o coprocessador de ponto flutuante. CP2 é utilizado para adicionar extensões personalizadas para a arquitetura MIPS. CP3 é utilizado para implementar instruções de ponto flutuante para MIPS32/64. O coprocessador CP0 não tem existência independente, de forma que todos os processadores MIPS comerciais o implementam.

O Plasma só implementa, do CP0, os registradores `$10`, que contém a página da memória de uma tarefa, `$12`, que é utilizado somente para habilitar ou desabilitar as interrupções, e

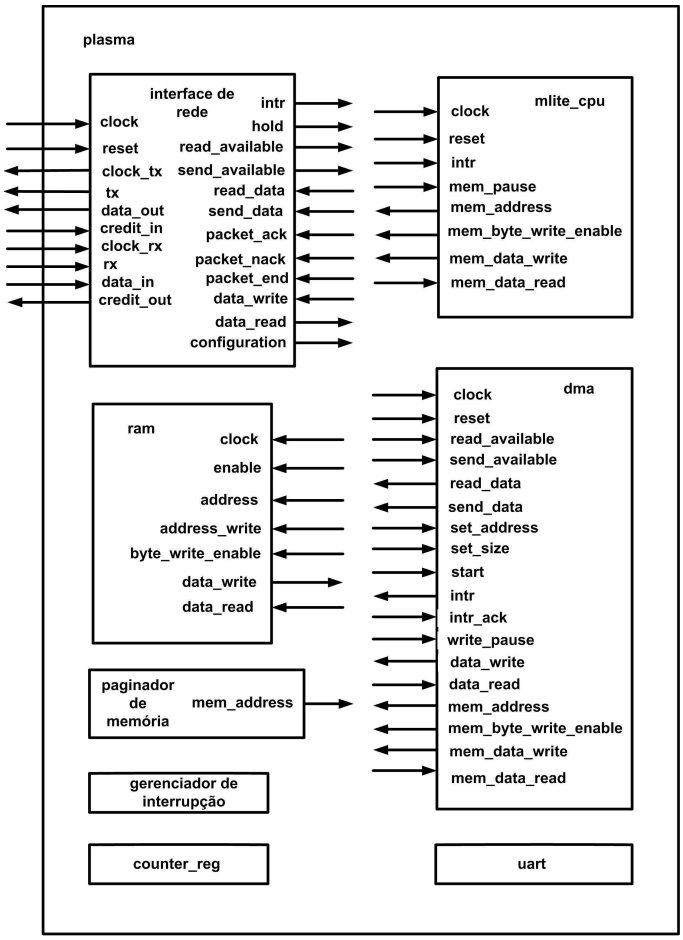


Figura 25: Processador Plasma

`$14`, que contém o endereço da instrução que causou uma exceção. Os outros coprocessadores não são implementados no Plasma.

A paginação de memória facilita enormemente a execução de múltiplas tarefas em uma mesma CPU. De acordo com a Figura 26, a organização de memória do Plasma, modificado pelo grupo GAPH, é dividida em quatro páginas de 16 KB, onde os 2 *bits* mais significativos (14 e 15) do endereço indicam a página da memória, onde uma aplicação será executada, e os restantes (0 a 13) indicam o deslocamento (*offset*) dentro da mesma.

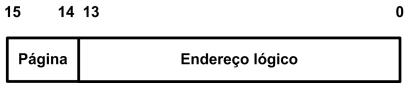


Figura 26: Espaço de endereçamento do processador Plasma

A configuração de página é realizada pela instrução `MTC0 $27, $10`. Nesta instrução, a página da memória de uma tarefa, que está armazenada no registrador `$27`, é carregada para

o registrador `$10` do CP0. O controlador de memória gera um endereço que não contém a página, mas o deslocamento dentro da mesma. Dessa forma, o endereço físico (`mem_address`), gerado pela CPU, é composto pela concatenação do endereço lógico, fornecido pelo controlador de memória, com a informação de página fornecida pelo registrador `$10` do CP0, como mostra a Figura 27.

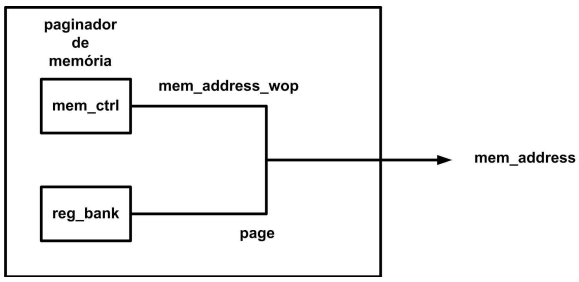


Figura 27: Geração do endereço físico

O mecanismo de paginação oferece segurança no acesso à memória, evitando violação de endereços. Isso significa que uma tarefa residente na página P_x jamais conseguirá acessar um endereço na página P_y (sendo $x \neq y$), uma vez que todo endereço lógico gerado por essa tarefa será concatenado com a página P_x.

A modificação do tamanho de página, como também do número de páginas é bastante trabalhosa, sujeita a erros e envolve mudanças no modelo do processador e no código fonte do microkernel. Pelo fato do tamanho do código objeto do algoritmo genético paralelo ser maior que 16 KB, optou-se por desenvolver um mecanismo que tornasse o tamanho página e o número de páginas facilmente parametrizável. Esse mecanismo será descrito na Seção 2.3.3 e Seção 3.3.2 (do Capítulo 3).

2.2.2 Controlador de interrupção

O Plasma pode ser interrompido via *hardware* ou via *software*. As interrupções de *hardware* são gerenciadas pelo controlador de interrupção, podendo ser:

- Do contador de tempo para interrupção, informando que o *timeslice* de uma tarefa acabou e uma nova tarefa deve ser escalonada.
- Da interface de rede, informando que um pacote veio da rede intrachip.
- Do controlador de DMA, informando que o código objeto de uma tarefa já se encontra na memória e que a mesma pode ser executada.

As interrupções são habilitadas por meio da instrução `MTC0 $1, $12`. Nessa instrução, o valor 1, armazenado no registrador `$10`, é carregado no registrador `$12` do CP0. Além desse registrador, o controlador de interrupção possui dois registradores mapeados em memória, que são utilizados para a comunicação com o microkernel. Esses registradores são mostrados na Tabela 2 e o registrador de estado é mostrado na Figura 28.

Tabela 2: Registradores mapeados em memória do controlador de interrupção

Registrador	Endereço	Descrição
`IRQ_mask`	0x20000010	Este registrador contém a máscara de interrupções
`IRQ_status`	0x20000020	Este registrador contém o estado das interrupções

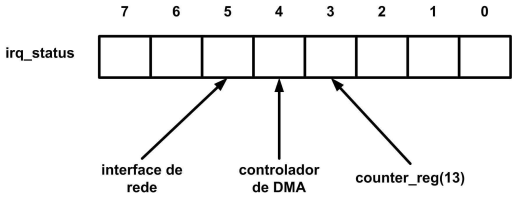


Figura 28: Registador de estado das interrupções

Cada interrupção de *hardware* contém uma máscara e uma função de tratamento definidas no microkernel. Essas interrupções, máscaras e funções de tratamento são mostradas na Tabela 3. A função *OS_InterruptRegister(máscara,função_tratamento)* é utilizada para armazenar, em um vetor de ponteiros para função (ISR), o endereço da função responsável pelo tratamento de determinada interrupção, associando a posição nesse vetor com a máscara. Por exemplo, quando ocorrer uma interrupção proveniente do contador de *timeslice*, cuja máscara é 00001000, a função *Scheduler()* será executada. A função *OS_InterruptMaskSet(máscara)* é utilizada para configurar a máscara de interrupções, inicializando o registrador `IRQ_mask`.

Tabela 3: Máscaras das interrupções

Interrupção	Máscara	Função de tratamento
Contador de *timeslice*	0b00001000	*Scheduler()*
Controlador de DMA	0b00010000	*DMA_Handler()*
Interface de rede	0b00100000	*DRV_Handler()*

O microkernel do processador mestre inicializa o registrador `IRQ_mask` com o valor 00100000, habilitando somente a interrupção proveniente da interface de rede. Já o microkernel do processador escravo inicializa esse registrador com o valor 00111000 habilitando todas as interrupções. O processador é interrompido quando a operação AND, entre o conteúdo dos

registradores `IRQ_mask` e `IRQ_status`, retorna um resultado diferente de 0. Quando esse evento ocorre, o fluxo de execução desvia para o endereço 0x3c, onde a interrupção é tratada. Enquanto uma interrupção é tratada, não podem ocorrer novas interrupções. Desta forma, as interrupções são desabilitadas no início do tratamento (via *hardware*) e reabilitadas no final (via *software*).

2.2.3 Interface de rede

A interface de rede é utilizada para realizar a conexão entre o processador e a rede intrachip. Ela é responsável pelo envio de pacotes para a rede, segmentando os dados em *flits* e o recebimento dos pacotes da rede, armazenando-os em uma fila. Quando existir um pacote completo na fila ou quando o mesmo estiver cheio, a interface de rede interromperá o processador para que este receba os dados. Esta interface também é responsável pelo repasse do código objeto das tarefas, recebido da rede, para a memória, através do DMA. Além disso, informar ao microkernel, executado pelo processador, qual é o seu endereço na rede (`netaddress`).

A modificação do tamanho do *flit* da interface de rede, como também do tamanho da sua fila, é bastante trabalhosa, sujeita a erros e envolve mudanças no modelo do processador e no código fonte do microkernel. Neste trabalho, optou-se por desenvolver um mecanismo que tornasse o tamanho do *flit* e da fila da interface de rede facilmente parametrizáveis. Esse mecanismo será descrito na Seção 2.3.2 e na Seção 3.3.1 (do Capítulo 3). A Figura 29 mostra os sinais de interfaceamento da interface de rede e a Tabela 4 descreve cada um desses sinais.

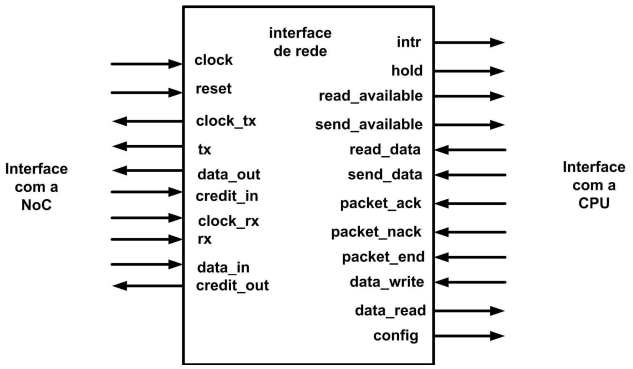


Figura 29: Sinais de interfaceamento da interface de rede

A interface de rede possui registradores mapeados em memória, que são utilizados para a comunicação com os *drivers* do microkernel. Esses *drivers*, que fazem parte da infra-estrutura de *software* da plataforma, como explicado na Seção 3.1.6 e Seção 3.2.6 (do Capítulo 3), além de montar os pacotes, os enviam para a interface de rede e os recebem desta. Esses registradores são descritos na Tabela 5.

Tabela 4: Sinais de interfaceamento da interface de rede

Sinal	Tipo	# de bits	Descrição
clock	Entrada	1	Sinal de *clock* da interface de rede
reset	Entrada	1	Sinal de *reset* da interface de rede
clock_tx	Saída	1	*Clock* da porta de saída que sincroniza a transmissão de dados
data_out	Saída	16 ou 32	Saída de dados
Tx	Saída	1	Informa que a interface de rede tem dado para enviar
credit_in	Entrada	1	Informa que a interface de rede pode enviar dados
clock_Rx	Entrada	1	Clock da porta de entrada que sincroniza a recepção de dados
data_in	Entrada	16 ou 32	Entrada de dados
Rx	Entrada	1	Informa que a interface de rede tem dado para ser recebido
credit_out	Saída	1	Informa que a interface de rede pode receber dados
intr	Saída	1	Interrompe o processador
hold	Saída	1	Informa que o processador pode começar a executar
send_av	Saída	1	Informa que o processador pode enviar um dado para a rede intrachip
read_av	Saída	1	Informa que o processador pode receber um dado que vem da rede intrachip
send_data	Entrada	1	Informa que o dado em data_write deve ser enviado
read_data	Entrada	1	Informa que o dado em data_read foi lido
packet_ack	Entrada	1	Informa que o processador pode receber o pacote
packet_nack	Entrada	1	Informa que o processador não pode receber o pacote
packet_end	Entrada	1	Informa que o processador terminou de receber os dados de um pacote
data_write	Entrada	32	Dado recebido do processador a ser enviado para a rede intrachip
data_read	Saída	32	Dado recebido da rede e repassado para o processador
configuration	Saída	32	Informa ao processador o seu endereço de rede

Um pacote que trafega pela rede tem o seguinte formato `<target><size><payload>`, onde `target` é o *flit* que contém o destino do pacote, `size` é o *flit* que contém o tamanho do pacote e `payload` contém vários `flits` com o conteúdo do pacote. Os campos `target` e `size` só utilizam os 8 primeiros *bits* do *flit* e os *bits* restantes são zerados. O campo `payload` é constituído por `<service><service_parameters>`, onde `service` é o *flit* que contém o serviço solicitado e `service_parameters` contém vários *flits* com os parâmetros desse serviço. O serviço será executado pelo microkernel depois de ter recebido o pacote. Os serviços que um pacote pode carregar são descritos na Tabela 6. Os parâmetros desses serviços são descritos logo em seguida:

Tabela 5: Registradores mapeados em memória para a comunicação entre *drivers* e interface de rede

Registrador	Endereço	Descrição
status_read	0x20000100	*Driver* lê este registrador para verificar se existe dado que veio da rede intrachip
status_send	0x20000110	*Driver* lê este registrador para verificar se pode enviar um dado para a rede intrachip
read_data (sinal `data_read`)	0x20000120	*Driver* lê neste registrador o dado que veio da rede intrachip
write_data (sinal `data_write`)	0x20000130	*Driver* escreve neste registrador o dado que será enviado para rede intrachip
configuation (sinal `configuation`)	0x20000140	Microkernel lê neste registrador o endereço de rede do processador (netaddress)
packet_ack (sinal `packet_ack`)	0x20000150	*Driver* escreve neste registrador para informar à interface de rede que pode receber o pacote
packet_nack (sinal `packet_nack`)	0x20000160	*Driver* escreve neste registrador para informar à interface de rede que não pode receber o pacote
packet_end (sinal `packet_end`)	0x20000170	*Driver* escreve neste registrador para informar à interface de rede que recebeu o pacote

Tabela 6: Descrição dos serviços que um pacote carrega

Serviço	Código	Descrição
REQUEST_MESSAGE	0x10	Requisição de uma mensagem
DELIVER_MESSAGE	0x20	Entrega de uma mensagem previamente solicitada
NO_MESSAGE	0x30	Aviso de que a mensagem solicitada não existe
TASK_ALLOCATION	0x40	Alocação de tarefa por meio do DMA
ALLOCATED_TASK	0x50	Aviso de que uma nova tarefa está alocada no sistema
REQUEST_TASK	0x60	Requisição de uma tarefa
TERMINATED_TASK	0x70	Aviso de que uma tarefa terminou a sua execução
DEALLOCATED_TASK	0x80	Aviso de que uma tarefa pode ser desalocada
FINISHED_ALLOCATION	0x90	Aviso de que a alocação inicial das tarefas foi concluída

```
REQUEST_MESSAGE  <source_slave_processor>
                 <message_target> <message_source>,
```

onde `source_slave_processor` é o *flit* que contém o processador escravo que está requisitando a mensagem, `message_target` é o *flit* que contém o identificador da tarefa que gerou o pedido de mensagem e `message_source` é o *flit* que contém o identificador da tarefa que enviará a mensagem.

```
DELIVER_MESSAGE  <source_slave_processor>
                 <message_target> <message_source>
                 <message_size> <message>},
```

onde `source_slave_processor` é o *flit* que contém o processador escravo que está entregando a mensagem, `message_target` é o *flit* que contém o identificador da tarefa que gerou o pedido de mensagem, `message_source` é o *flit* que contém o identificador da tarefa que enviará a mensagem, `message_size` é o *flit* que contém o tamanho da mensagem em palavras de 32 *bits* e *message* contém vários *flits* com a mensagem.

```
NO_MESSAGE  <source_slave_processor>
            <message_target> <message_source>,
```

onde `source_slave_processor` é o *flit* que contém o processador escravo que está entregando a resposta, `message_target` é o *flit* que contém o identificador da tarefa que gerou o pedido de mensagem e `message_source` é o *flit* que contém o identificador da tarefa que receberá mensagem.

```
TASK_ALLOCATION  <task_id> <code_size> <code>,
```

onde `task_id` é o *flit* que contém o identificador da tarefa que deve ser alocada, `code_size` é o *flit* que contém o tamanho, em palavras de 32 *bits*, do código objeto da tarefa e `code` contém vários *flits* com o código objeto.

```
ALLOCATED_TASK  <processor> <task_id>,
```

onde `processor` é o *flit* que contém o processador escravo e `task_id` é o *flit* que contém a tarefa que foi alocada a esse processador.

```
REQUEST_TASK  <master_processor>
              <slave_processor> <task_id>,
```

onde `master_processor` é o *flit* que contém o processador mestre, `slave_processor` é o *flit* que contém o processador escravo que está requisitando uma tarefa e `task_id` é o *flit* que contém o identificador da tarefa requisitada.

```
TERMINATED_TASK  <master_processor>
                 <slave_processor> <task_id>,
```

onde `master_processor` é o *flit* que contém o processador mestre, `slave_processor` é o *flit* que contém o processador escravo onde a tarefa está alocada e `task_id` é o *flit* que contém o identificador da tarefa que terminou a execução.

```
DEALLOCATED_TASK  <task_id>,
```

onde `task_id` é *flit* que contém o identificador da tarefa que está sendo liberada.

```
FINISHED_ALLOCATION
```

2.2.3.1 Envio de pacotes para a rede intrachip

O dado que a interface de rede recebe do processador é armazenado em um registrador de 32 *bits* (`buffer_out`). Se a chave e a interface de rede estão configuradas para enviar e receber dados em 32 *bits*, o dado enviado para a rede intrachip será o conteúdo desse registrador. Caso contrário, o dado que será enviado para a rede intrachip será a metade mais significativa desse registrador. Para enviar a metade menos significativa, a interface de rede a desloca para a metade mais significativa para, então, enviá-la.

A Figura 30 mostra um pacote de requisição de mensagem enviado em 32 *bits* e a Figura 31 mostra o mesmo pacote enviado em 16 *bits*. Convém observar aqui que o primeiro *flit* (`target`) e o segundo *flit* (`size`) não são segmentados.

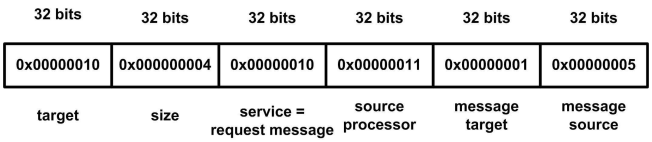


Figura 30: Pacote não segmentado

A Figura 32 mostra a máquina de estados de envio da interface de rede. A máquina de estados inicia no estado `Starget`. Nesse estado, é enviado para a rede intrachip o *flit* que contém o destino do pacote. Se o processador deseja enviar mais um *flit* (`send_data = 1`) e

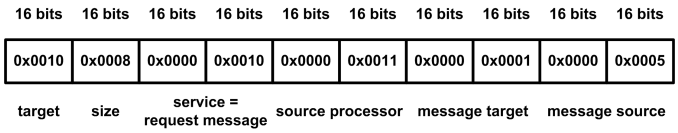


Figura 31: Pacote segmentado em flits de 16 bits

a rede intrachip pode recebê-lo (`waiting_out = 0`), o estado avança para `Ssize`. No estado `Ssize`, o *flit* que contém o tamanho do pacote é enviado para a rede intrachip e armazenado na variável `size_out`. Se o processador deseja enviar mais um *flit* (`send_data = 1`) e a rede intrachip pode recebê-lo (`waiting_out = 0`), o estado avança para `Spayload`. Em `Spayload` o *flit* é enviado e o parâmetro `size_out` é decrementada. A máquina de estados permanece em `Spayload` até que todos os *flits* do pacote sejam enviados. Então, o valor da variável `size_out` será 0 e a máquina de estados volta para `Starget`.

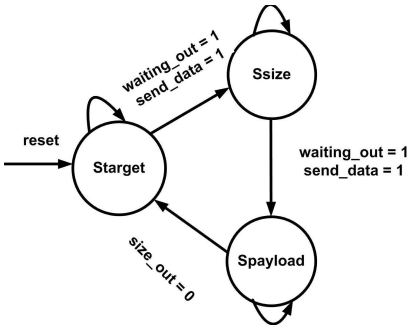


Figura 32: Máquina de estados para o envio de pacotes para a rede intrachip

2.2.3.2 Recepção de pacotes da rede intrachip

Os dados que a interface de rede recebe da rede intrachip são armazenados em uma fila. Essa fila possui dois ponteiros `first` e `last`. O ponteiro `first` aponta para a posição da fila onde se encontra o dado a ser lido pelo processador. Já o ponteiro `last` aponta para a posição onde o dado recebido da rede intrachip deve ser escrito.

A Figura 33 mostra a máquina de estados de recebimento da interface de rede. A máquina de estados inicia no estado `Swait`. Nesse estado, a rede intrachip espera receber o *flit* que contém o destino do pacote. Se existe mais um *flit* para receber (`Rx = 1`) e existe espaço na fila da interface de rede (`tem_espaco = 1`), o estado avança para `Ssize`. No estado `Ssize`, o *flit* que contém o tamanho do pacote é armazenado na fila e no parâmetro `size_in`. Se existe mais um *flit* para receber (`Rx = 1`) e existe espaço na fila da interface de rede (`tem_espaco = 1`), o estado avança para `Swasting`. No estado `Swasting` a variável `size_in` é decrementada, o

ponteiro `last` é incrementado e o *flit* é armazenado na fila. A máquina de estados permanece no estado Spayload até que todos os *flits* do pacote sejam recebidos e armazenados na fila. Então, o valor da variável `size_out` será 1 e o estado avança para `Sending`. No estado `Sending`, o ponteiro last é incrementado e o estado volta para `Starget`.

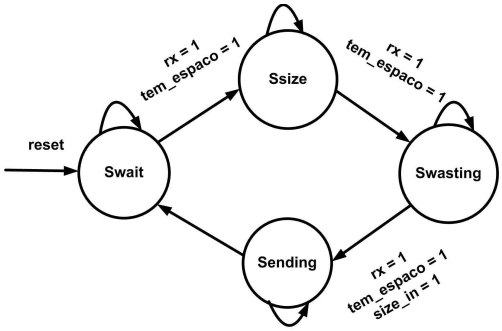


Figura 33: Máquina de estados para o recebimento de pacotes da rede intrachip

2.2.4 Controlador de DMA

O controlador de DMA é utilizado para transferir o código objeto de uma tarefa, recebido pela interface de rede, para a memória do processador. A Figura 34 mostra os sinais de interfaceamento do controlador de DMA e a Tabela 7 descreve cada um desses sinais.

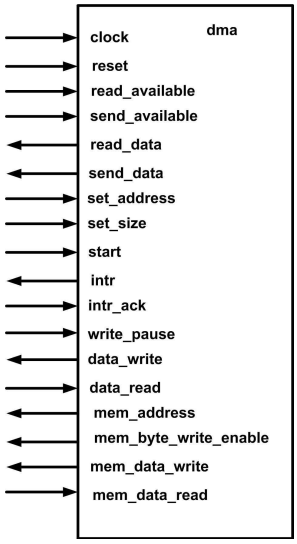


Figura 34: Sinais de interfaceamento do controlador de DMA

O controlador de DMA possui registradores mapeados em memória, que são utilizados para a comunicação com o microkernel. Esses registradores são descritos na Tabela 8.

Tabela 7: Sinais de interfaceamento do controlador de DMA

Sinal	Tipo	# de bits	Descrição
clock	Entrada	1	Sinal de clock do controlador de DMA
reset	Entrada	1	Sinal de reset do controlador de DMA
set_address	Entrada	1	Informa o endereço de memória a partir do qual é o código objeto deve ser transferido
set_size	Entrada	1	Informa o tamanho do código
start	Entrada	1	Informa que a transferência será iniciada
read_av	Entrada	1	Informa que existe um dado disponível para leitura na interface de rede
read_data	Saída	1	Informa a interface de rede que recebeu um dado
send_av	Entrada	1	Informa que a interface de rede pode receber um dado para envio
send_data	Saída	1	Informa para a interface de rede que existe um dado para envio
intr	Saída	1	Interrompe o processador para avisar que a transferência terminou
intr_ack	Entrada	1	Informa que o processador já reconheceu a interrupção
write_pause	Entrada	1	Informa que não pode escrever na memória
data_write	Saída	32	Informa o dado recebido da memória que deve ser gravado na interface de rede
data_read	Entrada	32	Informa o dado recebido da interface de rede que deve ser gravado na memória
mem_address	Saída	32	Informa a memória onde deve ser escrito o dado
mem_data_write	Saída	32	Grava dado na memória
mem_data_read	Entrada	32	Lê dado da memória
mem_write_enable	Saída	3	Habilita a escrita

Tabela 8: Registradores mapeados em memória para a comunicação entre o Microkernel e o controlador de DMA

Registrador	Endereço	Descrição
set_dma_size (sinal `set_dma_size`)	0x20000200	Microkernel escreve nesse registrador o tamanho do código objeto
set_dma_address (sinal `set_dma_address`)	0x20000210	Microkernel escreve nesse registrador o endereço de início da transferência
start_dma (sinal `start_dma`)	0x20000220	Microkernel escreve nesse registrador para iniciar a transferência
dma_ack (sinal `dma_ack`)	0x20000220	Microkernel escreve nesse registrador para informar que a interrupção foi aceita

A interface de rede interrompe o processador, informando que chegou um pacote. Em seguida, O microkernel interpreta esse pedido de interrupção como uma nova tarefa a ser alocada, obtém o identificador da tarefa e o tamanho do código objeto, e verifica a disponibilidade de página livre na memória. Depois disso, o microkernel informa ao controlador de DMA o endereço de início de transferência e o tamanho do código objeto. Então, o controlador de DMA acessa a interface de rede para ler o código objeto e escrevê-lo na memória. Quando o DMA termina de armazenar o código na memória, ele interrompe o processador para avisar que uma nova tarefa está na memória. Finalmente, o microkernel inicializa a tarefa.

A Figura 35 apresenta a máquina de estados do controlador de DMA. A máquina de estados inicia no estado `Swait`. Nesse estado, são conhecidos o endereço de início de transferência e o tamanho do código que é armazenado no parâmetro `size`. Ainda nesse estado, interrupção é desabilitada como resultado do reconhecimento por parte do processador. Se o processador ativar o sinal `start` (`start` = 1), o estado avança para `Scopy`. Nesse estado, cada dado do código objeto é buscado na interface de rede e gravado na memória do processador. Para cada dado recebido, o parâmetro `size` é decrementado. Quando o valor de `size` for 0, o estado avança para `Send`, onde a escrita na memória é desabilitada, o processador é interrompido e o estado volta para `Swait`.

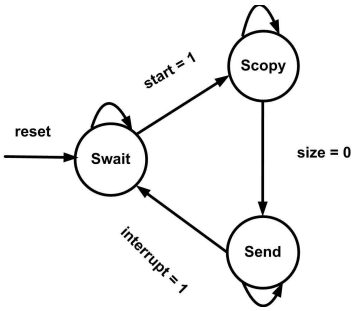


Figura 35: Máquina de estados do controlador de DMA

2.3 Melhorias no Modelo da Plataforma

A plataforma HMPS possui limitações de *hardware* e de *software* que levaram a modificações na mesma para que o algoritmo genético paralelo, a ser apresentado no Capítulo 5, pudesse ser compilado e executado. Foram realizadas modificações no modelo da chave, da rede intrachip, do processador Plasma e no código fonte do microkernel. As mudanças no modelo da plataforma são apresentadas a seguir.

2.3.1 Parametrização do tamanho do flit da chave

A primeira mudança no modelo da plataforma é o desenvolvimento de um mecanismo que permite a fácil parametrização do tamanho do *flit* da chave. A primeira limitação encontrada na plataforma HMPS original é que o tamanho do *flit* é de 16 bits e não pode ser modificado. Para permitir a interconexão de recursos, cujo tamanho da porta de conexão correspondente é diferente do tamanho da porta de conexão da chave, foi necessário modificar o modelo da chave, especificamente a lógica de controle. O tamanho do *flit* influencia diretamente na velocidade do tráfego da rede, como também na área consumida pelo MPSoC. Em relação à velocidade do tráfego da rede, quanto maior for o tamanho do *flit*, maior será a velocidade do tráfego, devido ao fato que menos *flits* terão que ser enviados e recebidos. Em relação à área consumida, quanto maior o tamanho do *flit*, maior será o espaço consumido, uma vez que será necessário um número maior de conexões. De acordo com (PAMUNUWA et al., 2004), a densidade de integração disponível pela tecnologia atual deverá ser capaz de acomodar um número grande de recursos, por exemplo, uma rede malha 5×5 ou maior com tamanho de *flit* igual a 128 *bits*. Convém mencionar aqui que a chave utilizada pela rede intrachip Nostrum (MILLBERG et al., 2004) permite um *flit* desse tamanho. O tamanho do *flit* da chave é definido pelo parâmetro `TAM_FLIT` no modelo.

2.3.2 Parametrização do tamanho da rede intrachip

A segunda mudança no modelo da plataforma é o desenvolvimento de um mecanismo que permite a fácil parametrização do tamanho da rede intrachip. A segunda limitação encontrada na plataforma HMPS original é que ela possui somente 6 processadores organizados em uma rede malha 3×2. Pelo fato de que o processador mestre só realiza alocação de tarefas para os processadores escravos, sobram 5 processadores para executar o algoritmo genético paralelo. Para executar o esse algoritmo com 8 (malha 3×3) ou 15 processadores (malha 4×4), é necessário acrescentar manualmente as chaves no modelo da plataforma. Além disso, a chave assume um dentre nove modelos diferentes, conforme mostrado na Figura 10 do Capítulo 1, dependendo da posição em que é conectada na rede.

Para resolver esse problema, o modelo da chave foi modificado com a finalidade de criar uma chave genérica que englobasse esses 9 modelos. Este modelo é mostrado no Apêndice C. Após a criação da chave genérica, foi possível modificar o modelo das interconexões das chaves, a fim de possibilitar a fácil parametrização do tamanho da rede intrachip. Este modelo é mostrado no Apêndice D. O modelo do sistema HMPS é apresentado no Apêndice E.

As conexões de borda da rede intrachip que não foram utilizadas precisam ser tratadas. Dos vários meios de tratar as conexões de borda (NILSSON, 2002), decidiu-se por aterrar as entradas não utilizadas das chaves para que pudéssemos utilizar a chave genérica. Se as conexões de borda de saída forem conectadas nas conexões de borda de entrada do lado oposto, e vice-versa, teremos um toroide (ver Seção 1.2.2), o que é interessante, em termos de endereçamento (NILSSON, 2002) e latência, mas a sua realização implica em mudanças no algoritmo de roteamento e produz conexões longas dentro do *chip*, o que não é desejável.

O modelo do processador Plasma também foi modificado com a finalidade de facilitar a parametrização do tamanho da rede intrachip. Para tal, a interface de rede foi modificada. Além disso, agora é possível definir o tamanho do *flit* e da fila da interface de rede.

O tamanho da rede intrachip é definido pelo parâmetro `MAX_X` e `MAX_Y` no arquivo `hermes_package.vhd`, o tamanho do *flit* da interface de rede é definido pelo parâmetro `TAM_NI_FLIT`[1] no arquivo `hermes_package.vhd` e o tamanho da fila da interface de rede é definido pelo parâmetro `TAM_NI_BUFFER` no arquivo `hermes_package.vhd`.

2.3.3 Parametrização do tamanho de página de memória

A terceira mudança no modelo da plataforma é a parametrização do tamanho da página de memória e do número de páginas. A outra limitação encontrada na plataforma original é que o tamanho da página de memória é de 16 KB. Tendo em vista que o tamanho do código objeto do algoritmo genético paralelo é maior do que 16 KB, foi necessário modificar o modelo do processador.

Para resolver esse problema, os registradores `page`, `current_page` e `mem_address` do processador Plasma foram modificados para permitir a parametrização do tamanho de página de memória e do número de páginas. Através do parâmetro `TAM_PAGINA`, no arquivo `hermes_package.vhd`, o tamanho da página é obtido por $2^{\texttt{TAM\_PAGINA}}$ e o número de páginas por $2^{(\texttt{28 - TAM\_PAGINA})}$.

2.3.4 Configuração do sistema embutido multiprocessado HMPS

Os parâmetros do arquivo `hermes_package.vhd`, utilizados para configurar o tamanho do *flit* da chave, da fila da chave, do *flit* da interface de rede, da fila da interface de rede, da rede intrachip e do tamanho de página, são mostrados na Tabela 9.

[1]Na plataforma HMPS, o tamanho do *flit* da interface de rede deve ser igual ao tamanho do *flit* da chave.

Tabela 9: Configuração do tamanho do *flit* da chave, da fila da chave, do *flit* da interface de rede, da fila da interface de rede, da rede intrachip e do tamanho de página

Parâmetro	Descrição	Valores possíveis
TAM_FLIT	Tamanho do *flit* da chave	16 ou 32 bits
TAM_BUFFER	Tamanho da fila da chave	4 a 15 posições
TAM_NI_FLIT	Tamanho do *flit* da interface	16 ou 32 bits de rede
TAM_NI_BUFFER	Tamanho da fila da interface	16 ou 32 posições de rede
MAX_X	Endereço X máximo da rede intrachip (o endereço X mínimo é 0)	1 a 15
MAX_Y	Endereço Y máximo da rede intrachip (o endereço Y mínimo é 0)	1 a 15
TAM_PAGINA	Tamanho de página	14 a 27

2.4 Considerações Finais

Este capítulo apresentou uma descrição da rede intrachip Hermes e do processador Plasma, especificamente a estrutura interna da chave da rede intrachip e dos principais componentes do processador Plasma. Também foram apresentadas as melhorias realizadas no *hardware* da plataforma, a fim de torná-la parametrizável. No capítulo seguinte será descrita a infraestrutura de *software* da plataforma HMPS e as melhorias realizadas no microkernel da mesma.

Capítulo 3

O MICROKERNEL

ESSE capítulo apresenta a infra-estrutura de *software* da plataforma Hermes (*Hermes MultiProcessor System on chip* - HMPS) (WOSZEZENKI, 2007) onde será executado o algoritmo genético paralelo. Esta infra-estrutura consiste, basicamente, do microkernel. O microkernel possui duas versões diferentes: uma para o processador mestre e outra para os processadores escravos. A função do microkernel do processador mestre é a alocação de tarefas nos processadores escravos. As principais funções do microkernel dos processadores escravos são o suporte à execução de múltiplas tarefas e a comunicação entre as mesmas. Na Seção 3.1, é apresentado o microkernel do processador mestre e, em seguida, na Seção 3.2, é apresentado o microkernel do processador escravo. Finalmente, na Seção 3.3, são apresentadas as mudanças no microkernel da plataforma realizadas neste trabalho.

3.1 O Microkernel do Processador Mestre

O processador mestre possui um repositório de tarefas que é responsável pelo armazenamento do código executável de todas as tarefas que devem ser executadas pela plataforma. O processador mestre executa somente a alocação de tarefas dentre os processadores escravos. Existem dois modos de alocação de tarefas, a serem descritas mais adiante: estática e dinâmica.

3.1.1 Repositório de tarefas

O repositório de tarefas está implementado em uma memória RAM, conectada diretamente ao processador mestre por meio dos sinais `address`, que indica o endereço de memória, e `data_read`, que indica o dado presente nesse endereço de memória. O espaço de endereçamento da memória externa começa em `0X10000000H` e termina em `0X1FFFFFFFH`.

A Figura 36 mostra a estrutura do repositório de tarefas, com duas tarefas (t_1 e t_2). Cada tarefa armazenada no repositório possui as seguintes informações: identificador da tarefa (`id`),

tamanho do código objeto (`size`) e endereço inicial do código objeto (`initial_address`). Essas informações estão nos primeiros endereços do repositório, formando um cabeçalho. Depois do cabeçalho, encontram-se os códigos objetos das tarefas.

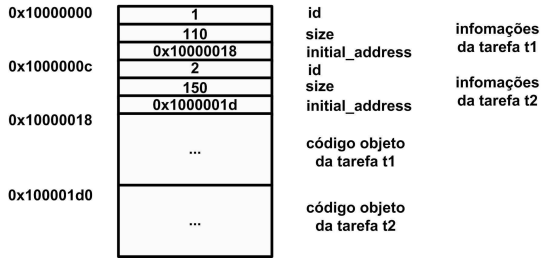


Figura 36: Estrutura do repositório de tarefas

3.1.2 Estrutura do microkernel do processador mestre

A estrutura do microkernel do processador mestre é mostrada na Figura 37. Ela consiste de três níveis de serviços. No nível 1, encontra-se o serviço de inicialização do sistema. No nível 2, encontram-se os *drivers* de comunicação. No nível 3, encontram-se os serviços de tratamento de interrupções e alocação de tarefas. Esses níveis são explicados mais adiante.

Os serviços do microkernel do processador mestre foram implementados parte em linguagem de montagem e parte em linguagem C. Os *drivers* de comunicação e o serviço de tratamento de interrupções foram implementados em linguagem de montagem. O serviço de alocação de tarefas foi implementado em linguagem C.

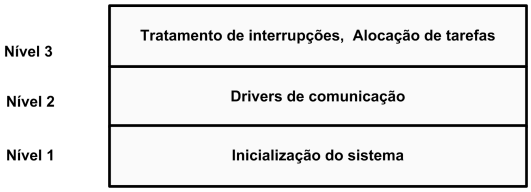


Figura 37: Níveis do microkernel do processador mestre

3.1.3 Estruturas de dados do processador mestre

Os serviços do microkernel do processador mestre utilizam quatro estruturas. A estrutura *TaskLocation*, mostrada na Figura 38, forma uma tabela que contém a associação entre tarefa (`task`) e processador (`processor`), sendo consultada toda vez que ocorre uma comunicação entre tarefas, conforme será explicado na Seção 3.2.8.

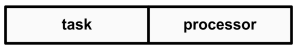


Figura 38: Estrutura *TaskLocation*

A estrutura *TaskPackage*, mostrada na Figura 39, contém o identificador (`id`), o tamanho do código objeto (`size`) e o endereço inicial do código objeto (`*address`) de cada tarefa armazenada no repositório de tarefas, sendo utilizada para gerar o cabeçalho da mesma.

Figura 39: Estrutura *TaskPackage*

A estrutura *processors* consiste de um vetor, mostrado na Figura 40, cujo tamanho é definido pelo parâmetro `MAXPROCESSORS` e indica o endereço dos processadores escravos.

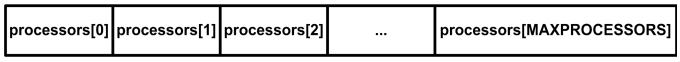


Figura 40: Estrutura *processors*

A estrutura *free_pages* consiste de um vetor, mostrado na Figura 41, cujo tamanho é também definido pelo parâmetro `MAXPROCESSORS` e indica o número de páginas livres que cada processador escravo possui.

3.1.4 Inicialização do microkernel do processador mestre

O microkernel do processador mestre começa a sua execução inicializando `MAXPROCESSORS` com o número de processadores que serão utilizados. Logo após, o microkernel inicializa o vetor *free_pages* com o número de páginas livres que cada processador escravo possui. Em seguida, o microkernel inicializa a tabela de localização de tarefas (*tasks_location*) com as tarefas que serão executadas pela plataforma e a lista de processadores (*processors*) da mesma que serão utilizados para executar essas tarefas. Depois disso, o microkernel executa a função *TasksAllocation*() para alocar as tarefas nos processadores escravos. Então, o microkernel habilita as interrupções vindas da interface de rede. Por último, o microkernel entra no estado (*idle*). Enquanto o microkernel está nesse estado, o processador mestre aguarda por uma interrupção da interface de rede, podendo ser um pedido de requisição de tarefa ou um pedido de finalização de tarefa.

free_pages[0]	free_pages[1]	free_pages[2]	...	free_pages[MAXPROCESSORS]

Figura 41: Estrutura *free_pages*

3.1.5 Tratamento de interrupções

O microkernel do processador mestre só habilita as interrupções vindas da interface de rede. Isto é realizado através da execução das funções *OS_InterruptMaskSet*(`0B00100000`), responsável por configurar a máscara de interrupções, e a função *OS_AsmInterruptEnable*(1), responsável por habilitar as interrupções.

No tratamento de interrupções, inicialmente, o microkernel interrompe o estado (*idle*) e salva o conteúdo dos registradores do processador. Em seguida, a função *DRV_Handler*(), responsável por realizar o tratamento da interrupção proveniente da interface de rede, é executada. Após o tratamento da interrupção, o conteúdo dos registradores é restaurado e o microkernel volta ao estado (*idle*).

3.1.6 Drivers de comunicação

Na Seção 2.2.3, foi explicado que os *drivers* são responsáveis pelo envio e recepção dos pacotes da rede intrachip. Também foram mostrados os diferentes serviços que um pacote pode carregar e o formato do pacote para cada serviço, sendo que para cada serviço é realizado um tratamento diferente. A função *DRV_Handler*(), cujo pseudocódigo é mostrado no Algoritmo 1, é executada pelo processador mestre quando ocorre uma interrupção devido à chegada de pacotes da rede intrachip. Essa função faz chamadas aos *drivers* de comunicação necessários para o tratamento dos pacotes. Todos os *drivers* são escritos em linguagem de montagem. Essa função também possui um ponteiro para a estrutura *TaskPackage*, definida na Seção 3.1.3, sendo utilizado para percorrer o cabeçalho de informações de tarefas no repositório. Para acessar a memória externa, é atribuído a esse ponteiro o endereço `0X10000000H`. Quando chega um pacote da rede intrachip, o *driver DRV_ReadService*(&service,&size) é executado, cuja função é obter o serviço carregado pelo pacote.

Se o serviço for REQUEST_TASK, o *driver DRV_RequestTask*(&*taskID*,&*processor*) é executado, lendo, da interface de rede, o identificador da tarefa requisitada e o processador escravo que a está requisitando. O cabeçalho de informações do repositório de tarefas é percorrido e, quando a tarefa requisitada é encontrada, é procurado um processador escravo com página livre para alocar a tarefa. O *driver DRV_AllocationTask*(*processor*,&*task*[*i*]) é execu-

Algoritmo 1 Função $DRV\_Handler()$ do processador mestre

1: Seja ts o conjunto de tarefas disponíveis no endereço 0X10000000H;
2: $DRV\_ReadService(s, l)$ para obter o identificador do serviço;
3: **Se** s = REQUEST_TASK **Então**
4: $DRV\_RequestTask(t, p)$ para obter o identificador da tarefa;
5: $i := 0$;
6: **Enquanto** $i \leq$ MAXGLOBALTASKS e $t = ts[i]$ **Faça**
7: $DRV\_AllocationTask(p, t)$ para alocar a tarefa a um processador disponível (p);
8: **Insere** (t, p) na tabela de tarefas;
9: **Para** $j := 0 \ldots$ MAXPROCESSORS **Faça**
10: **Se** $processor[j] \neq p$ **Então**
11: $DRV\_Allocated(p, ts[i], processors[j])$ para informar a alocação;
12: **Fim Se**
13: **Fim Para**
14: $i := i + 1$;
15: **Fim Enquanto**
16: **Senão Se** s = TERMINATED_TASK **Então**
17: $DRV\_TerminatedTask(t, p)$ para obter o identificador da tarefa e o processador;
18: **Para** $j := 0 \ldots$ MAXPROCESSORS **Faça**
19: **Se** $processor[j] \neq p$ **Então**
20: $DRV\_DeallocatedTask(t, processors[j])$ para informar a desalocação;
21: **Fim Se**
22: **Fim Para**
23: **Fim Se**

tado para alocar a tarefa, recebendo como parâmetros o processador escravo, onde a tarefa deve ser alocada, e o endereço para a estrutura com as informações da tarefa. Esse *driver* lê as informações do repositório, montando e enviando um pacote com a seguinte estrutura:

```
<target> <size> <service> <task_id> <code_size> <code>
```

onde <target> é *processor*; <size> = 3 + $task[i].size$, se o tamanho do $flit = 32$, ou $6+2\times task[i].size$, se o tamanho do $flit = 16$; <service> é TASK_ALLOCATION; <task_id> é $task[i].id$; <code_size> = $task[i].size$; <code> = conteúdo do intervalo $[task[i].initial\_address, task[i].initial\_address + task[i].size]$.

O número de páginas livres desse processador escravo é decrementado e a tarefa é inserida na tabela de localização de tarefas. Então, $DRV\_AllocatedTask(processor, task[i].id, processors[j])$ é executado para informar a todos os processadores escravos que uma nova tarefa foi alocada, recebendo como parâmetros o processador escravo, onde a tarefa foi alocada, o identificador da tarefa e o processador escravo que está sendo informado. Essa operação é realizada por várias transmissões *unicast*, pelo fato da rede Hermes não possuir serviço de *multicast*. Esse *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <processor> <task_id>
```

onde <target> é $processors[j]$; <size> = 3, se o tamanho do $flit = 32$, ou 6, se o tamanho do $flit = 16$; <service> é ALLOCATED_TASK; <processor> é *processor*; <task_id> é $task[i].id$.

Se o serviço for TERMINATED_TASK, o *driver DRV_TerminatedTask*(&*taskID*, &*processor*) é executado, lendo da interface de rede o identificador da tarefa, cuja execução terminou, e o processador escravo onde ela está alocada. Então, o *driver DRV_DeallocatedTask*(&*taskID*, &$processors[j]$) é executado para informar a todos os processadores escravos que uma tarefa deve ser desalocada. Essa operação é realizada por várias transmissões *unicast*. Esse *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <task_id>
```

onde <target> é $processors[j]$; <size> = 2, se o tamanho do $flit = 32$, ou 4, se o tamanho do $flit = 16$; <service> é DEALLOCATED_TASK; <task_id> é $task[i].id$.

3.1.7 Alocação estática

O microkernel do mestre começa a alocação estática armazenando, no vetor `processors`, os endereços dos processadores escravos e, no vetor `free_pages`, o número de páginas livres que cada escravo possui. Sempre que uma tarefa for alocada a um processador escravo, seu número de páginas livres é decrementado. Em seguida, o microkernel insere as tarefas na tabela `tasks_location`. Então, o microkernel executa a funcão *TasksAllocation*(), cujo pseudocódigo é mostrado no Algoritmo 2. Essa função possui um ponteiro para a estrutura `TasksPackage`, utilizado para percorrer o cabeçalho de informações de tarefas no repositório. Para acessar a memória externa, é atribuído a esse ponteiro o endereço `0X10000000H`.

Quando a função *TasksAllocation*() é executada, a tabela `tasks_location` é percorrida de forma a alocar todas as tarefas nela contidas. Em seguida, o cabeçalho de informações de tarefas no repositório é percorrido e, quando a tarefa a ser alocada é encontrada, o *driver DRV_AllocationTask*(*processor*,&$task[i]$) é executado para alocar a tarefa. Após alocar cada tarefa, o processador mestre deve informar aos outros processadores escravos que uma nova tarefa está alocada no sistema. Então, o *driver DRV_AllocatedTask*($processor, task[i].id, processors[j]$) é executado. Após a alocação da última tarefa, o processador mestre deve informar aos outros processadores escravos que a alocação estática foi concluída. O *driver DRV_FinishedAllocation* ($processors[j]$) é executado, recebendo como parâmetro o processador escravo que está sendo informado. Esta operação é realizada por várias transmissões *unicast*. Este *driver* monta e envia um pacote com os seguintes campos:

Algoritmo 2 Função $TasksAllocation()$

1: Seja ts o conjunto de tarefas disponíveis no endereço 0X10000000H;
2: **Para** i := 0 Até número máximo de tarefas **Faça**
3: **Insere** tarefa e processador na tabela *tasks_location*;
4: **Decremente** o número de páginas livres desse processador;
5: **Fim Para**
6: **Para** $i := 0 \ldots$ MAXGLOBALTASKS **Faça**
7: **Se** uma tarefa foi encontrada na tabela *tasks_location* **Então**
8: **Reserve** um processador (p) no qual a tarefa será alocada;
9: $k := 0$;
10: **Enquanto** $k \leq$ MAXGLOBALTASKS e $tasks\_location[i] = ts[k]$ **Faça**
11: $DRV\_AllocationTask(p, task)$ para alocar essa tarefa no processador reservado;
12: **Para** $j := 0 \ldots$ MAXPROCESSORS **Faça**
13: $DRV\_AllocatedTask(p, ts[k], processors[j])$ para informar a alocação;
14: **Fim Para**
15: $k := k + 1$;
16: **Fim Enquanto**
17: **Fim Se**
18: **Fim Para**

```
<target> <size> <service>
```

onde <target> é $processors[j]$; <size> = 1, se o tamanho do $flit$ = 32, ou 2, se o tamanho do $flit$ = 16; <service> é FINISHED_ALLOCATION.

Após informar aos processadores escravos que a alocação estática de tarefas foi concluída, o processador mestre habilita as interrupções vindas da rede intrachip para aguardar pacotes de comunicação dos escravos, a fim de realizar a alocação dinâmica de tarefas.

3.1.8 Alocação dinâmica

A alocação dinâmica de tarefas consiste no envio de uma tarefa (t_i), pelo processador mestre, a um processador escravo, mediante a requisição de outra tarefa (t_j). A tarefa (t_i) está armazenada no repositório do processador mestre e a tarefa (t_j) está sendo executada em um processador escravo. Essa requisição de alocação é transparente a t_j, sendo requisitada pelo microkernel do processador escravo quando t_j tenta enviar uma mensagem para t_i e ela não está alocada no sistema.

A alocação dinâmica de tarefas, utiliza a função *WritePipe*(&msg, (t_i)), cujo pseudocódigo é mostrado no Algoritmo 3, no lado do processador escravo, e utiliza a função *DRV_Handler*(), já discutida na Seção 3.1.6, no lado do processador mestre.

Quando t_j tenta enviar uma mensagem para t_i, o processador escravo verifica, na tabela de localização de tarefas, se t_i está alocada. Se a tarefa está alocada, a mensagem é

Algoritmo 3 Função $WritePipe(msg, t)$

1: **Se** a tarefa não esta alocada **Então**
2: **Se** a alocação estática foi concluída **Então**
3: **Insere** a tarefa na tabela de requisição de tarefas;
4: $DRV\_RequestTask$(MASTERADDRESS, $netAddress, targetID$) para requisitar uma tarefa;
5: **Senão Se** a alocação estática não foi concluída **Então**
6: **Coloque** a tarefa requisitante em estado WAIT;
7: **Escalone** outra tarefa;
8: **Fim Se**
9: **Senão Se** a tarefa está alocada **Então**
10: $DRV\_DeliverPreviousMessage(processor, t, msg)$ para enviar a mensagem;
11: **Fim Se**

enviada. Caso contrário, é verificado se a alocação estática das tarefas já foi concluída. Se a alocação estática das tarefas foi concluída, o *driver* $DRV\_RequestTask$(MASTERADDRESS, $netAddress, targetID$) é executado recebendo como parâmetros o endereço do processador mestre, o endereço do processador escravo e o identificador da tarefa requisitada. Esse *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <processor> <task_id>
```

onde <target> é MASTERADDRESS; <size> = 3, se o tamanho do $flit = 32$, ou 6, se o tamanho do $flit = 16$; <service> é REQUEST_TASK; <processor> é $netAddress$; <task_id> é t_i.

Se a alocação estática não foi concluída, o estado da tarefa requisitante é colocada em espera e uma nova tarefa deve ser escalonada. Em ambos os casos, a requisição é armazenada em uma tabela de requisições de tarefas.

Enquanto o processo de alocação estática não tiver terminado, tarefas que já foram alocadas podem solicitar tarefas que ainda deverão ser alocadas. Dessa forma, as requisições são armazenadas na tabela de requisições de tarefas e, na medida que as tarefas requisitadas são alocadas, as tarefas que fizeram as requisições são desbloqueadas, podendo ser escalonadas novamente. Ou seja, se existe requisição para (t_i) na tabela *RequestTask*, essa requisição vai ser removida e (t_j) vai ser desbloqueada.

3.2 O Microkernel dos Processadores Escravos

Conforme mencionado anteriormente, o processador escravo suporta a execução de múltiplas tarefas e a comunicação entre as mesmas. Um sistema operacional multitarefa permite que várias tarefas compartilhem a utilização de um mesmo processador, ou seja, várias tarefas são

executadas concorrentemente. Essa abordagem requer gerenciamento de memória, escalonamento e mecanismos de comunicação entre as tarefas (SILBERCHATZ, 2000).

Conforme mostrado na Seção 2.2.1, a memória é dividida em páginas. O microkernel reside na primeira página e as outras tarefas nas páginas seguintes. Dessa forma, o gerenciamento de memória, utilizado na plataforma, trata apenas de determinar as páginas nas quais residem tarefas que estão sendo executadas.

O escalonamento de tarefas é preemptivo. O algoritmo de escalonamento utilizado é o *Round Robin*, no qual as tarefas são escalonadas de forma circular e cada uma delas é executada durante uma fatia de tempo (*timeslice*). Sempre que uma tarefa nova for escalonada, o contexto da tarefa que estava sendo executada é salvo. Após o escalonamento, o contexto da tarefa nova é restaurado.

A comunicação entre as tarefas ocorre por meio de troca de mensagens que utilizam *pipes*. Um *pipe* é uma área da memória onde são armazenadas, até serem consumidas, as mensagens que as tarefas enviam para outras tarefas. Cada tarefa possui o seu respectivo *pipe*, no qual são armazenadas as mensagens que essa tarefa envia para as demais.

3.2.1 Estrutura do microkernel dos processadores escravos

A estrutura do microkernel dos processadores escravos é mostrada na Figura 42. Ela consiste de três níveis de serviços. No nível 1, encontra-se o serviço de inicialização do sistema. No nível 2, encontram-se os *drivers* de comunicação. No nível 3, encontram-se os serviços de tratamento de interrupções, chamadas de sistema, escalonamento de tarefas e comunicação entre tarefas.

Os serviços do microkernel dos processadores escravos foram implementados parte em linguagem de montagem e parte em linguagem C. Os *drivers* de comunicação e o serviço de tratamento de interrupções foram implementados em linguagem de montagem. Os serviços de chamadas de sistema, escalonamento de tarefas e comunicação entre tarefas foram implementados em linguagem C.

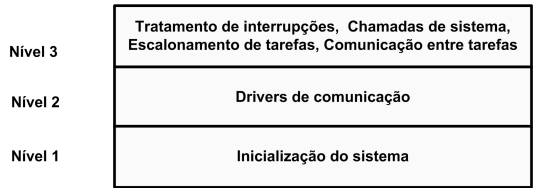


Figura 42: Microkernel do processador escravo

3.2.2 Estruturas de dados dos processadores escravos

Os serviços do microkernel dos processadores escravos possuem várias estruturas de dados. A estrutura *TaskLocation*, idêntica àquela apresentada na Seção 3.1.3, forma uma tabela que contém a associação de que tarefa (`task`) está localizada a que processador (`processor`), sendo consultada toda vez que ocorre comunicação entre tarefas, conforme será explicado na Seção 3.2.8. A estrutura *Message*, mostrada na Figura 43, é utilizada para gerenciar mensagens. Cada mensagem armazenada no *pipe* possui um tamanho (`lenght`), uma tarefa de destino (`target`), uma tarefa de origem (`source`) e o conteúdo (*msg*). A estrutura *TCB* (*Task Control Block* ou bloco de controle de tarefas) é utilizada para gerenciar a execução das tarefas. Para cada tarefa em execução, o microkernel mantém um *TCB*, cuja estrutura é mostrada na Figura 44.

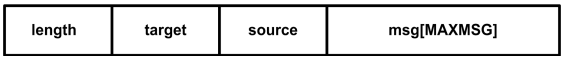


Figura 43: Estrutura *Message*

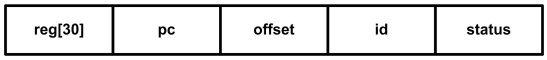


Figura 44: Estrutura do TCB

O *TCB* mantém os valores de 30 registradores do Plasma, do contador de programa (`PC`), o endereço inicial da tarefa (`offset`), a identificação da tarefa (`id`) e o estado da tarefa (`status`). Os registradores do Plasma, salvos no *TCB*, consistem dos registradores de retorno de valor de função ($v0 e $v1), registradores de parâmetros de função ($a0 a $a3), registradores de valores temporários de função ($t0 a $t9), registradores salvos por meio de chamadas de função ($s0 a $s8), o ponteiro para dados globais ($gp), ponteiro para pilha de dados ($sp), endereço de retorno de chamada de função ($ra), e dois registradores ($lo e $hi), utilizados em operações de multiplicação e divisão. O estado de uma tarefa t pode ser *ready*, quando t está pronta para ser executada, *running*, quando t está utilizando a CPU, *terminated*, quando t terminou a sua execução, *waiting*, quando t requisita uma mensagem e aguarda a resposta, e *allocating*, quando o *TCB* está sendo alocado.

A estrutura *RequestTask*, mostrada na Figura 45, forma uma tabela que contém a associação de uma tarefa (`requesting`) que está requisitando outra tarefa (`requested`), sendo utilizada para armazenar a requisição de uma tarefa. A estrutura *RequestMessage*, mostrada na Figura 46, forma uma tabela que contém a associação de uma tarefa (`requesting`) que está

requisitando que uma outra (`requested`) envie uma mensagem, sendo utilizada para armazenar a requisição de uma mensagem.

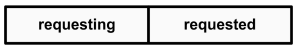


Figura 45: Estrutura *RequestTask*

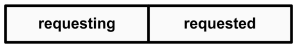


Figura 46: Estrutura *RequestMessage*

Além dessas estruturas, o microkernel mantém um *pipe* global de mensagens. Associados ao *pipe*, estão os vetores `pipe_order`, que indica a ordem de chegada das mensagens no *pipe*, e `pipe_ocupation`, que indica quais posições do *pipe* estão ocupadas. O microkernel também possui uma tarefa inativa (`idle`). Essa tarefa é uma função que executa um laço infinito. Ela é utilizada para permitir que o microkernel fique aguardando por interrupções vindas da interface de rede, quando não tem tarefas a serem escalonadas e executadas. Há um vetor de *TCB*s com quatro posições,em que cada índice do vetor corresponde a uma tarefa. A tarefa `idle` está localizada no última posição do vetor.

3.2.3 Inicialização do microkernel dos processadores escravos

O microkernel dos processadores escravos começa a sua execução inicializando os registradores $gp e $sp do Plasma com, respectivamente, o ponteiro de dados global e o ponteiro para a pilha referentes ao microkernel. Em seguida, o microkernel lê o endereço do processador (`netAddress`), inicializa a tabela de localização de pipes (`pipe_location`) e a tabela de localização de tarefas (`task_location`), indicando que as mesmas estão livres. Depois disso, o microkernel inicializa o vetor de *TCB*s, indicando que todos estão livres. Conforme mostra a Figura 47, para cada *TCB* é associada uma página da memória e um deslocamento (`offset`), sendo que o microkernel reside na página 0.

Depois disso, o microkernel inicializa a tabela de requisição de tarefas (*RequestTask*) e a tabela de requisição de mensagens (*RequestMessage*), indicando que as mesmas estão livres. Então, o microkernel habilita as interrupções vindas da interface de rede, do controlador de DMA e do contador de *timeslice*. Por último, o microkernel executa a função *Scheduler()* para escalonar as tarefas.

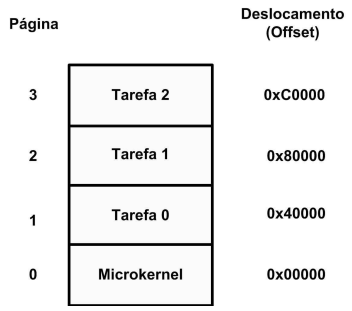


Figura 47: Configuração de memória

Após a inicialização do microkernel, a tarefa `idle` é escalonada. Enquanto essa tarefa está em execução, o processador escravo aguarda por uma interrupção da interface de rede (chegada de pacotes) ou do DMA (nova tarefa na memória).

3.2.4 Tratamento de interrupções

Conforme foi dito anteriormente, o microkernel do processador mestre habilita todas as interrupções. Isso é realizado quando a função *OS_InterruptMaskSet*(0B00111000H), responsável por configurar a máscara de interrupções, e a função *OS_AsmInterruptEnable*(1), responsável por habilitar as interrupções, são executadas são executadas.

No tratamento de interrupções, inicialmente, o microkernel salva o contexto da tarefa interrompida, armazenando no *TCB* dessa tarefa os registradores, o `pc`, o `offset` e o identificador da tarefa (`id`). Em seguida, os registradores $sp e $gp são configurados com valores referentes ao microkernel. Então, a função *OS_InterruptServiceRoutine*(*status*) é executada para verificar a origem da interrupção e, consequentemente, chamar a função designada para tratar a interrupção. Após o tratamento da interrupção, o contexto da tarefa interrompida é restaurado. Se a origem da interrupção foi o contador de *timeslice*, uma nova tarefa foi escalonada e começa a ser executada. Caso contrário, a tarefa que estava sendo executada, antes da interrupção, retoma a sua execução.

3.2.5 Chamadas de sistema

Uma chamada de sistema é uma interrupção gerada por *software*, cujo objetivo é requisitar um serviço do sistema operacional (SILBERCHATZ, 2000). Os sistemas operacionais possuem chamadas de sistema utilizadas para diversos propósitos, tais como, operações de entrada e saída, gerenciamento de processos, gerenciamento de arquivos. Nesta plataforma, as chamadas de sistema são utilizadas para terminar a execução de uma tarefa, realizar a comunicação entre

tarefas, informar a uma tarefa o valor do contador de ciclos de relógio, exibir uma mensagem e informar para uma tarefa qual é o seu identificador. A Tabela 10 mostra os serviços das chamadas de sistema utilizadas pelo microkernel da plataforma.

Tabela 10: Serviços das chamadas de sistema

Chamada de sistema	Identificador
EXIT	0
WRITEPIPE	1
READPIPE	2
GETTICK	3
ECHO	4
GETPROCESSID	5

O serviço EXIT é de utilização interna do sistema operacional, não podendo ser utilizado pelas tarefas. Os serviços são definidos, na verdade, como sendo uma função chamada $SystemCall(service, \&msg, TaskID)$. Essa função é implementada utilizando a instrução de montagem *Syscall*. Quando uma chamada de sistema ocorre, o fluxo de execução salta para o endereço 0x44H, onde a chamada de sistema é tratada.

No tratamento das chamadas de sistema, inicialmente, o microkernel salva o contexto da tarefa interrompida. Em seguida, a função $Syscall(service, \&msg, TaskID)$, cujo pseudocódigo é mostrado no Algoritmo 4, é executada. Então, o contexto da tarefa interrompida é restaurado e a tarefa interrompida retoma a sua execução.

Algoritmo 4 Função $Syscall(s, msg, t)$

1: **Se** s = EXIT **Então**
2: **Coloque** a tarefa cuja execução terminou no estado TERMINATED;
3: **Se** existirem ainda mensagens no pipe **Então**
4: **Consuma** as mensagens em msg;
5: **Fim Se**
6: **Remova** tarefa da tabela $tasks\_location$;
7: **Escalone** outra tarefa;
8: $DRV\_TerminatedTask$(MASTERADDRESS, $netAddress, task$)
9: **Senão Se** s = WRITEPIPE **Então**
10: $WritePipe(msg, t)$ para enviar a mensagem;
11: **Senão Se** s = READPIPE **Então**
12: $ReadPipe(msg, t)$ para receber a mensagem;
13: **Senão Se** s = ECHO **Então**
14: **Exiba** mensagem;
15: **Senão Se** s = GETPROCESSID **Então**
16: **Retorne** o identificador da tarefa corrente;
17: **Fim Se**

Uma chamada de sistema espera 3 parâmetros: o primeiro indica o serviço desejado, sendo 0 para EXIT, 1 para WRITEPIPE, 2 para READPIPE, 3 para GETTICK, 4 para ECHO e 5 para GETPROCESSID. Se o serviço for EXIT, que é solicitado somente pelo microkernel quando uma tarefa t_i terminou sua execução, os dois parâmetros seguintes são ignorados, o estado dessa tarefa passa a ser *terminated*, uma nova tarefa deve ser escalonada e, se ainda existirem mensagens enviadas por t_i no *pipe*, estas serão consumidas. Quando as mensagens de t_i tiverem sido todas consumidas, a mesma é removida da tabela `tasks_location`, o seu estado passa a ser *free* e o *driver* $DRV\_TerminatedTask$(MASTERADDRESS, $netAddress$, $TaskID$) é executado, recebendo como parâmetros o endereço do processador mestre, o endereço do processador escravo, onde a tarefa está alocada, e o seu identificador. Esse *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <processor> <task_id>
```

onde <target> é MASTERADDRESS; <size> = 3, se o tamanho do $flit$ = 32, ou 6, se o tamanho do $flit$ = 16; <service> é TERMINATED_TASK; <processor> é $netAddress$; <task_id> é *TaskID*.

Se o serviço solicitado for WRITEPIPE, os dois parâmetros seguintes possuem, respectivamente, o ponteiro onde se encontra a mensagem que será escrita no *pipe* e o identificador da tarefa destino. Se o serviço desejado for READPIPE, os dois parâmetros seguintes possuem, respectivamente, o ponteiro onde se encontra a mensagem que será lida do *pipe* e o identificador da tarefa origem. Se o serviço for GETTICK, os dois parâmetros seguintes são ignorados e o valor do relógio é informado à tarefa. Se o serviço for ECHO, o segundo parâmetro possui o ponteiro no qual se encontra a mensagem e o terceiro parâmetro é ignorado. Se o serviço for GETPROCESSID, os dois parâmetros seguintes são ignorados e o identificador da tarefa, que está sendo executada, é retornado.

3.2.6 Drivers de comunicação

Na Seção 2.2.3, foi dito que os *drivers* são responsáveis pelo envio e recepção dos pacotes da rede intrachip. Também foram mostrados os diferentes serviços que um pacote pode carregar e o formato do pacote para cada serviço. Para cada serviço é realizado um tratamento diferente. Assim sendo, o Algoritmo 5 mostra o pseudocódigo da função $DRV\_Handler()$, que é executada pelos processadores escravos quando ocorre um interrupção devido a chegada de pacotes da rede intrachip. Essa função faz chamadas aos *drivers* de comunicação necessários para o tratamento dos pacotes. Todos os *drivers* são escritos em linguagem de montagem. Quando chega um

pacote da rede intrachip, o *driver* $DRV\_ReadService(\&service, \&size)$ é executado, cuja função é obter o serviço carregado pelo pacote.

Algoritmo 5 Função $DRV\_Handler()$ dos processadores escravos

1: $DRV\_ReadService(s)$ para obter o identificador do serviço;
2: **Se** s = REQUEST_MESSAGE **Então**
3: $DRV\_DeliverMessage(netAddress)$ para enviar a mensagem para a tarefa fez o requerimento da mensagem;
4: **Senão Se** s = DELIVER_MESSAGE **Então**
5: $DRV\_ReadMessage()$ para ler a mensagem e depois entregá-la para a tarefa de destino;
6: **Senão Se** s = TASK_ALLOCATION **Então**
7: **Para** $i := 0 \ldots$ MAXGLOBALTASKS **Faça**
8: **Se** foi encontrado um TCB livre **Então**
9: **Modifique** o estado do TCB para *allocating*;
10: PC := 0;
11: **Desabilite** as interrupções vindas da interface de rede;
12: $DRV\_StartAllocation(offset)$ para alocar o código objeto na memória
13: **Fim Se**
14: **Fim Para**
15: **Senão Se** s = ALLOCATED_TASK **Então**
16: $DRV\_AllocatedTask()$ para informar que uma tarefa foi alocada;
17: **Senão Se** s = DEALLOCATED_TASK **Então**
18: $DRV\_DeallocatedTask()$ para informar que uma tarefa foi desalocada;
19: **Remova** tarefa da tabela *tasks_location*;
20: **Senão Se** s = FINISHED_ALLOCATION **Então**
21: $DRV\_FinishedAllocation()$ para informar que a alocação de tarefas terminou;
22: **Para** $j := 0 \ldots$ MAXGLOBALTASKS **Faça**
23: **Se** foi encontrada uma requisição de tarefa **Então**
24: $DRV\_RequestTask$(MASTERADDRESS, $netAddress, task$) para requisitar a alocação de uma tarefa;
25: **Fim Se**
26: **Fim Para**
27: **Fim Se**

Se o serviço for REQUEST_MESSAGE, o *driver* $DRV\_DeliverMessage(netAddress)$ é executado, recebendo como parâmetro o endereço do processador local. Esse *driver* lê, da interface de rede, o processador de origem, a tarefa de destino e a tarefa de origem da mensagem. Com essas informações, é procurada uma mensagem no *pipe*. Se ela for encontrada, o *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <source_slave_processor>
<message_target> <message_source> <message_size> <message>
```

onde <target> é o processador de origem da mensagem; <size> = 4 + tamanho da mensagem, se o tamanho do $flit$ = 32, ou 8 + 2× (tamanho da mensagem), se o tamanho do $flit$

= 16; <service> é DELIVER_MESSAGE; <source_slave_processor> é *netAddress*; <message_target> é a tarefa de destino da mensagem; <message_source> é a tarefa de origem da mensagem; <message_size> é o tamanho da mensagem; <message> é conteúdo da mensagem.

Se o serviço for DELIVER_MESSAGE, o *driver DRV_ReadMessage*(), responsável por ler a mensagem e entregá-la para a aplicação, é executado. Esse *driver* lê, da interface de rede, o processador de origem, a tarefa de destino e a tarefa de origem da mensagem. O endereço da memória para a onde a mensagem deve ser copiada é procurado no *TCB* da tarefa.

Se o serviço for TASK_ALLOCATION, o código objeto de uma tarefa está sendo transferido da interface de rede para a memória do processador escravo. Então, para alocar a tarefa, é procurado um *TCB* livre. O estado desse *TCB* passa a ser *allocating* e o `pc` é carregado com 0. A variável *allocatingTCB* contém o endereço do *TCB* utilizado para armazenar a tarefa. As interrupções vindas da interface de rede são, então, desabilitadas. O *driver DRV_StartAllocation*(tcbs[i].offset) é executado para alocar o código objeto da tarefa na memória do processador escravo, recebendo como parâmetro o endereço a partir do qual a tarefa vai ser alocada. Esse *driver* lê, da interface de rede, o identificador da tarefa, que é armazenado no *TCB* referenciado por *allocatingTCB*, e o código objeto da mesma. Então, ele informa ao controlador de DMA o tamanho do código objeto da tarefa (escrevendo no registrador SET_DMA_SIZE), o endereço da memória a partir do qual o código será transferido (escrevendo no registrador SET_DMA_ADDRESS) e inicia o DMA (escrevendo no registrador START_DMA).

O processador escravo continua sua execução em paralelo com o controlador de DMA, que realiza a transferência do código objeto para a memória. O DMA interrompe a CPU quando a transferência for concluída. O Algoritmo 6 mostra o pseudocódigo da função que trata a interrupção do DMA. A tarefa, cujo código foi transferido para a memória, é colocada na tabela *task_location*. A tarefa, que está ocupando o *TCB* referenciado por `allocatingTCB`, passa a ter estado *ready*. O microkernel avisa ao DMA que a interrupção foi aceita. Se a tarefa que estava executando antes da interrupção era a tarefa `idle`, uma nova tarefa deve ser escalonada. As interrupções vindas da interface de rede são, então, habilitadas.

Se o serviço for ALLOCATED_TASK, o *driver DRV_AllocatedTask*() é executado para informar que uma tarefa foi alocada. Esse *driver* lê, da interface de rede, o endereço do processador, no qual a tarefa foi alocada, e o identificador da tarefa. Esses dados são inseridos na tabela de alocação de tarefas (`task_location`). Se o serviço for DEALLOCATED_TASK, o *driver DRV_DeallocatedTask(&task)* é executado para informar que uma tarefa foi desalocada.

Algoritmo 6 Função $DMA\_Handler()$

1: **Insira** identificador da tarefa alocada na tabela tasks_location;
2: **Modifique** o estado da tarefa para READY;
3: **Informe** ao microkernel que a interrupção foi aceita;
4: **Se** a tarefa que estava sendo executada antes da interrupção for idle **Então**
5: **Escalone** uma nova tarefa;
6: **Fim Se**
7: **Habilite** as interrupções da interface de rede;

Este *driver* lê, da interface de rede, o identificador da tarefa que deve ser desalocada. Em seguida, a tarefa é desalocada. Se o serviço for FINISHED_ALLOCATION, o processador mestre terminou a alocação estática e, se existe alguma requisição de tarefa pendente na tabela de requisição de tarefas (`RequestTask`), então o *driver* $DRV\_RequestTask$(MASTERADDRESS, $netAddress$, $requestTask[i].requested$) é executado, recebendo como parâmetros o endereço do processador mestre, o endereço do processador escravo, que está solicitando a tarefa, e o identificador da mesma. Esse *driver* monta e envia um pacote com os seguintes campos:

```
<target> <size> <service> <processor> <task_id>
```

onde <target> é MASTERADDRESS; <size> = 3 se o tamanho do $flit$ = 32 ou 6 se o tamanho do $flit$ = 16; <service> é REQUEST_TASK; <processor> é $netAddress$; <task_id> é $requestTask[i].requested$.

3.2.7 Escalonamento de tarefas

A pseudocódigo da função de escalonamento de tarefas é mostrado no Algoritmo 7. O escalonamento utilizado pelo microkernel é preemptivo e sem prioridades. As tarefas são escalonadas de maneira circular de acordo com a política *Round Robin* (TANENBAUM, 1997).

O estado da tarefa interrompida passa a ser *ready*. O escalonador procura a próxima tarefa que será executada: se a tarefa que estava sendo executada é a tarefa `idle`, então a próxima tarefa que deve ser executada será a primeira; caso contrário, será a próxima. Se essa tarefa está pronta para executar, ela é escalonada, o seu estado passa a ser *running*, as interrupções vindas do contador de *timeslice* são habilitadas e o mesmo é reinicializado. Se nenhuma tarefa foi escalonada, a tarefa `idle` é executada e as interrupções vindas do contador de *timeslice* são desabilitadas.

O escalonador pode entrar em execução sem que uma interrupção do contador de *timeslice* ocorra. As situações em que esse evento ocorre são: uma tarefa terminou sua execução (Seção 3.2.5); uma tarefa está esperando uma mensagem que está em outro processador (Seção

Algoritmo 7 Função *Scheduler*

1: **Se** o estado da tarefa interrompida é *running* **Então**
2: **Modifique** o estado dessa tarefa para *ready*;
3: **Fim Se**
4: **Para** $i := 0$ Até número máximo de tarefas **Faça**
5: **Se** a tarefa que estava sendo executada é *idle* **Então**
6: **Execute** a primeira tarefa;
7: **Senão**
8: **Execute** a próxima tarefa;
9: **Armazene** o endereço da tarefa a ser executada;
10: **Se** estado = *ready* **Então**
11: **Selecione** a tarefa para escalonamento;
12: **Fim Se**
13: **Se** a tarefa está selecionada para ser escalonada **Então**
14: **Mude** o estado dessa tarefa para *running*;
15: **Habilite** as interrupções do contador de *timeslice*;
16: **Reinicialize** o contador de *timeslice*;
17: **Fim Se**
18: **Fim Se**
19: **Fim Para**

3.2.8); uma nova tarefa é alocada na memória do processador escravo, enquanto a tarefa `idle` está sendo executada (Seção 3.2.6); e resposta de uma requisição de mensagem, enquanto a tarefa *idle* está sendo executada (Seção 3.2.8). Nessas situações, uma nova tarefa deve ser escalonada.

3.2.8 Comunicação entre tarefas

A comunicação entre tarefas ocorre através de *pipes*. As mensagens enviadas pelas tarefas são escritas em um *pipe* global e só enviadas pela rede mediante requisição da tarefa destino. Quando a mensagem não está disponível, é enviado um pacote de controle indicando que a mensagem não existe.

As tarefas se comunicam através de duas funções. A função utilizada para enviar uma mensagem é *WritePipe*(&mensagem,id_destino), onde &mensagem especifica o endereço lógico dentro da página onde está a tarefa, que armazena a mensagem, e id_destino é o identificador da tarefa para a qual será enviada a mensagem. A função utilizada para receber uma mensagem é $ReadPipe(\&mensagem, id\_origem)$, onde $\&mensagem$ especifica o endereço lógico dentro da página onde está a tarefa, que armazenará a mensagem, e id_origem é o identificador da tarefa que enviou a mensagem.

A comunicação pode acontecer entre tarefas que residem no mesmo processador ou processadores diferentes. Considere a Figura 48. Quando uma tarefa t_1, no processador $Proc_1$, de-

seja receber uma mensagem de uma tarefa t_2, o microkernel verifica na tabela `tasks_location` qual a localização de t_2. Se t_2 encontra-se no processador local, o microkernel copia a mensagem do *pipe* para a página de t_1, conforme mostra a Figura 48. Caso t_2 encontre-se em um outro processador, por exemplo $Proc_2$, o microkernel em $Proc_1$ monta um pacote de requisição (`request_msg`) e o envia para $Proc_2$, requisitando uma mensagem de t_2 para t_1, conforme mostra a Figura 49a. Em seguida, a tarefa t_1 é colocada em espera (estado *waiting*) e uma nova tarefa é escalonada em $Proc_1$. Então, o microkernel em $Proc_2$ recebe a requisição e verifica se existe no *pipe* uma mensagem para t_1. Se existe, o pacote, contendo as informações e o conteúdo da mensagem, é enviado a $Proc_1$, como mostra a Figura 49b. Se não, é enviado um pacote com o serviço NO_MESSAGE. Quando $Proc_1$ recebe a resposta da requisição, t_1 passa a ter estado *ready* e pode ser novamente escalonada para continuar sua execução. Se a resposta contiver a mensagem esperada, ela é copiada para o endereço especificado por $\&msg$. Se a resposta for um pacote com o serviço NO_MESSAGE, t_1 pode tentar novamente receber a mensagem. Para isso, o recebimento da mensagem, na aplicação, deve ser implementado por várias chamadas à função $ReadPipe(\&mensagem, id\_origem)$.

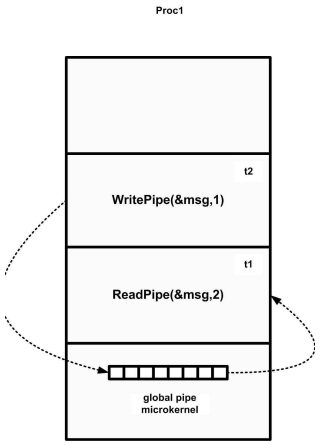


Figura 48: Comunicação entre tarefas residentes no mesmo processador

Uma tarefa entra em estado de espera somente quando executa *ReadPipe* de uma mensagem, cuja tarefa está em um processador remoto. Caso contrário, isto é, quando a mensagem é de uma tarefa local, não é necessário entrar em estado de espera.

Assim como a função *ReadPipe* pode não ser concluída com sucesso, ou seja, quando a mensagem esperada não está disponível, o envio de mensagens também pode falhar. Isso acontece quando o *pipe* está cheio e, portanto, não há mais espaço para novas mensagens. Dessa forma, o envio de uma mensagem também pode ser implementado através de uma sequência de

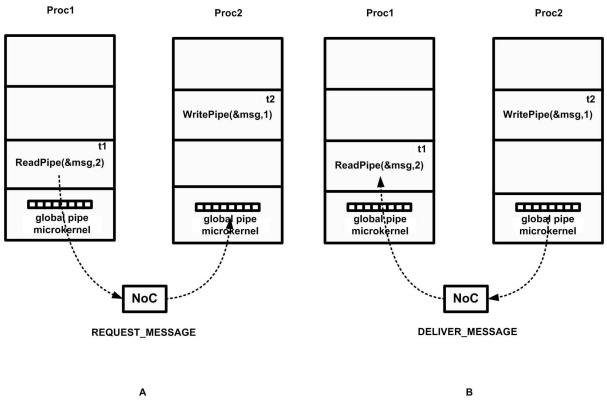


Figura 49: Comunicação entre tarefas residentes em processadores diferentes

execuções da função *WritePipe*(&mensagem,id_destino), garantindo que, em algum momento, a mensagem vai ser escrita no *pipe* para posterior leitura.

As mensagens são ordenadas no momento em que são armazenadas no *pipe*. Para cada mensagem, é associado um número inteiro indicando sua ordem (para isso é utilizado o vetor `pipe_order` (Seção 3.2.2). No envio de uma mensagem, *pipe* é percorrido para verificar se já existem mensagens da tarefa fonte para a tarefa destino. Se existirem, é verificado qual a maior ordem existente. A nova mensagem é armazenada no *pipe*, indicando, em `pipe_order`, que sua ordem é a maior ordem encontrada mais 1. Quando uma tarefa desejar receber uma mensagem, a mensagem repassada a ela será a que tiver menor ordem.

As funções de comunicação *WritePipe* e *ReadPipe* ocasionam chamadas de sistema. Dessa forma, o microkernel assume o controle, gerenciando a leitura e escrita nos *pipes*, bem como a leitura e escrita de mensagens em endereços de memória de diferentes páginas.

3.3 Melhorias Realizadas no Microkernel da Plataforma

A plataforma HMPS possui limitações de *hardware* e *software* que levaram a modificações na mesma para que o AGPE pudesse ser compilado e executado. As mudanças no microkernel da plataforma são apresentadas a seguir.

3.3.1 Parametrização do tamanho do flit

A parametrização do tamanho do *flit* da chave e da interface de rede do processador Plasma, apresentadas na Seção 2.3.4 (do Capítulo 2), implicou em mudanças no código fonte dos *drivers* de comunicação do microkernel dos processadores mestre e escravo. Agora, dependendo do

tamanho do *flit*, devemos compilar a aplicação que será executada pelo sistema embutido multiprocessado, utilizando o kernel16, quando o tamanho do *flit* for de 16 *bits*, ou o kernel32, quando o tamanho do *flit* for 32 *bits*.

3.3.2 Parametrização do tamanho de página de memória

A parametrização do tamanho de página da memória e do número de páginas também implicou em mudanças no microkernel. Pelo fato do tamanho do código objeto do AGPE ser maior do que o tamanho de página da memória original de 16 KB, foi necessário modificar o código fonte do *mirokernel*. Para resolver esse problema, a função *OS_init()*, do microkernel do processador escravo, foi modificada para permitir um tamanho variável de página da memória e um número de páginas variável. Essa função é responsável por definir o endereço de início de cada tarefa do processador local. Devido ao fato de que cada página de memória pode abrigar somente uma tarefa, esse endereço de início de tarefa também corresponde ao endereço de início de página.

Também é necessário informar ao microkernel dos processadores escravos o tamanho de página e o número de páginas. Nesse trabalho, o modelo do processador e o código fonte do microkernel foram modificados para permitir a fácil alteração do endereço do processador mestre. Além disso, foram desenvolvidos o serviço *getprocessid*(), cuja função é informar o identificador da tarefa que está sendo executada, e a função *getprocessor*(), cuja função é informar o processador que está sendo utilizado para executar a tarefa corrente.

3.4 Considerações Finais

Esse capítulo apresentou uma descrição do microkernel, introduzindo os conceitos de alocação de tarefas, tratamento de interrupções, chamadas de sistema, escalonamento de tarefas e comunicação entre tarefas. Também foram apresentadas as modificações realizadas no *software* da plataforma. No próximo capítulo serão abordados os algoritmos genéticos paralelos.

Capítulo 4

ALGORITMOS GENÉTICOS

ESSE capítulo apresenta os princípios dos algoritmos genéticos. Seu objetivo é descrever os principais conceitos relacionados com algoritmos genéticos. Na Seção 4.1 são descritos os algoritmos genéticos sequenciais e em seguida, na Seção 4.2 são descritos os algoritmos paralelos.

4.1 Conceitos de Algoritmos Genéticos

Algoritmos genéticos são métodos de otimização global e busca, inspirados nos princípios de seleção natural e evolução de populações de seres vivos, encontrados na genética. Isto é realizado através de várias iterações, onde cada iteração é chamada de *geração*.

Em cada geração, são aplicados os princípios de seleção e reprodução a uma população de soluções. Através da seleção, se determina quais indivíduos dessa população conseguirão se reproduzir, gerando um número determinado de descendentes para a próxima geração. Em outras palavras, os indivíduos que representam as melhores soluções têm melhor chance de sobreviver e gerar descendentes.

Nos algoritmos genéticos, uma população de soluções candidatas para o problema em questão evolui quando operadores genéticos, criados a partir dos seus análogos biológicos, são aplicados às mesmas. De acordo com esse processo, existe uma tendência de que, em média, os indivíduos representem soluções cada vez melhores, à medida que são realizadas mais iterações.

Embora os algoritmos genéticos utilizem um método heurístico e probabilístico para obter os novos indivíduos, ele não pode ser considerado uma simples busca aleatória, uma vez que ele explora de forma inteligente as informações disponíveis para buscar novos indivíduos ou soluções otimizadas para um determinado problema. As técnicas de otimização e busca, geralmente, são caracterizadas pelo *espaço de busca*, onde estão todas as possíveis soluções do problema, assim como uma *função objetivo*,também chamada de *função de aptidão*, que é uti-

lizada para avaliar as soluções produzidas, associando a cada uma delas uma nota denominada *índice de aptidão*.

Devido ao fato dos algoritmos genéticos serem altamente inspirados na genética e na teoria da evolução das espécies, há uma analogia muito forte entre os termos utilizados na biologia e aqueles utilizados nos algoritmos. Na biologia, os cromossomos são formados por genes e se combinam para formar as características genéticas básicas de um indivíduo. Nos algoritmos genéticos, o cromossomo representa uma estrutura de dados que codifica uma solução para um problema, ou seja, um ponto no espaço de busca. Na biologia, o indivíduo é um simples membro da população. Nos algoritmos genéticos, um indivíduo é formado pelo cromossomo e sua aptidão. Os termos cromossomo e indivíduo são intercambiáveis na área de algoritmos genéticos, sendo utilizados de forma razoavelmente aleatória na literatura. Na biologia, alelo é a unidade de hereditariedade, que é transmitida pelos cromossomos e que controla as características de um organismo. Nos algoritmos genéticos, é um parâmetro codificado no cromossomo. Tanto na biologia como nos algoritmos genéticos, locus é a posição fixa em um cromossomo onde está localizado um determinado gene. Genótipo representa, na biologia, a composição genética contida no genoma, que é o conjunto completo de genes de um organismo. Nos algoritmos genéticos, representa a informação contida no cromossomo. Fenótipo, na biologia, é o conjunto de características físicas observáveis de um organismo. Nos algoritmos genéticos, representa o objeto, estrutura ou organismo construído a partir das informações do genótipo. É o cromossomo decodificado. Por exemplo, considere que o cromossomo codifica parâmetros, como as dimensões das vigas, no projeto de um edifício em construção. O fenótipo seria o edifício construído.

Em termos matemáticos, a otimização consiste em encontrar uma solução que corresponda ao ponto de máximo ou mínimo da função objetivo. Os algoritmos genéticos procuram privilegiar indivíduos com melhores aptidões, com isto tentam dirigir a busca para regiões do espaço de busca onde é mais provável que os pontos ótimos estejam. O fluxo de controle de um algoritmo genético típico é ilustrado no Algoritmo 8, onde g representa o número da geração atual.

O primeiro passo de um algoritmo genético típico é a geração de uma população inicial de indivíduos, que representam possíveis soluções do problema em questão. Durante o processo evolutivo, que ocorre a cada geração, esta população é avaliada e cada cromossomo recebe uma nota, o índice de aptidão, refletindo a qualidade da solução que ele representa. De um modo geral, os cromossomos mais aptos são selecionados e os menos aptos são descartados.

Algoritmo 8 Fluxograma de um algoritmo genético típico

1: **Geração** da população inicial;
2: **Avaliação** da aptidão dos indivíduos da população;
3: $g := 0$;
4: **Repita**
5: **Seleção** de indivíduos para formar a nova população;
6: **Cruzamento** entre os indivíduos selecionados;
7: **Mutação** dos indivíduos obtidos;
8: **Avaliação** de aptidão dos indivíduos da nova população;
9: $g := g + 1$;
10: **Até** critério de parada satisfeito
11: **Retorna** Indivíduo mais apto;

Os cromossomos selecionados podem sofrer modificações em suas características fundamentais através dos operadores genéticos de cruzamento e mutação, gerando descendentes para a próxima geração. Este processo é repetido até que uma solução satisfatória seja encontrada. As seções seguintes descrevem com mais detalhes cada etapa desse algoritmo.

4.1.1 Representação dos parâmetros

O ponto de partida para a solução de um problema de otimização ou busca qualquer, por meio de algoritmos genéticos, é a representação do problema a ser analisado, de modo que os algoritmos genéticos possam atuar adequadamente sobre ele. Naturalmente, para cada representação devem haver operadores genéticos correspondentes.

Os algoritmos genéticos processam populações de cromossomos. O cromossomo é uma estrutura de dados, geralmente um vetor de valores binários, inteiros, reais ou combinações dos três, que representa uma possível solução do problema a ser otimizado. De um modo geral, o cromossomo representa o conjunto de parâmetros da função objetivo, cuja resposta será otimizada. Dependendo da função, a resposta otimizada pode ser constituída do seu valor de máximo ou mínimo. O conjunto de todos os valores que o cromossomo pode assumir constitui o seu espaço de busca. Se o cromossomo representa n parâmetros de uma função objetivo, então o seu espaço de busca é um espaço com n dimensões. Vale observar que, nessas funções, cada parâmetro ocupa uma seção do cromossomo.

A maioria das representações são genotípicas. O *genótipo* é o conjunto de genes que define a constituição genética de um indivíduo e sobre eles é que serão aplicados os algoritmos genéticos. Essas representações utilizam vetores de tamanho finito. O genótipo de um indivíduo é tradicionalmente representado por um vetor de números binários, inteiros ou reais, onde cada elemento desse vetor está relacionado com a presença ou a ausência de uma determinada

característica relevante deste indivíduo. Esses elementos podem ser combinados para formar as características reais de um indivíduo, ou seja, o seu *fenótipo*. Essa representação é independente da solução a ser encontrada, uma vez que, representando a mesma em vetores binários, inteiros ou reais, as operações padrões podem ser utilizadas, facilitando também o seu emprego em outros tipos de soluções.

A representação binária, conforme a Figura 50, é a mais utilizada por ser de fácil manipulação e análise (DEJONG, 1975)(GOLDBERG, 1989)(HOLLAND, 1975). Entretanto, se um problema tem múltiplos parâmetros e o usuário desejar trabalhar com maior precisão, acabará utilizando cromossomos longos para representar soluções, o que exigirá não só uma quantidade maior de memória, como também um processador mais rápido.

A representação real, conforme a Figura 51, gera cromossomos menores, sendo de mais fácil compreensão (MICHALEWICZ, 1994)(ALDEN, 1991), e requer uma quantidade menor de memória do que a representação binária.

Além das representações binária e real, existem outros tipos de representação, como, por exemplo, a representação com inteiros e a representação para problemas de permutação e controle. No contexto desse trabalho foi utilizada a representação binária.

0	1	1	0	1	0	1	1

Figura 50: Exemplo de cromossomo com representação binária

3.5671	0.1945	2.6786	5.7113	1.2197	3.679	9.0012	1.1334

Figura 51: Exemplo de cromossomo com representação real

Para obter o valor de uma variável real codificado em um cromossomo utilizando a representação binária, utiliza-se a Equação 1 (LACERDA; CARVALHO, 1999).

$$x = min + (max - min)\frac{c_{10}}{2^t - 1}, \tag{1}$$

onde c_{10} é o cromossomo convertido da base 2 para a base 10, que está no intervalo [max, min], e t é o tamanho da variável em *bits*. Convém observar que, neste caso, a função objetivo possui somente um parâmetro. Se a função objetivo em questão possuir múltiplos parâmetros, cada um deles ocupará uma seção desse cromossomo. Um exemplo de cromossomo na base 2 poderia ser conforme Equação 2.

$$(c)_2 = 11110010 \tag{2}$$

Suponha que esse cromossomo de 8 *bits* de tamanho represente um número no intervalo $[-3, 3]$. Para decodificar c_2, primeiramente precisamos encontrar o valor de c_{10}, que é obtido convertendo c_2 da base 2 para a base 10, como ilustrado na Equação 3.

$$(c)_{10} = (11110010)_2 = (242)_{10} \tag{3}$$

A partir de $(c)_{10}$, tem-se o valor fracionário, conforme Equação 4.

$$x = -3 + (3 + 3)\frac{242}{2^8 - 1} = 2,69411 \tag{4}$$

4.1.2 Inicialização da população

A população inicial de cromossomos pode ser gerada basicamente pelos métodos descritos a seguir:

- Inicialização aleatória (GOLDBERG, 1989)(SILVA, 2005): Os cromossomos da população são gerados de forma aleatória dentro do espaço de busca.
- Inicialização determinística (MICHALEWICZ, 1994)(MITCHELL, 1988): Os cromossomos da população são gerados uniformemente dentro do espaço de busca.

O tamanho da população inicial deve ser grande o suficiente para garantir a diversidade e cobrir a maior área possível do espaço de busca. Isto implica em uma probabilidade de convergência maior, uma vez que a probabilidade de encontrar a solução desejada na população aumenta (CANTU-PAZ, 1995) (SILVA, 2005). No entanto, se o tamanho da população inicial for muito pequeno, a diversidade será menor. Isto implica em que a área do espaço de busca coberta será menor, a convergência poderá ser prematura e a solução obtida poderá não estar próxima do ótimo global (CANTU-PAZ, 1995) (SILVA, 2005). No contexto desse trabalho foi utilizado o método da inicialização aleatória.

4.1.3 Avaliação

A avaliação de cada cromossomo resulta em um valor denominado *aptidão*. Nos casos mais simples, utiliza-se o valor da função objetivo. Entretanto, o valor da função objetivo nem sempre é adequado para ser utilizado como aptidão (LACERDA; CARVALHO, 1999). Por exemplo,

a função objetivo pode assumir valores negativos. Neste caso, o método da roleta viciada não funciona, conforme será explicado na Seção 4.1.4.1. Por outro lado, também pode apresentar alguns valores muito elevados em relação ao resto da população. Neste caso, pode causar a convergência prematura. Em ambos os casos é necessário mapear os valores da função objetivo para valores de aptidão, que pode ser feito de vários métodos, dois dos quais serão discutidos a seguir.

4.1.3.1 Ordenamento linear

No método do Ordenamento Linear (BAKER, 1987)(WHITLEY, 1989), a aptidão é obtida pela Equação 5.

$$f_i = min + (max - min)\frac{N - i}{N - 1}, \tag{5}$$

onde i é o índice do cromossomo na população em ordem decrescente de valor da função objetivo. Normalmente, $1 \leq max \leq 2$ e $max + min = 2$ (BAKER, 1987). Neste método, a aptidão representa o número de filhos esperados do cromossomo e (max-min) representa a pressão de seleção, $\frac{f_max}{f}$, que é a razão entre a maior aptidão e a aptidão média. A alta pressão de seleção favorece bastante os melhores cromossomos, direcionando a busca para encontrar as melhores soluções até então. A baixa pressão de seleção favorece um pouco mais os cromossomos de baixa aptidão, direcionando a busca para regiões desconhecidas do espaço de busca. No contexto desse trabalho foi utilizado o método do ordenamento linear.

4.1.3.2 Ordenamento exponencial

No método do Ordenamento Exponencial (MICHALEWICZ, 1994), a aptidão é obtida pela Equação 6.

$$f_i = q(1 - q)^{i-1}, \tag{6}$$

onde $q \in [0, 1]$ e i é o índice do cromossomo na população em ordem decrescente de valor da função objetivo. O ordenamento exponencial permite maior pressão de seleção do que o ordenamento linear.

4.1.4 Seleção

O método básico de funcionamento dos algoritmos genéticos é baseado no princípio da sobrevivência dos mais aptos, que é inspirado na seleção natural observada na biologia. De acordo com

esse princípio, os indivíduos melhor adaptados ao seu ambiente possuem naturalmente mais oportunidades para se reproduzirem e passarem as suas características genéticas para as próximas gerações, do que aqueles indivíduos considerados mais fracos. Nos algoritmos genéticos, os indivíduos com maior aptidão tem maior probabilidade de serem selecionados para participar na criação de indivíduos para a próxima geração. Os métodos mais comuns de seleção são descritos no restante da seção.

4.1.4.1 Roleta viciada

O método da roleta viciada é o mais simples e o mais utilizado (GOLDBERG, 1989). Os indivíduos de uma geração são selecionados para participar na criação de indivíduos para a próxima geração, utilizando uma roleta, semelhante à roleta utilizada nos jogos de azar. Neste método, cada indivíduo da população é representado na roleta conforme a sua aptidão. Ou seja, os indivíduos com elevada aptidão ocuparão um segmento maior na roleta do que aqueles que possuem baixa aptidão. Após a distribuição na roleta, é gerado um número aleatório no intervalo entre 0 e a soma das aptidões de todos os indivíduos. O indivíduo que possuir, em seu segmento, o valor gerado é selecionado e armazenado em uma população intermediária. Esse processo é repetido até que a população intermediária seja preenchida. De forma simplificada, o método da roleta pode ser realizado através dos seguintes passos:

1. Somar a aptidão de todos os indivíduos da população e armazenar o resultado em S;
2. Gerar um número aleatório R no intervalo $[0, S]$;
3. Somar a aptidão de cada um dos indivíduos da população e armazenar o valor da soma parcial em σ;
4. Se $\sigma \geq R$, então o indivíduo corrente é selecionado;
5. Se o número de indivíduos ainda não foi obtido, então retorne para o segundo passo.

Na Figura 52 é mostrado um exemplo do método de seleção pela roleta, onde os indivíduos com maiores valores de aptidão ocupam as maiores porções da roleta e possuem uma probabilidade maior de serem selecionados. Na Tabela 11 são mostrados os valores utilizados nesse exemplo. Na primeira coluna estão os indivíduos, na segunda o valor de aptidão e na terceira a porcentagem de espaço que cada um deles ocupa na roleta. No contexto desse trabalho foi utilizado o método da roleta viciada.

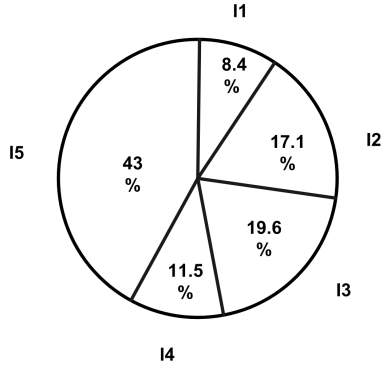


Figura 52: Método de seleção pela roleta

Tabela 11: Exemplo de seleção pelo método da roleta

Indivíduo	Aptidão	Porcentagem
I_1	1.7241	8.4
I_2	1.8212	17.1
I_3	1.9301	19.6
I_4	1.7919	11.5
I_5	1.9999	43

4.1.4.2 Torneio

No método do torneio, um número determinado de indivíduos da população é escolhido aleatoriamente, com a mesma probabilidade, para participar de um torneio e o indivíduo que possuir maior aptidão é selecionado para preencher a população intermediária. O processo é repetido até que a população intermediária seja preenchida (GOLDBERG, 1989).

4.1.4.3 Amostragem estocástica universal

Este método é uma variação do método da roleta viciada em que são selecionados N indivíduos de uma vez só. Desta forma, ao invés de girar a roda da roleta N vezes, ela é girada uma única vez (BAKER, 1987).

4.1.4.4 Elitismo

O elitismo consiste em fazer com que o algoritmo genético armazene o melhor indivíduo ou alguns dos melhores para serem utilizados na próxima geração. Desta forma, evita-se que o melhores indivíduos sejam destruídos pelos operadores de cruzamento e mutação (DEJONG, 1975). Vários estudos (MITCHELL, 1988) tem apontado que uso do elitismo contribui para o aumento do desempenho dos algoritmos genéticos.

4.1.5 Operadores genéticos

Um algoritmo de otimização global deve ser capaz de explorar novos pontos no espaço de busca e intensificar a busca em regiões promissoras. Esse mecanismo de diversificação e intensificação é obtido pela aplicação dos operadores genéticos, que transformam a população através de sucessivas gerações, estendendo a busca até obter um resultado satisfatório. Os operadores genéticos básicos encontrados na literatura são: cruzamento e mutação. Esses operadores são apresentados a seguir.

4.1.5.1 Cruzamento

O operador de cruzamento é baseado no fenômeno de mesmo nome, no qual ocorre a troca de fragmentos entre pares de indivíduos. Nos algoritmos genéticos, o operador de cruzamento seleciona genes de dois indivíduos pais para gerar dois indivíduos filhos que farão parte da nova população. A ideia central do operador de cruzamento é a propagação das características dos indivíduos mais aptos da população por meio da troca de informações entre os mesmos, o que dará origem a novos indivíduos. Os dois tipos principais de cruzamento são:

- Cruzamento binário (GOLDBERG, 1989): É utilizado na representação binária. Pode ser realizado de três diferentes maneiras, apresentados a seguir.

  1. Cruzamento de um ponto: Este método, mostrado na Figura 53, consiste na seleção aleatória de um ponto de corte, a partir do qual os segmentos dos indivíduos pais serão permutados, gerando dois indivíduos filhos diferentes.

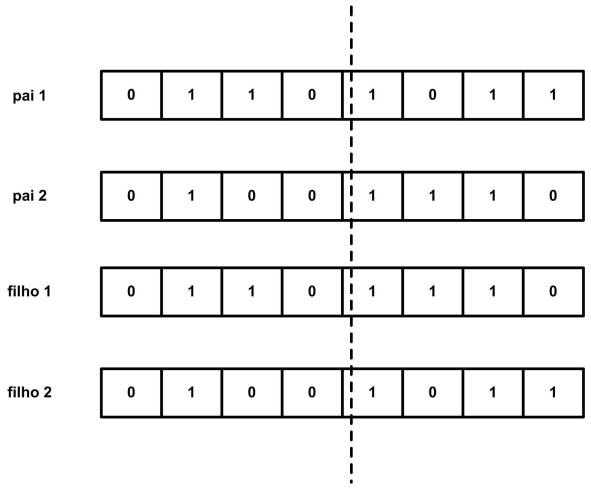


Figura 53: Cruzamento de um ponto

2. Cruzamento de dois pontos: Este método, mostrado na Figura 54, consiste na seleção aleatória de dois pontos de corte, que são utilizados para definir o intervalo dos segmentos dos indivíduos pais que serão permutados para gerar dois indivíduos filhos diferentes.

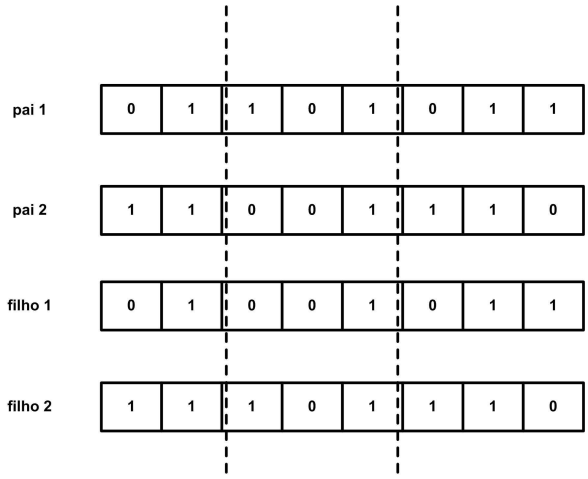


Figura 54: Cruzamento de dois pontos

3. Cruzamento uniforme: Neste método, mostrado na Figura 55, é gerada uma máscara de *bits* aleatórios para cada par de cromossomos pais. Se o primeiro *bit* da máscara possui o valor 1, então o primeiro *bit* do pai_1 é copiado para o primeiro *bit* do $filho_1$. Caso contrário, o primeiro *bit* do pai_2 é copiado para o primeiro *bit* do $filho_1$. O processo se repete para os *bits* restantes do $filho_1$. Na geração do $filho_2$ o procedimento é invertido, ou seja, se o *bit* da máscara é 1, então será copiado o bit do pai_2. Se o *bit* for igual a 0, então será copiado o *bit* do pai_1.

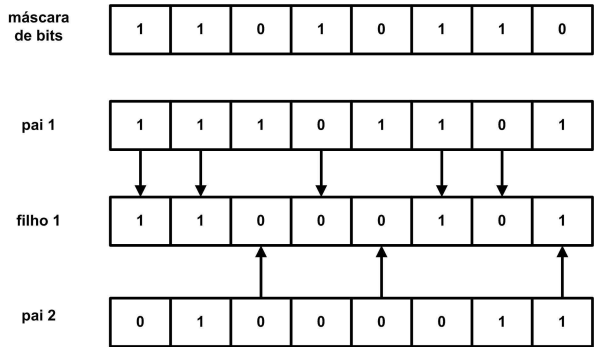


Figura 55: Cruzamento uniforme

- Cruzamento real: É utilizado com a representação real. Os principais métodos de cruzamento real são apresentados a seguir.

1. Cruzamento por media (DAVIS, 1991): Dados dois cromossomos pais P_1 e P_2, é produzido um cromossomo filho C da forma descrita na Equação 7.

$$C_i = (P_{1_i} + P_{2_i})/2, \tag{7}$$

onde i é o índice do elemento dentro do cromossomo.

2. Cruzamento por media geométrica (DAVIS, 1991): Dados dois cromossomos pais P_1 e P_2, é produzido um cromossomo filho C conforme a Equação 8.

$$C_i = \sqrt{P_{1_i} + P_{2_i}}, \tag{8}$$

onde i é o índice do elemento dentro do cromossomo. Este método pode causar perda de diversidade. Isto pode ser melhorado com o cruzamento por mistura.

3. Cruzamento por mistura (ESHELMAN L. J.; SHAFFER, 1992): Dados dois cromossomos pais P_1 e P_2, é produzido um cromossomo filho C da forma descrita na Equação 9.

$$C_i = P_{1_i} + \beta(P_{2_i} - P_{1_i}), \tag{9}$$

onde i é o índice do elemento dentro do cromossomo e β é um número aleatório gerado a partir de uma distribuição uniforme no intervalo $[-\alpha, 1+\alpha]$. O valor usual de α é 0.5 (LACERDA; CARVALHO, 1999). Este método evita a perda de diversidade que ocorre no cruzamento por media geométrica.

4. Cruzamento linear (ALDEN, 1991): Dados dois cromossomos pais P_1 e P_2, são produzidos três cromossomos filhos C_1, C_2 e C_3 conforme a Equação 10.

$$\begin{array}{rl} C_{1_i} = & 0.5P_{1_i} + 0.5P_{2_i} \\ C_{2_i} = & 1.5P_{1_i} - 0.5P_{2_i} \\ C_{3_i} = & -0.5P_{1_i} + 1.5P_{2_i}, \end{array} \tag{10}$$

onde i é o índice do elemento dentro do cromossomo. Desses três filhos, apenas o melhor é escolhido e os outros dois são descartados.

5. Cruzamento aritmético (MICHALEWICZ, 1994): Dados dois cromossomos pais P_1 e P_2, são produzidos dois cromossomos filhos C_1 e C_2 de acordo com a Equação 11.

$$\begin{array}{rl} C_{1_i} = & \beta P_1 + (1-\beta)P_2 \\ C_{2_i} = & (1-\beta)P_1 + \beta P_2, \end{array} \tag{11}$$

onde i é o índice do elemento dentro do cromossomo e β é um número aleatório gerado a partir de uma distribuição uniforme no intervalo [0, 1].

6. Cruzamento heurístico (MICHALEWICZ, 1994): Dados dois cromossomos pais P_1 e P_2, em que P_1 tem aptidão melhor que P_2, é produzido um cromossomo filho C, como descrito na Equação 12.

$$C_{1_i} = P_{1_i} + r(P_{1_i} - P_{2_i}), \tag{12}$$

onde i é o índice do elemento dentro do cromossomo e r é um número aleatório gerado a partir de uma distribuição uniforme no intervalo [0, 1].

No contexto desse trabalho foi utilizado o método do cruzamento de um ponto.

4.1.5.2 Mutação

O operador de mutação é baseado no fenômeno de mesmo nome, no qual são alterados alguns genes de alguns indivíduos da população após o cruzamento. Esse operador é utilizado para garantir a diversidade da população, que tende a tornar-se homogênea a longo prazo devido à utilização do operador de cruzamento. Os dois tipos principais de mutação são descritos a seguir. No contexto desse trabalho foi utilizada a mutação binária.

- Mutação binária (GOLDBERG, 1989): É utilizado na representação binária. Neste método, mostrado na Figura 56, é escolhido aleatoriamente um *bit* do indivíduo para ser alterado.

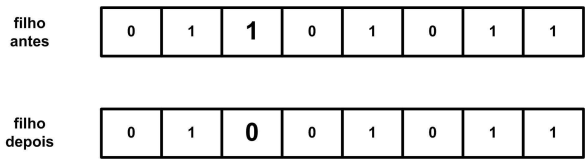


Figura 56: Mutação binária

- Mutação real: É utilizado na representação real. Os principais métodos de mutação real são descritos a seguir:

  - Mutação uniforme (LACERDA; CARVALHO, 1999)(MICHALEWICZ, 1994): Neste método, é escolhido aleatoriamente um elemento do indivíduo para ser substituído por um número aleatório, gerado a partir de uma distribuição uniforme no intervalo [a, b], onde a e b representam os limites inferior e superior para o elemento.
  - Mutação Gaussiana (LACERDA; CARVALHO, 1999)(MICHALEWICZ, 1994): Neste método, é escolhido aleatoriamente um elemento do indivíduo para ser substituído por um número aleatório gerado a partir de uma distribuição normal com media p e desvio padrão σ.

- Mutação por escorregamento (LACERDA; CARVALHO, 1999)(MICHALEWICZ, 1994): Neste método, é escolhido aleatoriamente um elemento do indivíduo ao qual será adicionado um número aleatório, gerado a partir de uma distribuição normal com media zero e desvio padrão pequeno. Alternativamente, a mutação por escorregamento pode ser realizada multiplicando-se esse elemento por um número aleatório próximo de um. O número aleatório deve ser pequeno o suficiente para causar apenas uma pequena perturbação no indivíduo, que, se estiver perto do ponto ótimo, pode movê-lo rapidamente para esse ponto. A taxa de mutação por escorregamento pode ser relativamente alta, visto que ela é utilizada apenas para explorar localmente o espaço de busca.
- Mutação limite (MICHALEWICZ, 1994): Neste método, é escolhido aleatoriamente um elemento do indivíduo para ser substituído por um dos limites do intervalo [a, b]. a será escolhido se $r < 0.5$ e b será escolhido se $r \geq 0.5$, onde r é um número aleatório gerado a partir de uma distribuição uniforme no intervalo [0, 1].
- Mutação não uniforme (MICHALEWICZ, 1994): Neste método, é escolhido aleatoriamente um elemento do indivíduo para ser substituído por um número aleatório, gerado a partir de uma distribuição não uniforme.

4.1.6 Parâmetros utilizados pelos algoritmos genéticos

O desempenho de um algoritmo genético é fortemente influenciado pela definição dos parâmetros a serem utilizados pelo mesmo. Sendo assim, é importante analisar de que maneira esses parâmetros influenciam no comportamento dos algoritmos genéticos, para que se possa estabelecê-los conforme as necessidades do problema e dos recursos disponíveis (CANTU-PAZ, 1995) (SILVA, 2005). Os principais parâmetros utilizados pelos algoritmos genéticos são:

- *Tamanho do cromossomo*: O tamanho do cromossomo está fortemente relacionado ao problema abordado e define a precisão das soluções candidadas para o problema a ser solucionado.

- *Tamanho da população*: Além de determinar o número de cromossomos da população, o tamanho da população afeta o desempenho dos algoritmos genéticos. Uma população pequena fornece uma cobertura reduzida do espaço de busca. Uma população grande geralmente fornece uma cobertura abrangente do espaço de busca, além de prevenir convergências prematuras em regiões de ótimos locais ao invés da região do ótimo global. Entretanto, grandes populações exigem esforço computacional maior.

- *Taxa de cruzamento*: Determina a probabilidade com que a operação de cruzamento ocorrerá. Um valor muito baixo para este parâmetro resulta num baixo aproveitamento da informação genética existente, tornando lento o processo de convergência para uma solução. Por outro lado, um valor muito alto pode resultar numa convergência prematura. O valor para esta taxa é normalmente alto, situando-se entre cinquenta 50 e 80 por cento (GOLDBERG, 1989) (LACERDA; CARVALHO, 1999).

- *Taxa de mutação*: Determina a probabilidade com que uma mutação ocorrerá. Um valor muito baixo para este parâmetro previne que a busca fique estagnada em certas regiões do espaço de busca, além de possibilitar que qualquer ponto do espaço de busca seja alcançado. Por outro lado, um valor muito alto torna a busca completamente aleatória. O valor para esta taxa é normalmente baixo, situando-se entre 0,1 e 10 por cento (GOLDBERG, 1989) (LACERDA; CARVALHO, 1999).

- *Condição de parada*: Seria interessante que o algoritmo genético terminasse a sua execução assim que o ponto de ótimo fosse encontrado. Entretanto, na maioria dos casos não se pode afirmar com certeza que um determinado ponto corresponde ao ótimo global. Então, usa-se normalmente o critério do número máximo de gerações para terminar a execução do algoritmo genético. Outro critério plausível é terminar a execução do algoritmo genético quando a população estagnar, não se observando melhoria após um determinado número de gerações consecutivas, ou seja, quando a aptidão media ou do melhor indivíduo não melhoria mais, ou quando as aptidões dos indivíduos tornarem-se muito parecidas. Também é possível utilizar o valor máximo da função objetivo, quando conhecida, como critério de parada.

4.2 Algoritmos Genéticos Paralelos

Os algoritmos genéticos sequenciais são inspirados em um processo evolutivo de populações de indivíduos que ocorre na natureza. Sendo assim, eles possuem uma estrutura computacional altamente paralelizável. A partir da análise da sua estrutura, pode-se chegar às seguintes conclusões (CANTU-PAZ, 1995) (SILVA, 2005):

- Cada indivíduo tem um índice de aptidão que pode ser avaliado independentemente de qualquer outro fator.

- Os operadores genéticos de cruzamento e mutação de diferentes indivíduos são independentes e podem ser aplicados em qualquer ordem, sequencial ou não, a qualquer elemento da população.

Entretanto, os algoritmos genéticos não exploram esse paralelismo intrínseco para melhorar o seu desempenho, o que motivou o desenvolvimento de algoritmos genéticos que utilizam o processamento paralelo.

O processamento paralelo é uma estratégia utilizada em computação para resolver mais rapidamente problemas computacionais complexos, dividindo-os em tarefas pequenas que serão alocadas a vários processadores para serem executadas simultaneamente. Esses processadores comunicam-se entre si para que haja sincronização, quando necessária, na execução das diversas tarefas em paralelo. A Figura 57 mostra o exemplo de uma tarefa complexa que é dividida em três tarefas pequenas que serão alocadas a três processadores. Cada uma dessas três tarefas pequenas consome um tempo de processamento menor do que a tarefa complexa e são executadas simultaneamente. Com isso, ocorre uma redução no tempo de processamento e o desempenho aumenta.

A sincronização entre as tarefas é uma alternativa que pode ser utilizada no processamento paralelo, onde, em um determinado instante, as tarefas executadas em paralelo aguardam a finalização mútua para trocar dados e reiniciar novas tarefas em paralelo. Entretanto, pode causar atraso em caso de desbalanceamento de carga.

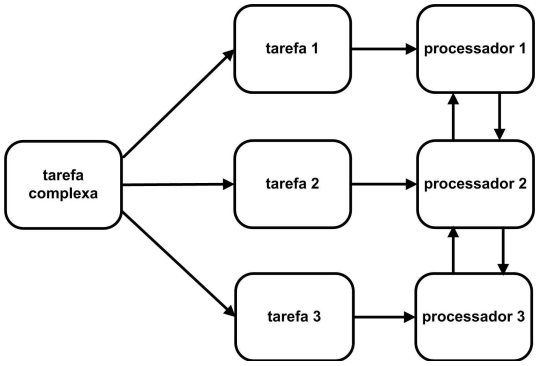


Figura 57: Particionamento de uma tarefa em três subtarefas, com subsequente alocação a três processadores

4.2.1 Tipos de paralelismo

Os principais tipos de paralelismo encontrados na literatura são apresentados a seguir:

4.2.1.1 Paralelismo de dados

Neste tipo de paralelismo, mostrado na Figura 58, uma mesma tarefa é alocada a vários processadores sendo que cada processador trabalha com um conjunto de dados diferente do outro. Neste caso, o conjunto de dados da tarefa em questão é decomposto em subconjuntos, onde cada subconjunto é tratado por um processador (BATISTA, 2005) (SILVA, 2005) (BLELLOCH, 1990) (GOMES, 2009). Por exemplo, essa estratégia pode ser usada na resolução de sistemas de equações (LONGHIN, 2001), multiplicação de matrizes (COSTA, 2002) e integração numérica (BARBOSA, 1998).

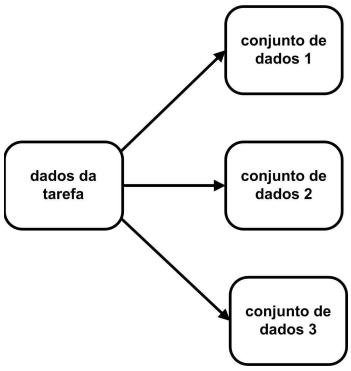


Figura 58: Paralelismo de dados

4.2.1.2 Paralelismo funcional

Neste tipo de paralelismo, mostrado na Figura 59, tarefas diferentes são alocadas a vários processadores(BATISTA, 2005) (SILVA, 2005) (GOMES, 2009). Neste caso, um programa é decomposto em um conjunto de tarefas, onde cada uma é executada por um processador. Por exemplo, essa estratégia pode ser utilizada para implementar o paradigma produtor-consumidor (HAUSEN, 2005) e em processamento de imagens (SALES, 2008).

4.2.1.3 Paralelismo de objetos

Neste tipo de paralelismo, mostrado na Figura 60, utiliza-se o conceito de objetos distribuídos por uma rede, capazes de serem acessados por tarefas em execução em vários processadores para uma determinada finalidade (BATISTA, 2005) (GOMES, 2009).

4.2.2 Plataformas para processamento paralelo

Uma plataforma para processamento paralelo é constituída de vários processadores interconectados por uma rede, um sistema operacional capaz de executar processamento paralelo e

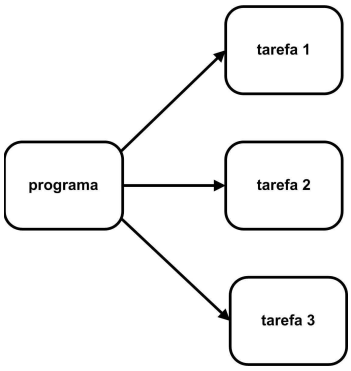


Figura 59: Paralelismo funcional

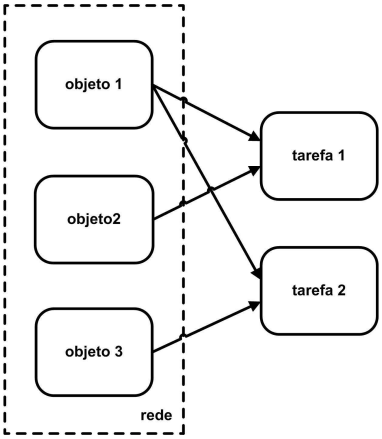


Figura 60: Paralelismo de objetos

também uma linguagem de programação que suporte um modelo de programação paralela. Existem dois modelos principais de programação paralela: o modelo baseado em memória compartilhada e o modelo baseado em passagem de mensagens.

4.2.2.1 Modelo de memória compartilhada

Neste modelo, as tarefas compartilham uma memória comum, na qual elas leem e escrevem de forma assíncrona (HAUSEN, 2005) (PACKARD, 1988). Para preservar a ordem de leitura e escrita, usam-se diversos mecanismos de sincronismo, como, por exemplo, semáforos (TANENBAUM, 1997) (SILBERCHATZ, 2000). Neste modelo, não existe o conceito de proprietário da informação e, assim, torna-se desnecessário explicitar a comunicação entre as tarefas. Isto pode simplificar o desenvolvimento de aplicações.

4.2.2.2 Modelo de troca de mensagens

Neste modelo, a comunicação entre as tarefas é realizada através do envio de mensagens pela da rede. O programador é responsável pela sincronização entre as mesmas (HAUSEN, 2005) (PACKARD, 1988). Modelos de troca de mensagens são implementados, em geral, por bibliotecas de comunicação que permitem a criação de programas paralelos, ou seja, o programa é escrito em uma linguagem sequencial, como C (KERNIGHAN, 1998), Fortran90 (CHIVERS, 2008), Fortran95 (CHIVERS, 2008) e HPF (*High Performance Fortran*) (KOELBEL C., 1993), estendida através de uma biblioteca que inclui funções para troca de mensagens entre as tarefas.

Os programas que se utilizam do modelo de troca de mensagens criam múltiplas tarefas, as quais encapsulam dados locais. Cada tarefa é identificada através de um número e interage com outras tarefas através de mensagens.

As principais bibliotecas de comunicação que implementam o modelo de troca de mensagens são o PVM (*Parallel Virtual Machine*) (GEIST A., 1994) e MPI (*Message Passing Interface*) (PACHECO, 1996).

4.2.2.3 Modelo de threads

Neste modelo, uma tarefa simples pode ter múltiplos fluxos de execução concorrentes (HAUSEN, 2005). Esforços de padronização, não relacionados entre si, resultaram em duas implementações diferentes de *threads*: POSIX Threads (BUTENHOF, 1997) e OpenMP (CHAPMAN, 2007).

As bibliotecas PVM, MPI e as linguagens Fortran 90, Fortran 95 e HPF são utilizadas normalmente em *clusters* para processamento paralelo (Beowulf), que executam sistemas operacionais Unix. Plataformas MPSoC, conforme detalhado no Capítulo 1, 2 e 3 possuem várias limitações, tais como: capacidade de processamento pequena, tamanho de memória limitado e sistema operacional com recursos reduzidos que impossibilitam a utilização dessas bibliotecas e linguagem. No contexto desse trabalho, foi utilizado o modelo de troca de mensagens implementado pelas funções *WritePipe* e *ReadPipe* do *microkernel* da plataforma Hermes MPSoC.

4.2.3 Modelos de algoritmos genéticos paralelos

A paralelização dos algoritmos genéticos tem sido bastante investigada nos últimos anos e vários modelos têm sido propostos por vários pesquisadores, tais como (CANTU-PAZ, 1995), (GOLDBERG, 1989), (ADAMIDIS, 1994) e outros, podendo ser classificados em três modelos:

4.2.3.1 Modelo de paralelização global

Este modelo, mostrado na Figura 61, é uma versão paralela de algoritmo genético sequencial que opera sobre uma população global, sendo adequado para a arquiteturas paralelas com memória compartilhada. Neste modelo, o processador principal ou mestre realiza as operações de seleção, cruzamento e mutação. Além disso, envia os indivíduos para avaliação de aptidão nos processadores secundários ou escravos e espera o resultado dessa avaliação para continuar a sua execução (CANTU-PAZ, 1995) (SILVA, 2005) (BARBOSA, 1998).

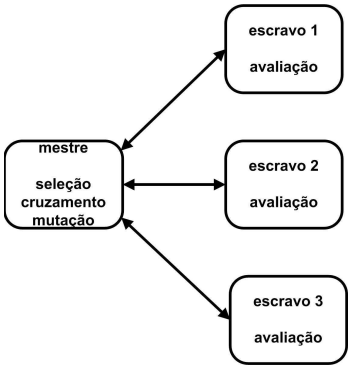


Figura 61: Modelo da paralelização global

4.2.3.2 Granularidade fina

Este modelo, mostrado na Figura 62, também é conhecido como modelo de vizinhança (*neighborhood model*), onde uma única população evolui e cada indivíduo é alocado a um processador de uma malha 2D de processadores, sendo adequado para as arquiteturas massiçamente paralelas com memória distribuída. Os processos de seleção e cruzamento são aplicados somente entre indivíduos vizinhos na malha 2D (CANTU-PAZ, 1995) (SILVA, 2005) (BARBOSA, 1998).

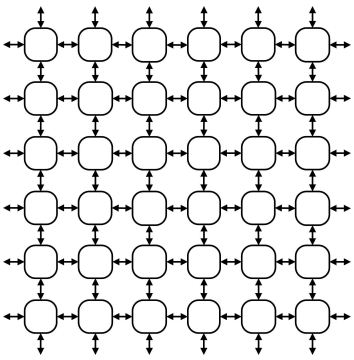


Figura 62: Modelo da granularidade fina

4.2.3.3 Granularidade grossa

Este modelo, mostrado na Figura 63, também é conhecido como modelo das ilhas (*island model*), pelo fato de cada processador ser considerado como uma ilha. Neste, várias subpopulações isoladas evoluem em paralelo e periodicamente trocam informações através da migração dos seus melhores indivíduos para as subpopulações vizinhas, sendo adequado para as arquiteturas paralelas com memória distribuída (CANTU-PAZ, 1995) (SILVA, 2005) (BARBOSA, 1998). Este foi o modelo utilizado neste trabalho.

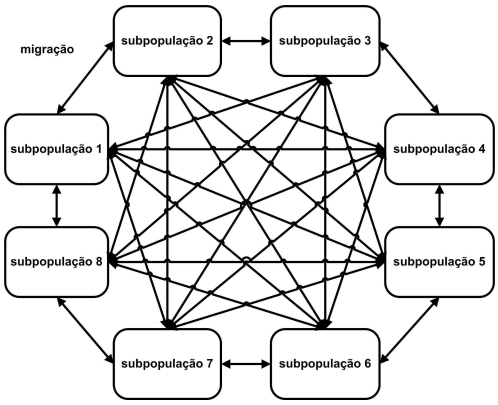


Figura 63: Modelo da granularidade grossa

O modelo de granularidade grossa introduz um operador de migração responsável por enviar e receber os melhores indivíduos de uma subpopulação para outra. Há várias topologias ou estratégias utilizadas para migrar indivíduos de uma subpopulação para outra. Entre eles podemos mencionar:

- Anel: Nesta topologia, mostrada na Figura, 64, os melhores indivíduos podem migrar somente para o vizinho da esquerda.
- Vizinhança: Nesta topologia, mostrada na Figura, 65, os melhores indivíduos podem migrar para os vizinhos da esquerda e da direita.
- Broadcast: Nesta topologia, mostrada na Figura, 66, os melhores indivíduos podem migrar para todos os vizinhos.

As migrações dos indivíduos podem ser implementadas de dois modos: síncrono e assíncrono. No modo síncrono, uma subpopulação, no final de cada geração, espera que as outras subpopulações enviem seus melhores indivíduos e também espera que as outras subpopulações recebam os seus melhores indivíduos. No modo assíncrono, os melhores indivíduos são enviados

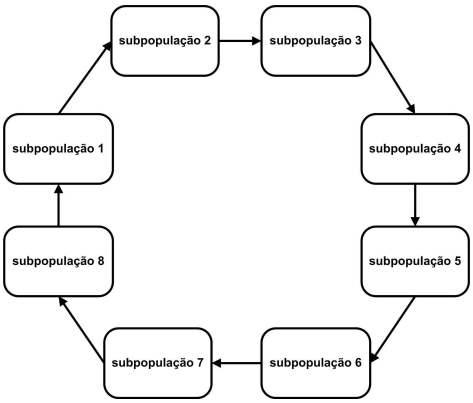


Figura 64: Topologia de migração *ring*

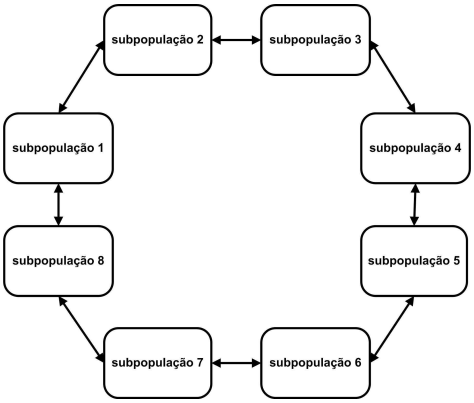


Figura 65: Topologia de migração *neighborhood*

para as subpopulações remotas e cada subpopulação continua a ser processada, não importando se as outras subpopulações receberam esses indivíduos ou não. As subpopulações não precisam esperar o recebimento de indivíduos para continuarem a ser processadas. As subpopulações simplesmente verificam se existem ou não indivíduos para serem recebidos em um *pool* intermediário; se existem, eles são recebidos. Na prática, a despeito de ser mais rápido que o modo síncrono, o modo assíncrono não é eficiente em termos de impedir a convergência prematura. A razão disso é que as diferentes velocidades de execução das instâncias do algoritmo genético paralelo nos processadores impedem uma migração consistente (HOMAYOUNFAR; AREIBI; WANG, 2003).

Há três fatores importantes em um algoritmo genético de granularidade grossa: a topologia de migração, que define as conexões entre as subpopulações; o intervalo de migração, que define o intervalo entre migrações; e a taxa de migração, que define quantos indivíduos irão migrar (CANTU-PAZ, 1995) (SILVA, 2005). A escolha dos melhores valores para esses parâmetros é fundamental para otimizar a eficiência do algoritmo genético paralelo. De outra forma,

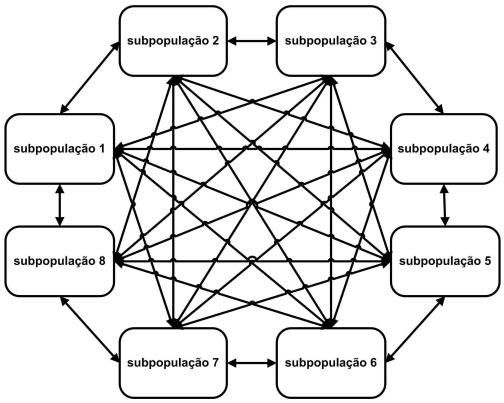


Figura 66: Topologia de migração *broadcast*

muitas migrações de muitos indivíduos por migração podem levar a convergência prematura. Isto significa que uma subpopulação força a outra a convergir para o mesmo ponto de ótimo local. Por outro lado, poucas migrações de poucos indivíduos por migração pode não ter efeito sensível nas subpopulações. Infelizmente, não existem regras para determinar o melhor valor desses parâmetros. Na realidade, a intuição é fortemente recomendada para ajustar a topologia de migração, o intervalo entre migrações e a taxa de migração (HUE, 1997).

4.3 Considerações Finais

Este capítulo apresentou uma introdução aos algoritmos genéticos sequenciais e paralelos. Foram apresentadas a terminologia utilizada, a representação dos parâmetros, os operadores genéticos. No capítulo seguinte será descrita a implementação do algoritmo genético paralelo para a plataforma utilizada.

Capítulo 5

ALGORITMO GENÉTICO PARALELO PARA SISTEMA EMBUTIDO MULTIPROCESSADO

ESSE capítulo trata da implementação de um algoritmo genético paralelo para a plataforma HMPS. A implementação desse algoritmo é apresentada na Seção 5.1 e os resultados e a discussão dos mesmos é apresentada na Seção 5.3.

5.1 Algoritmo Genético Paralelo Embutido

Partindo da abordagem de *granularidade grossa* para algoritmos genéticos paralelos, foi implementado um algoritmo genético paralelo embutido (AGPE) para a plataforma HMPS, utilizando topologias de migração *anel*, *vizinhança* e *broadcast*. Foi escolhida essa abordagem em virtude da mesma ser mais adequada para execução em máquinas MIMD (LIN et al., 1995), que é o caso de sistemas embutidos multiprocessados.

Quando o sistema embutido multiprocessado é inicializado, o *microkernel* do processador mestre obtém os endereços dos processadores escravos, o número máximo de processadores utilizados pela plataforma, o número máximo de tarefas a serem executadas por processador e o número máximo de tarefas utilizadas pela plataforma. Então, o *microkernel* do processador mestre faz a alocação das tarefas do AGPE nos processadores escravos. Cada processador escravo executa somente uma instância do AGPE.

A estrutura do AGPE pode ser explicada pela rede de Petri (AGUILERA, 1989), apresentada na Figura 67, em que *TamPop* corresponde ao tamanho da população e n ao número de processadores. Inicialmente, o *microkernel* do processador mestre da plataforma HMPS executa a função *TasksAllocation*() para alocar uma instância do AGPE em cada processador escravo. Após a alocação das instâncias do AGPE nos processadores escravos, começa a

execução das mesmas. Cada instância do AGPE recebe um identificador, sendo que a instância que recebe o identificador 0 (AG_0) executará a função *PopulaçãoInicial*(), responsável por gerar a população de indivíduos e, depois, a função *EnviaSubPopulaçãoInicial()*, responsável por dividir a população inicial em subpopulações e enviá-las para as demais instâncias. Essas últimas instâncias do AGPE, que receberam os outros identificadores, executarão a função *RecebeSubPopulaçãoInicial()*, responsável por receber a subpopulação inicial enviada pela instância correspondente ao identificador 0. Depois do recebimento da subpopulação, todas as instâncias do AGPE, inclusive a que gerou a população inicial, continuarão a execução do AGPE, segundo os passos seguintes no processo, descrito no Algoritmo 9.

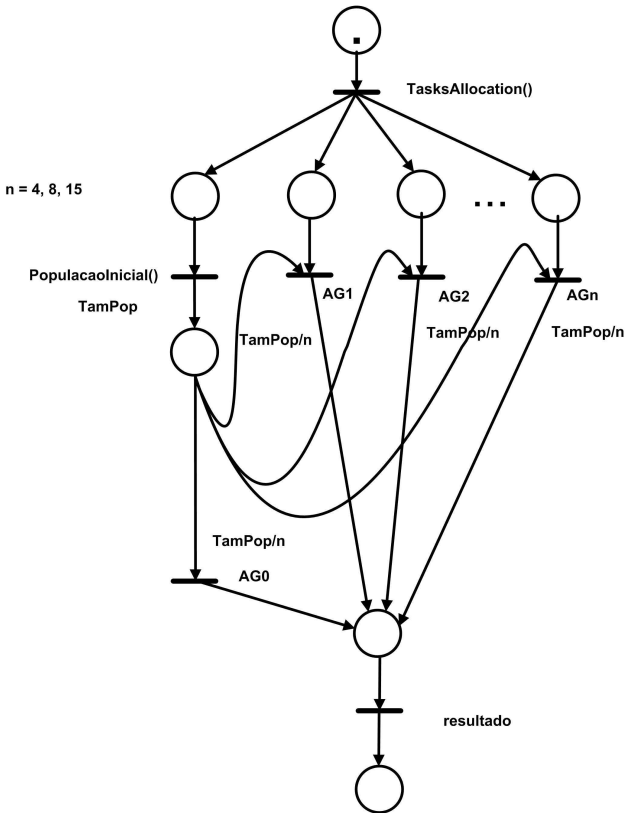


Figura 67: Rede de Petri ilustrando a operação do AGPE

Conforme apresentado na Seção 4.1.6 (do Capítulo 4), o desempenho de um algoritmo genético é fortemente influenciado pela definição dos parâmetros a serem utilizados pelo mesmo. Os parâmetros utilizados pelo AGPE são mostrados na Tabela 12. Esses parâmetros podem ser de dois tipos: global, quando aplicados ao algoritmo genético como um todo e local, quando aplicados somente em uma ilha, conforme apresentado na Seção 4.2.3.3 (do Capítulo 4).

Tabela 12: Parâmetros do AGPE

Variável	Parâmetro	Tipo
$NumProcessos$	Número de processos executados pelo AGPE	Global
$TamPop$	Tamanho da população	
$TamSubPop$	Tamanho da população de uma ilha. É definido internamente como $TamPop/NumProcessos$	
$TopMigracao$	Topologia de migração	
$IntMigracao$	Intervalo de migração	
$TxMigracao$	Taxa de migração	
$Gerao$	Número de gerações	
$NumVariaveis$	Número de variáveis da função objetivo	
$TamVariavel$	Tamanho das variáveis da função objetivo	
$TamCromo$	Tamanho do cromossomo. É definido internamente como $TamVariável \times NumVariáveis$	
$TxCross$	Taxa de cruzamento	Local
$TxMutacao$	Taxa de mutação	

5.2 Algoritmo Genético de uma Ilha

O fluxo de controle de um algoritmo genético paralelo para uma ilha é ilustrado no Algoritmo 9. Uma vez realizada a alocação das instâncias do AGPE, cada processador escravo obtém o identificador da tarefa em execução, através da função *getprocessid*().

Em seguida o AGPE executa a função *inicializa*(), que definirá os valores das variáveis que representam os parâmetros necessários para a configuração do algoritmo genético paralelo. Essa função invoca, no final da sua execução, a função *DefineIntervalo*(), na qual são definidos os limites inferior ($LimInf[NumVariável]$) e superior ($LimInf[NumVariável]$) de cada variável.

Após a execução da função *inicializa*(), o AGPE verifica a variável *NumProcessos* para definir se o mesmo será executado serialmente (quando $NumProcessos = 1$) ou em paralelo (quando $NumProcessos > 1$). Então o AGPE executa a função *PopulaçãoInicial*() para gerar a população inicial.

Depois que a população inicial é gerada, todos os cromossomos dos indivíduos são avaliados pela função *AvaliaçãoPop*(). Essa função invoca a função $DecodificaCromossomo($ *cromossomo*[], $TamCromo)$ que é utilizada para extrair os valores dos genes do cromossomo e os converter para real. Em seguida, esses valores em real são passados para a função *Função_objetivo*(), que retorna o valor da função objetivo para cada cromossomo. Agora, os valores da função objetivo são ordenados pela função *SortPop*() e recebem um índice de aptidão pela função *Ranking*(). Essa função utiliza o método de ordenamento linear para mapear os

valores da função objetivo em índices de aptidão. Em seguida, o AGPE inicia a emulação do ciclo de vida. Quando se iniciar a primeira iteração, a função *Seleção*() seleciona os melhores cromossomos da população inicial, ou seja, os pais, utilizando o método da roleta. Essa função também é responsável por calcular o índice de aptidão acumulado. Em seguida, os pais são combinados para gerar filhos por meio da função *Crossover*() e os filhos podem ou não sofrer mutações quando a função *Mutação*() é executada. A fim de preservar o melhor cromossomo da população anterior, foi empregada estratégia de *elitismo*. Em seguida, os cromossomos dos indivíduos são novamente avaliados, ordenados e recebem um índice de aptidão.

Nesse momento, a condição para a realização de uma migração é verificada. Em caso positivo, a função *Migração*() é executada e, em seguida, os cromossomos dos indivíduos são novamente avaliados, ordenados e recebem um índice de aptidão. Dessa forma, uma nova população será gerada a partir da inicial.

No final da primeira iteração e início da próxima, essa população irá gerar uma nova população. Com isso, ao longo de todo o processo de evolução (da primeira iteração até a última), serão consideradas somente duas populações: a nova, da iteração atual, e a antiga, da iteração anterior.

5.2.1 Codificação dos indivíduos

Duas estruturas de dados são utilizadas pelo AGPE para codificar os indivíduos: uma para representar os indivíduos e outra para os cromossomos.

A estrutura, apresentada na Figura 68, codifica os indivíduos. Cada indivíduo contém o cromossomo (*cromossomo*[MAXCROMO]), o resultado da função objetivo (*FuncObj*) desse cromossomo, a aptidão desse cromossomo (*Aptidão*) e a respectiva aptidão acumulada (*AptidãoAcum*). O parâmetro MAXCROMO define o tamanho máximo do cromossomo em *bits*.

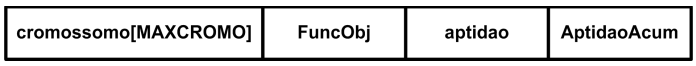


Figura 68: Estrutura do indivíduo

A estrutura, apresentada na Figura 69, consiste em um vetor de *bits*, no qual estão codificadas uma ou mais variáveis, de acordo com a número de argumentos da função objetivo. O parâmetro *TamVariavel* define o tamanho do cromossomo em *bits* e o parâmetro *NumVariáveis* define o número de variáveis. O tamanho do cromossomo é dado por

Algoritmo 9 Algoritmo genético paralelo para uma ilha

1: $local := getprocessid();$
2: **Se** $NumProcessos = 1$ ou $local = 0$ **Então**
3: $PopulaçãoInicial();$
4: **Se** $NumProcessos \neq 1$ **Então**
5: $EnvieSubPopulaçõesIniciais();$
6: **Fim Se**;
7: **Fim Se**;
8: **Se** $local \neq 0$ **Então**
9: $RecebaSubPopulaçãoInicial();$
10: **Fim Se**;
11: $AvaliaçãoPop();$
12: $g := 0;$
13: **Repita**
14: $Seleção();$
15: $Crossover();$
16: $Mutação();$
17: $AvaliaçãoPop();$
18: **Se** $NumProcessos \neq 1$ e critério de migração satisfeito **Então**
19: $Migração();$
20: $AvaliaçãoPop();$
21: **Fim Se**;
22: $g := g + 1;$
23: **Até** critério de parada satisfeito;
24: **Retorna** indivíduo mais apto.

$TamVariável \times NumVariáveis$. Os limites inferior e superior das variáveis são definidos respectivamente pelas variáveis *LimInf[NumVariavel]* e $LimSup[NumVariável]$.

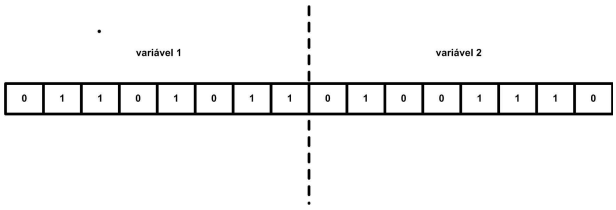


Figura 69: Estrutura do cromossomo com duas variáveis

5.2.2 Migração

O AGPE foi implementado usando vários algoritmos genéticos sequenciais simples, onde a única exceção é o operador de migração, que é empregado em algoritmos genéticos paralelos de granularidade grossa. Esse operador é executado no final de cada geração e é nesse momento que ocorre o sincronismo entre os processadores. Com isso, pode-se avaliar a capacidade do operador de migração de aumentar a *diversidade genética* nas populações que estagnaram.

Algoritmo 10 Função de migração para a comunicação em anel

1: *local* := *getprocessid*();
2: **Se** *local* = 0 **Então**
3: *próximo* := 1;
4: *anterior* := número de tarefas −1;
5: **Fim Se**
6: **Se** *local* > 0 e *local* < número de tarefas −1 **Então**
7: *próximo* := *local* + 1;
8: *anterior* := *local* − 1;
9: **Fim Se**
10: **Se** *local* = número de tarefas −1 **Então**
11: *próximo* := 0;
12: *anterior* := *local* − 1;
13: **Fim Se**
14: **Envie** os melhores indivíduos para a tarefa cujo identificador é *próximo*;
15: **Receba** os melhores indivíduos da tarefa cujo identificador é *anterior*;

A migração é a função mais importante do AGPE, devido ao fato de implementar a comunicação entre as ilhas. As variáveis que influenciam no comportamento do operador de migração são: *local*, que informa para o operador de migração o identificador da tarefa do AGPE que está sendo executada; *IntMigração*, que define o intervalo de tempo em gerações entre as migrações; *TopMigração*, que define a topologia de migração utilizada e *TxMigração*, que define quantos indivíduos irão migrar.

Na topologia de migração *anel*, ocorre comunicação entre duas ilhas quando uma migração é realizada. Para esta estratégia de comunicação, a migração é realizada conforme o procedimento descrito no Algoritmo 10.

Na topologia de migração *vizinhança*, ocorre comunicação entre três ilhas quando uma migração é realizada. Para esta estratégia de comunicação, a migração é realizada conforme o procedimento descrito no Algoritmo 11.

Na topologia de migração *broadcast*, ocorre comunicação entre todas as ilhas quando uma migração é realizada. Para essa estratégia de comunicação, a migração é realizada conforme o procedimento descrito no Algoritmo 12.

Pelo fato de envolver um número pequeno de ilhas durante a migração, as topologias de migração *anel* e *vizinhança* proporcionam um rendimento melhor, no que diz respeito ao tempo de comunicação entre as ilhas. Já a topologia de migração *broadcast* envolve todas as ilhas durante a migração, o que acarreta um alto custo de comunicação entre as ilhas.

Inicialmente, no Algoritmo 10, Algoritmo 11 e Algoritmo 12, a chamada da função de sistema *getprocessid*() é executada para obter o identificador da tarefa do AGPE corrente,

Algoritmo 11 Função de migração para a comunicação com a vizinhança

1: *local* := *getprocessid*();
2: **Se** *local* = 0 **Então**
3: *próximo* := 1;
4: *anterior* := número de tarefas −1;
5: **Fim Se**
6: **Se** *local* > 0 e *local* < número de tarefas −1 **Então**
7: *próximo* := *local* + 1;
8: *anterior* := *local* − 1;
9: **Fim Se**
10: **Se** *local* = número de tarefas −1 **Então**
11: *próximo* := 0;
12: *anterior* := *local* − 1;
13: **Fim Se**
14: **Envie** os melhores indivíduos para a tarefa cujo identificador é *anterior*;
15: **Envie** os melhores indivíduos para a tarefa cujo identificador *próximo*;
16: **Receba** os melhores indivíduos para a tarefa cujo identificador é *anterior*;
17: **Receba** os melhores indivíduos para a tarefa cujo identificador *próximo*;

Algoritmo 12 Função de migração para a comunicação em *broadcast*

1: local := *getprocessid*();
2: **Para** i := 0 ... número tarefas −1 **Faça**
3: **Se** *local* ≠ i **Então**
4: **Envie** os melhores indivíduos para a tarefa cujo identificador é i;
5: **Fim Se**
6: **Fim Para**
7: **Para** i := 0 ... número tarefas −1 **Faça**
8: **Se** *local* ≠ i **Então**
9: **Receba** os melhores indivíduos para a tarefa cujo identificador é i;
10: **Fim Se**
11: **Fim Para**

que está sendo executada, e o armazena na variável *local*. Em seguida, nos dois primeiros algoritmos (i.e., *anel* e *vizinhança*), o valor armazenado em *local* é utilizado para definir o valor das variáveis *próximo* e *anterior*. Essas variáveis são utilizadas pelas topologias para localizar os identificadores das tarefas para as quais os indivíduos serão enviados e/ou recebidos.

A função de migração emprega as funções $EnviaMelhorIndividuo(tarefa, individuo)$ e $RecebeMelhorIndividuo(tarefa, individuo)$ para enviar e receber os melhores indivíduos, respectivamente. A primeira função é utilizada para enviar o melhor indivíduo para a rede. Para tal, ela emprega a função $bin2dec(cromossomo[], TamCromo)$, para converter a sequencia de *bits* do cromossomo em um número decimal inteiro positivo que será enviado pela rede, e a função $WritePipe$, para enviar o indivíduo. Do outro lado, a segunda função é utilizada para receber o melhor indivíduo da rede. Para tal, ela emprega a função *ReadPipe*, para receber o

indivíduo da rede, e a função *dec2binnumero*, para converter o número decimal inteiro positivo recebido da rede na sequencia de *bits* que forma o cromossomo.

5.3 Resultados Experimentais

Nesta seção, apresentamos o ambiente de desenvolvimento do AGPE, as simulações realizadas e os resultados obtidos. O desempenho do AGPE é comparado na otimização de funções, considerando 1, 6, 9 ou 16 processadores.

5.3.1 Ambiente de desenvolvimento

O Ambiente de desenvolvimento de *software* para a plataforma HMPS, no qual o AGPE foi desenvolvido, consiste de um computador com o sistema operacional Windows ou Linux e um compilador cruzado para o processador MIPS. Se o sistema operacional utilizado for o Windows, será necessária, também, a utilização do *Cygwin* (CHAMBERLAIN, 2009).

O Cygwin é uma coleção de ferramentas de software livre desenvolvidas originalmente pela Cygnus Solutions e foi adquirido pela Red Hat, de maneira a permitir que o sistema operacional Windows possa, de certa forma, agir como um sistema Unix. Seu principal objetivo é portar softwares que rodam em sistemas operacionais Unix Like (Solaris, Linux, FreeBSD, NetBSD, OpenBSD e outros) para rodarem no sistema operacional Windows por meio de recompilação.

O compilador cruzado é capaz de criar código executável para uma plataforma diferente daquela onde o compilador é executado. Os compiladores cruzados são utilizados para gerar código executável para sistemas embutidos ou múltiplas plataformas. Normalmente, essas plataformas possuem recursos limitados de memória RAM que não permitem abrigar seus próprios compiladores. O objetivo fundamental da utilização de compiladores cruzados é separar o ambiente de desenvolvimento do *software* (computador PC) da plataforma onde o mesmo será executado (HMPS).

Os compiladores cruzados são construídos a partir do código fonte de vários *softwares*. Para construir o compilador cruzado utilizado na plataforma HMPS utilizamos os *softwares* GCC (GNU, 2009b), Binutils (GNU, 2009a) e a biblioteca Newlib (JOHNSTON, 2009). O GCC é uma coleção de *software* livre de compiladores, que pode ser utilizada para a compilação cruzada e possui suporte para muitas plataformas. Já o Binutils é uma coleção de *software* livre de ferramentas de programação utilizadas para a manipulação de código objeto em vários formatos.

Devido ao fato do compilador GCC-MIPS-ELF da plataforma HMPS gerar, sempre que possível, código executável com instruções de ponto flutuante, que não são suportadas pelo processador PLASMA, foi necessário construir um novo compilador cruzado utilizando a biblioteca Newlib, que gera código executável sem essas instruções. O processo de construção do compilador cruzado é realizado pelo código apresentado no Apêndice F.

A biblioteca Newlib é uma implementação da biblioteca C padrão, criada originalmente pela Cygnus Solutions, para sistemas embutidos. É um conglomerado de partes de várias bibliotecas de *software* livre, que facilmente permitem a sua utilização em produtos baseados em sistemas embutidos.

5.3.2 Configurações de simulação

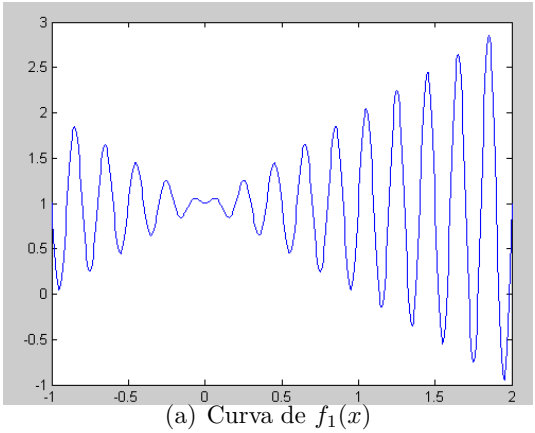
Após a realização das mudanças na plataforma HMPS, da geração do compilador cruzado e do desenvolvimento do AGPE, o mesmo foi executado tanto para a sua validação como também para validação das mudanças realizadas na plataforma. Para a realização das simulações foram escolhidas três funções para serem otimizadas pelo AGPE. Essas funções foram simuladas utilizando 1, 6, 9 ou 16 processadores alocando uma tarefa do AGPE em cada processador escarvo.

5.3.2.1 Funções objetivo

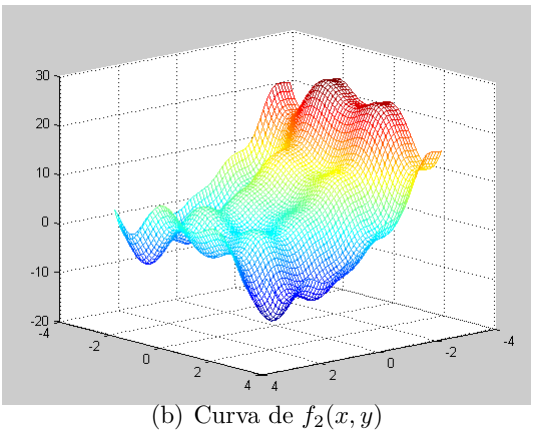
As três funções escolhidas para serem otimizadas pelo AGPE são não-lineares, sendo duas delas multi-modais. A primeira função, $f_1(x)$, é definida na Equação 13 e a curva correspondente é mostrada na Figura 70(a). Essa função, proposta por (LACERDA; CARVALHO, 1999), possui 14 máximos locais e um máximo global no intervalo de interesse [-1, 2], com um máximo global aproximado de $2,83917$ no ponto $x = 1,84705$.

A segunda função, $f_2(x, y)$, é mostrada na Equação 13 e a curva correspondente é mostrada na Figura 70(b). Essa função, proposta nesse trabalho, possui vários mínimos locais e um mínimo global no intervalo de interesse $-3 \leq x \leq 3$ e $-3 \leq y \leq 3$, e um mínimo global aproximado de -12.92393 no ponto $x = 2,36470$ e $y = 2,48235$.

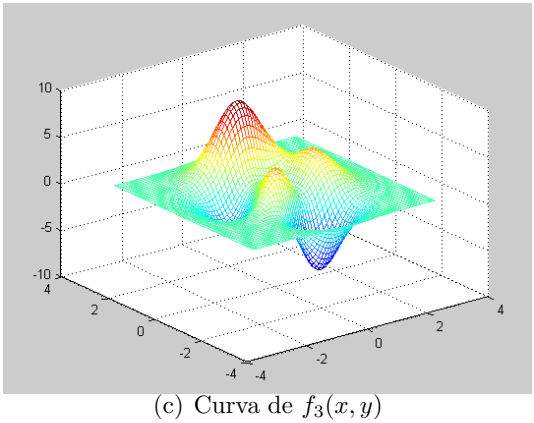
A terceira função, $f_3(x, y)$, é apresentada na Equação 13 a curva correspondente é mostrada na Figura 70(c). Essa função, proposta em (MATHWORKS, 2007), possui 2 máximos locais e um máximo global no intervalo de interesse $-3 \leq x \leq 3$ e $-3 \leq y \leq 3$ e um máximo



(a) Curva de $f_1(x)$



(b) Curva de $f_2(x, y)$



(c) Curva de $f_3(x, y)$

Figura 70: Curvas das funções utilizadas nos processos de otimização

global aproximado de $8,11152$ no ponto $x = 0,01176$ e $y = 1,58823$.

$$\begin{aligned}
\max_x f_1(x) &= sen(10\pi x) + 1 \\
\min_{x,y} f_2(x,y) &= cos(4x) + 3sen(2y) + (y-2)^2 - (y+1) \\
\max_{x,y} f_3(x,y) &= 3(1-x)^2 e^{(-x^2-(y+1)^2)} - 10(\tfrac{x}{5} - x^3 - y^5)e^{(-x^2-y^2)} - \tfrac{1}{3}e^{(-(x+1)^2-y^2)}
\end{aligned} \tag{13}$$

5.3.2.2 Configuração da plataforma e do AGPE

As tarefas do AGPE são alocadas em cada processador escravo de forma sequencial formando um anel. Esta forma de alocação foi utilizada tendo em vista facilitar a execução do AGPE utilizando as topologias de migração *anel* e *vizinhança*. A Figura 71(a), Figura 71(b) e Figura 71(c) ilustram a alocação das tarefas na plataforma HMPS.

Os valores dos parâmetros de configuração da plataforma HMPS, conforme descrito na Seção 2.3.4 (do Capítulo 2), e os valores dos parâmetros do AGPE, utilizados nas simulações, são mostrados na parte esquerda e direita da Tabela 13, respectivamente. Note que os parâmetros MAX_X e MAX_Y indicam os valores máximos para as três configurações da rede

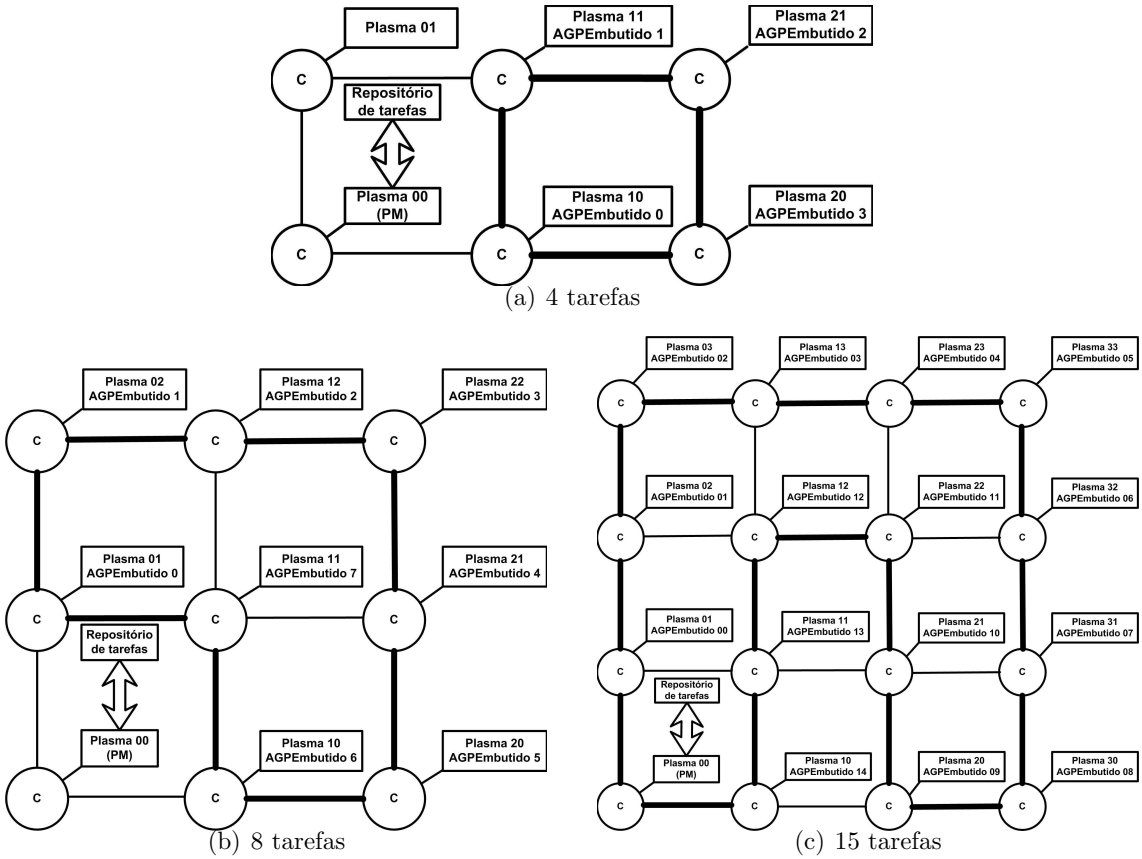


Figura 71: Alocação das tarefas do AGPE na plataforma HMPS

intrachip: 2×3, 3×3 e 4×4. O tamanho da memória é de 1 MB, sendo que cada página ocupa 256 KB. O número de gerações efetuadas é 40, quando um único processador é utilizado, e 12, nos outros casos. O número de variáveis é 1, para a função f_1, e 2 para as funções f_2 e f_3.

5.3.2.3 Métricas de desempenho do AGPE

O desempenho de um algoritmo genético paralelo pode ser avaliado pelos valores de *speedup* e eficiência. O *speedup* S_p (CHIWIACOWSKY et al., 1980) é definido conforme a Equação 14, onde T_1 é o tempo de processamento da versão sequencial do algoritmo genético e T_p é o tempo de processamento da versão paralela executada por p processadores.

$$S_p = \frac{T_1}{T_p} \tag{14}$$

A eficiência E_p (CHIWIACOWSKY et al., 1980) é definida conforme a Equação 15, onde $\frac{1}{p} < E_p \leq 1$, sendo p o número de processadores empregados.

$$E_p = \frac{S_p}{p} \tag{15}$$

Tabela 13: Configuração dos parâmetros da Plataforma HMPS e do AGPE

Parâmetro HMPS	Valor
TAM_FLIT	16
TAM_BUFFER	8
TAM_NI_FLIT	16
TAM_NI_BUFFER	16
MAX_X	2/2/3
MAX_Y	1/2/3
TAM_PAGINA	18
TAM_MEMORIA	20
MASTERADDRESS_X	0
MASTERADDRESS_Y	0
PAGE_SIZE	0x40000
MASTERADDRESS	x00
MAXLOCALTASKS	3
MAXGLOBALTASKS	30
MAXPIPE	128
MAXMSG	128

Parâmetro AGPE	Valor
Tamanho da população	240
Número de gerações	40/12
Taxa de cruzamento	0.8
Taxa de mutação	0.05
Tamanho da variável	8 *bits*
Número de variáveis	1/2
Limite inferior de x	$-1/-3$
Limite superior de x	$2/-3$
Limite inferior de y	-3
Limite superior de y	3

O tempo consumido pelas simulações foi obtido por sucessivas execuções da chamada de sistema $GetTick()$ do *microkernel*.

5.3.3 Resultados de simulação

O objetivo da realização das simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$, na execução de uma instância do AGPE por processador escravo utilizando 1, 6, 9 ou 16 processadores, é a obtenção dos valores de tempo consumido para alcançar a solução da função objetivo, o *speedup* e a eficiência. Os resultados obtidos foram organizados por topologia de migração. Em seguida, esses resultados são apresentados, discutidos e comparados.

Todas as 33 otimizações da função $f_1(x)$ efetuadas resultaram no valor ótimo aproximado $2,83917$. Similarmente ao caso da função $f_1(x)$, todas as simulações realizadas para a otimização das funções $f_2(x, y)$ e $f_3(x, y)$ obtiveram os valores ótimos aproximados de -12.92393 e $8,111521$ respectivamente.

5.3.3.1 Comunicação em anel

De acordo com a Seção 2.2.3.1, os dados do *payload* do pacote, antes de serem enviados, são segmentados em duas metades de 16 bits quando o tamanho do *flit* da chave e da interface de rede é de 16 bits, que é a configuração utilizada pela plataforma nas simulações realizadas. Esses dados são segmentados e enviados na transição de descida de `clock_tx` quando o nível

lógico do sinal `tx` é 1 e depois são recebidos e agrupados na transição de descida de `clock_rx` quando o nivel lógico de `rx` é 1.

A Figura 72 mostra o intervalo de tempo onde ocorre a migração de um indivíduo na simulação da função $f_1(x)$, empregando a topologia de migração em anel, utilizando 6 processadores e tarefas alocadas conforme mostrado na Figura 71(a). O processador 10 envia os dados pela porta `data_out`. O primeiro *flit* contém o endereço do destino(0x0010), o segundo *flit* contém o tamanho do *payload* (0x0012), o terceiro e quarto *flits* contém o identificador do serviço DELIVER_MESSAGE (0x0000, 0x0020), o quinto e sexto contém endereço do processador que está enviando o indivíduo (0x0000, 0x0010), o sétimo e oitavo contém o identificador da tarefa de destino do indivíduo (0x0000, 0x0001), o nono e décimo contém o identificador da tarefa de origem do indivíduo (0x0000, 0x0000), o décimo primeiro e décimo segundo contém o tamanho do indivíduo (0x0000, 0x0004). O décimo terceiro e décimo quarto *flits* contém o cromossomo (0x0000, 0x00f3), o décimo quinto e décimo sexto contém o valor da função objetivo (0x0000, 0x0002), o décimo sétimo e décimo oitavo contém o valor da aptidão (0x0000, 0x0002) e, finalmente, o décimo nono e vigésimo contém o valor da aptidão acumulada (0x0000, 0x0002).

No intervalo de tempo em que ocorre a migração de um indivíduo, as interrupções são desabilitadas, voltando a ser habilitadas no fim do processo. O indivíduo é recebido na porta `data_in` do processador 11, e quando a fila da chave está cheia ou quando a recepção terminou é gerada uma interrupção para sinalizar ao processador 11 para ler o indivíduo.

A Tabela 14, Tabela 15 e Tabela 16 mostram os resultados das simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ pelo AGPE configurado para utilizar a topologia de migração em anel. Dessas tabelas são obtidos gráficos do *speedup* e eficiência que são mostrados na Figura 73. Os dados dessas figuras são organizados em triplas formadas pelo número de processadores escravo usados, a taxa e o intervalo de migração impostos.

O comportamento de $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ pode ser analisado mantendo constante a taxa de migração e variando o intervalo de migração. Nesse caso, se a diminuição do intervalo de migração resultou em uma melhora do *speedup* e da eficiência, podemos admitir que a aptidão dos indivíduos, recebidos por uma ou mais populações na fase de migração, acelerou o processo evolutivo das mesmas, diminuindo o tempo de convergência, o que pode ser observado nas triplas (4,1,1) e (4,1,2) da Figura 73(a)(b), (15,1,1) e (15,1,2) da Figura 73(c)(d) e (8,1,1) e (8,1,2) da Figura 73(e)(f). Entretanto, se a diminuição do intervalo de migração resultou em uma piora do *speedup* e da eficiência, podemos admitir que a aptidão desses indivíduos não

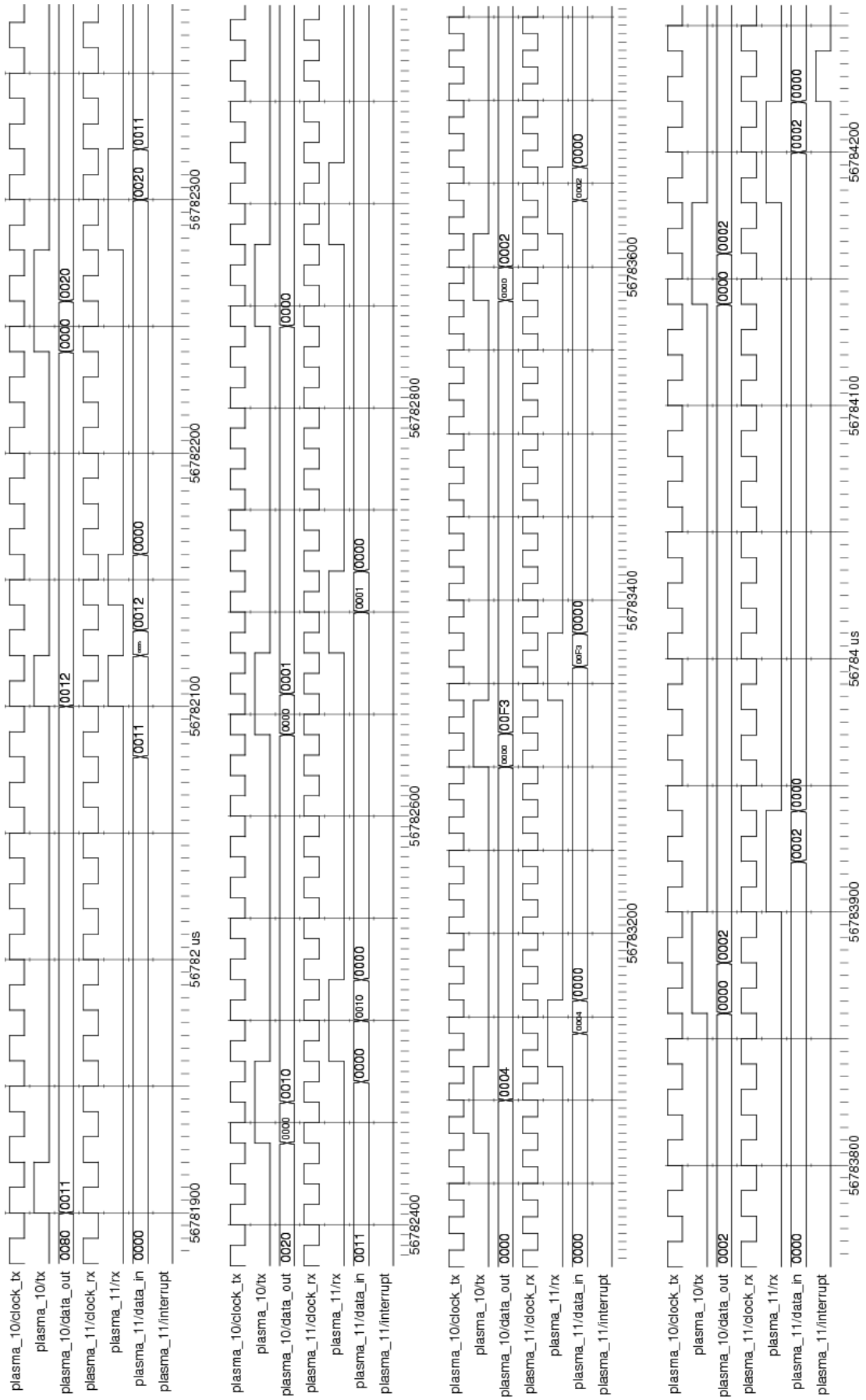


Figura 72: Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em anel

Tabela 14: Resultados de otimização da função $f_1(x)$ para a comunicação em anel

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	1127,5724	1	1
6	1	1	168,57284	6,68893	1,67223
		2	298,76094	3,77416	0,94354
	2	1	650,70556	1,73284	0,43321
		2	267,11808	4,22125	1,05531
9	1	1	112,10709	10,05799	1,25724
		2	102,16839	11,03641	1,37955
	2	1	381,15057	2,95834	0,36979
		2	101,16498	11,14587	1,39323
16	1	1	83,86655	13,44484	0,89632
		2	75,29244	14,97590	0,998393
	2	1	73,95938	15,24583	1,01638
		2	77,13687	14,61781	0,97452

Tabela 15: Resultados de otimização da função $f_2(x, y)$ para a comunicação em anel

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6024,11201	1	1
6	1	1	2569,06697	2,344863	0,586215
		2	2616,76305	2,302123	0,575530
	2	1	2507,48402	2,402452	0,600613
		2	1448,84485	4,157872	1,039468
9	1	1	1968,24989	3,06064	0,38258
		2	1250,55945	4,81713	0,60214
	2	1	1352,18413	4,45509	0,55688
		2	1112,49588	5,41495	0,67686
16	1	1	718,73197	8,38158	0,55877
		2	797,31202	7,55552	0,50370
	2	1	596,06991	10,10638	0,67375
		2	866,79268	6,94988	0,46332

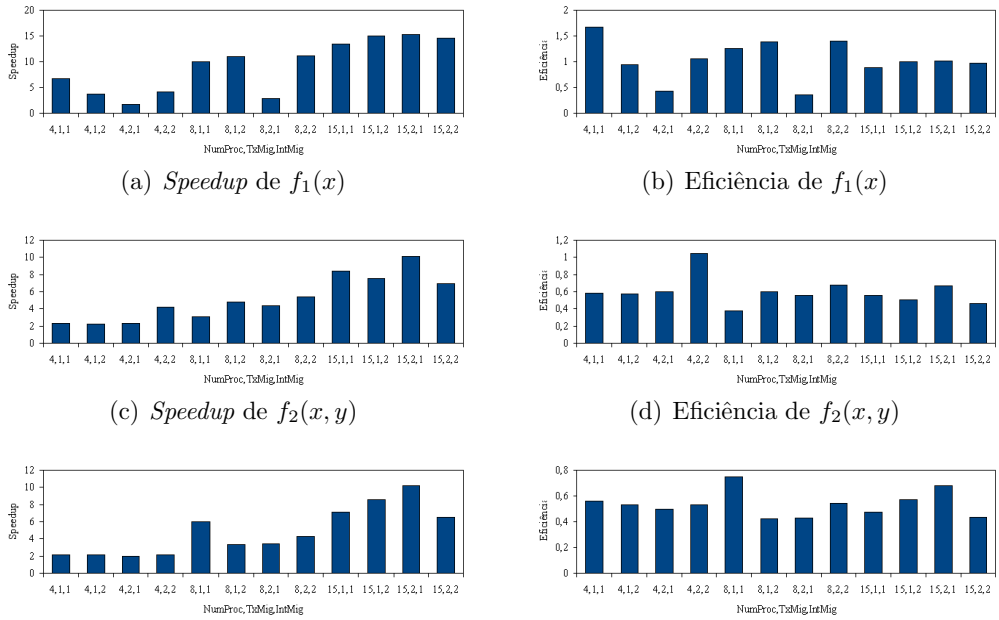
Tabela 16: Resultados de otimização da função $f_3(x, y)$ para a comunicação em anel

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6209,50022	1	1
6	1	1	2778,47764	2,23485	0,55871
		2	2927,64913	2,12098	0,53024
	2	1	3143,09053	1,97560	0,49390
		2	2925,58322	2,12248	0,53062
9	1	1	1037,66721	5,98409	0,74801
		2	1832,88554	3,38782	0,42347
	2	1	1799,06522	3,45151	0,43143
		2	1433,94829	4,33035	0,54129
16	1	1	873,31097	7,11029	0,47401
		2	723,58761	8,58154	0,57210
	2	1	607,38299	10,22336	0,68155
		2	942,71555	6,58682	0,43912

influenciou suficientemente o processo evolutivo das populações que os receberam. Então, o tempo de convergência não diminui, o que pode ser observado nas triplas (15,1,1) e (15,1,2) da Figura 73(a)(b), (8,2,1) e (8,2,2) da Figura 73(c)(d) e (8,2,1) e (8,2,2) da Figura 73(e)(f).

O comportamento de $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ pode ser analisado, também, mantendo constante o intervalo de migração e variando a taxa de migração. Nesse caso, se o aumento da taxa de migração resultou em uma melhora do *speedup* e da eficiência, podemos admitir que a aptidão dos indivíduos, recebidos por uma ou mais populações na fase de migração, acelerou o processo evolutivo das mesmas, diminuindo o tempo de convergência, o que pode ser observado nas triplas (15,1,1) e (15,2,1) da Figura 73(a)(b), (8,1,1) e (8,2,1) da Figura 73(c)(d) e (8,1,2) (8,2,2) da Figura 73(e)(f). Entretanto, se o aumento da taxa de migração resultou em uma piora do *speedup* e da eficiência, podemos admitir que a aptidão desses indivíduos não influenciou suficientemente o processo evolutivo das populações que os receberam. Então, o tempo de convergência aumenta, o que pode ser observado nas triplas (8,1,1) e (8,2,1) da Figura 73(a)(b), (15,1,2) e (15,2,2) da Figura 73(c)(d) e (8,1,1) e (8,2,1) da Figura 73(e)(f).

A topologia de migração em anel é a que possui menor custo de comunicação. Uma população envia o(s) seu(s) melhor(es) indivíduo(s) para a seguinte e recebe o(s) melhor(es) indivíduo(s) da anterior. Devido a esse fato, um indivíduo de alta aptidão é propagado apenas para a população seguinte do AGPE. Não foi observado congestionamento da rede intrachip utilizando essa topologia de migração.



(a) *Speedup* de $f_1(x)$ (b) Eficiência de $f_1(x)$

(c) *Speedup* de $f_2(x, y)$ (d) Eficiência de $f_2(x, y)$

(e) *Speedup* de $f_3(x, y)$ (f) Eficiência de $f_3(x, y)$

Figura 73: Impacto da taxa e intervalo de migração no *speedup* e eficiência considerando a topologia de migração em anel

5.3.3.2 Comunicação com vizinhança

As Figuras 74 e 75 mostram o intervalo de tempo onde ocorre a migração de um indivíduo na simulação da função $f_1(x)$, empregando a topologia de migração em vizinhança, utilizando 6 processadores e tarefas alocadas conforme mostrado na Figura 71(a). O processo é idêntico do que ocorre na topologia de migração em anel. Entretanto, O processador 10 enviará seu melhor indivíduo para os processadores 20 e 11.

A Tabela 17, Tabela 18 e Tabela 19 mostram os resultados das simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ pelo AGPE configurado para utilizar a topologia de migração em vizinhança. Dessas tabelas são obtidos gráficos do *speedup* e eficiência, mostrados na Figura 76. Como anteriormente, os dados dessas figuras são organizados em triplas formadas pelo número de processadores utilizados, a taxa e o intervalo de migração impostos.

A análise do comportamento de $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$, realizada para a topologia de migração em anel, também pode ser aplicada na topologia de migração em vizinhança. Essa topologia de migração possui custo de comunicação maior que na topologia de migração em anel. Uma população envia seus melhores indivíduos para a anterior e a seguinte, e recebe o(s) melhor(es) indivíduo(s) da anterior e da seguinte. Devido a esse fato, pode levar um número de gerações menor que na topologia em anel para que um indivíduo de alta aptidão seja propagado

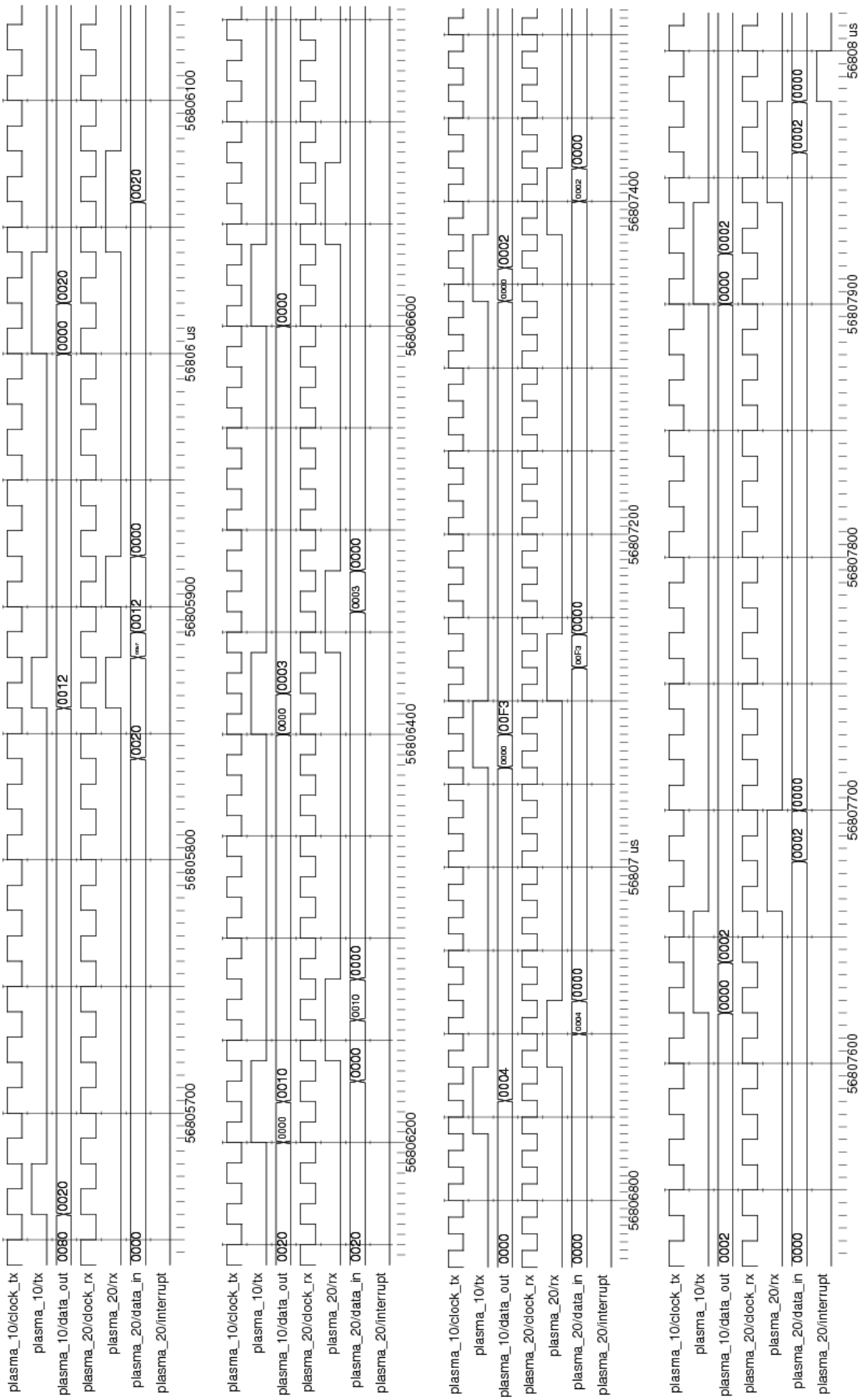


Figura 74: Migração de indivíduo do processador 10 para o 20 utilizando a comunicação em vizinhança

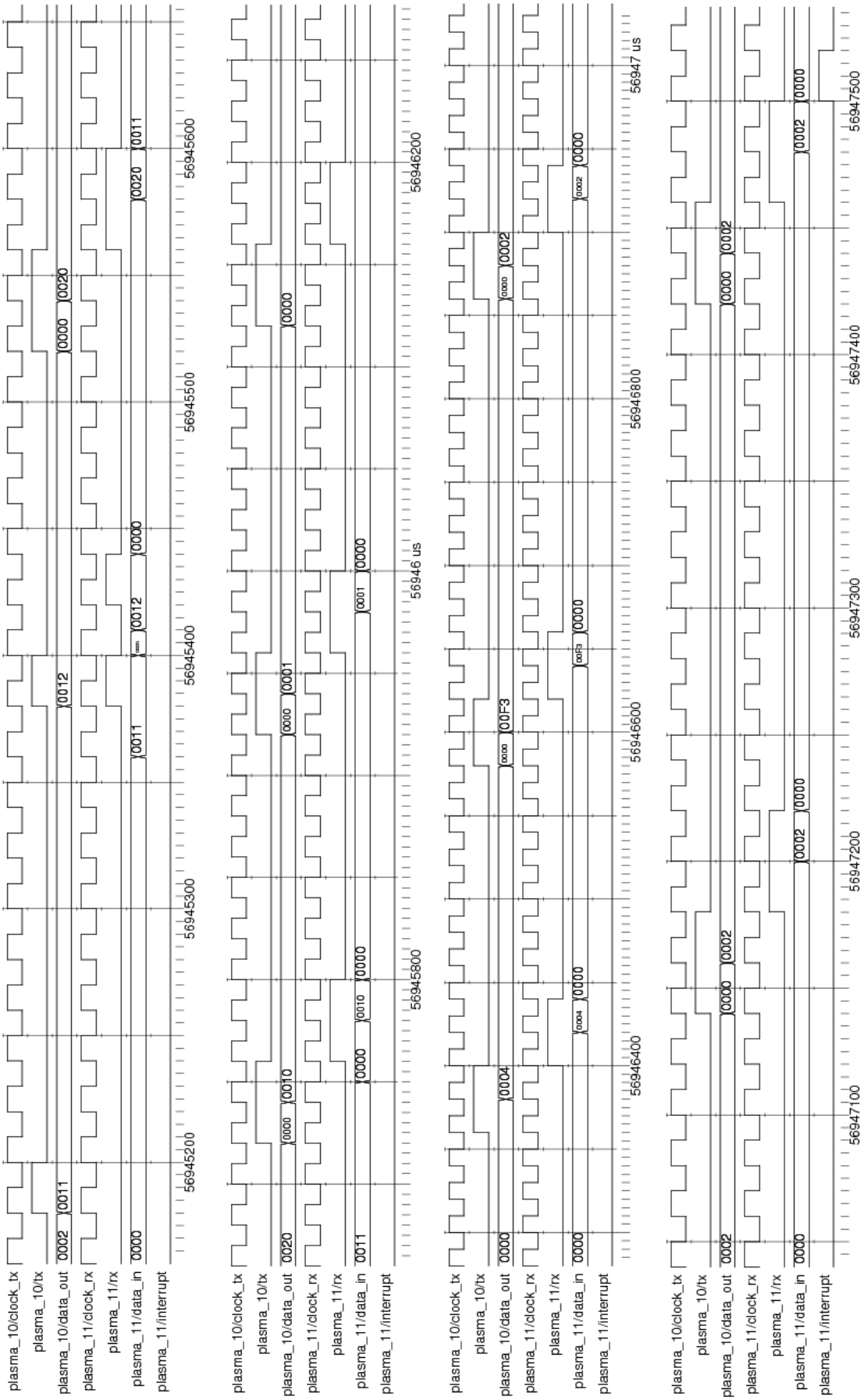


Figura 75: Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em vizinhança

Tabela 17: Resultados de otimização da função $f_1(x)$ para a comunicação em vizinhança

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	1127,5724	1	1
6	1	1	645,36593	1,74718	0,43679
		2	535,14461	2,10704	0,52676
	2	1	172,29855	6,54429	1,63607
		2	172,80265	6,52520	1,63130
9	1	1	217,88489	5,17508	0,64688
		2	304,68098	3,70082	0,46260
	2	1	104,90308	10,74870	1,34358
		2	188,31056	5,98783	0,74847
16	1	1	80,62822	13,98483	0,93232
		2	121,45834	9,28361	0,61890
	2	1	71,09218	15,86070	1,05738
		2	131,73707	8,55926	0,57061

Tabela 18: Resultados de otimização da função $f_2(x, y)$ para a comunicação com a vizinhança

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6024,11201	1	1
6	1	1	2970,49815	2,02798	0,50699
		2	2241,80203	2,68717	0,67179
	2	1	2977,43556	2,02325	0,50581
		2	2635,64829	2,28562	0,57140
9	1	1	1560,87682	3,85944	0,48243
		2	1370,53135	4,39545	0,54943
	2	1	1772,67139	3,39832	0,42479
		2	1161,60725	5,18601	0,64825
16	1	1	719,73603	8,36989	0,55799
		2	951,55986	6,33077	0,42205
	2	1	574,84260	10,47958	0,69863
		2	700,59551	8,59855	0,57323

Tabela 19: Resultados de otimização da função $f_3(x, y)$ para a comunicação com a vizinhança

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6209,50022	1	1
6	1	1	2534,68066	2,44981	0,61245
		2	2497,41481	2,48637	0,62159
	2	1	3075,95908	2,01872	0,50468
		2	2737,40887	2,26838	0,56709
9	1	1	1698,95341	3,65489	0,45686
		2	1398,89571	4,43885	0,55485
	2	1	830,546335	7,47640	0,93455
		2	1296,38967	4,78984	0,59873
16	1	1	1235,58877	5,02553	0,33503
		2	910,60102	6,81912	0,45460
	2	1	777,34866	7,98805	0,53253
		2	683,45716	9,08542	0,60569

para todas as populações do AGPE. Também não foi detectada, nas simulações, ocorrência de congestionamento utilizando esta topologia de rede.

5.3.3.3 Comunicação em broadcast

As Figuras 77, 78 e 79 mostram o intervalo de tempo onde ocorre a migração de um indivíduo na simulação da função $f_1(x)$, empregando a topologia de migração em *broadcast*, utilizando 6 processadores e tarefas alocadas conforme mostrado na Figura 71(a). O processo é idêntico do que ocorre na topologia de migração em anel. Entretanto, O processador 10 enviará seu melhor indivíduo para os processadores 11, 21 e 20.

A Tabela 20, Tabela 21 e Tabela 22 mostram os resultados das simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ pelo AGPE configurado para utilizar a topologia de migração em *broadcast*. Dessas tabelas são obtidos gráficos do *speedup* e eficiência, mostrados na Figura 80. Da análise dos resultados também não podemos determinar facilmente os valores da taxa e intervalo de migração para os quais serão obtidos os valores máximos de *speedup* e eficiência. Observe que, no caso dessa topologia, as simulações realizadas se restringiram a 6 e 9 processadores somente.

A análise do comportamento de $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$, realizada para a topologia de migração em anel, também pode ser aplicada na topologia de migração em *broadcast*. Essa topologia de migração possui custo de comunicação maior que na topologia de migração em vizinhança. Uma população envia seus melhores indivíduo(s) para todas as outras e recebe

Tabela 20: Resultados de otimização da função $f_1(x)$ para a comunicação em *broadcast*

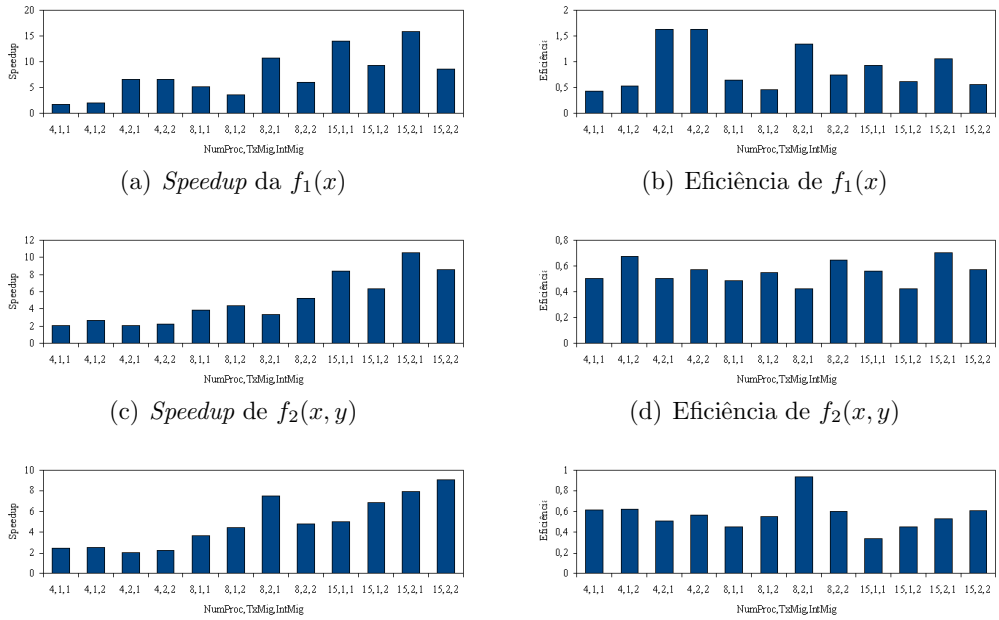
Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	1127,5724	1	1
6	1	1	428,61639	2,63072	0,65768
		2	168,11060	6,70732	1,67683
	2	1	174,61342	6,45753	1,61438
		2	167,56867	6,72901	1,68225
9	1	1	99,55644	11,32596	1,41574
		2	133,05054	8,47476	1,05934
	2	1	98,19798	11,48264	1,43533
		2	98,03011	11,50230	1,43778

Tabela 21: Resultados de otimização da função $f_2(x, y)$ para a comunicação em *broadcast*

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6024,11201	1	1
6	1	1	2590,52731	2,32543	0,58135
		2	2704,34186	2,22757	0,55689
	2	1	2567,87804	2,34594	0,58648
		2	2575,99724	2,33855	0,58463
9	1	1	1262,59628	4,77121	0,59640
		2	1063,52475	5,66428	0,70803
	2	1	1278,56308	4,71162	0,58895
		2	1308,16002	4,60502	0,57562

Tabela 22: Resultados de otimização da função $f_3(x, y)$ para a comunicação em *broadcast*

Número de processadores	Taxa de migração	Intervalo de migração	Tempo (ms)	S_p	E_p
1	–	–	6209,50022	1	1
6	1	1	2430,42431	2,55490	0,63872
		2	2401,95947	2,58518	0,64629
	2	1	2956,31811	2,10041	0,52510
		2	2406,31812	2,58049	0,64512
9	1	1	1177,03205	5,27555	0,65944
		2	1200,30208	5,17328	0,64666
	2	1	1192,47639	5,20723	0,65090
		2	1372,70702	4,52354	0,56544



(a) *Speedup* da $f_1(x)$ (b) Eficiência de $f_1(x)$

(c) *Speedup* de $f_2(x, y)$ (d) Eficiência de $f_2(x, y)$

(e) *Speedup* de $f_3(x, y)$ (f) Eficiência de $f_2(x, y)$

Figura 76: Impacto da taxa e intervalo de migração no *speedup* e eficiência, considerando a topologia de migração vizinhança

o(s) melhor(es) indivíduo(s) das outras. Devido a esse fato, um indivíduo de alta aptidão é propagado para todas as populações do AGPE.

Não foi possível obter resultados para avaliação executando o AGPE com 16 processadores, devido à ocorrência de congestionamento da rede intrachip. Vale lembrar que, utilizando essa topologia de migração, no momento em que ocorre uma migração, todas as tarefas do AGPE tentam enviar para depois receber indivíduos das demais. Essa comunicação acontece, aproximadamente, no mesmo intervalo de tempo, causando, então, o congestionamento da rede.

5.3.4 Discussão dos resultados

A Tabela 23, Tabela 24 e Tabela 25, mostram os melhores resultados das simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ em relação ao tempo consumido. Observou-se que, com exceção da simulação com 6 processadores na Tabela 24 e da simulação com 16 processadores na Tabela 25, existe uma tendência a encontrar o valor de otimização desejado utilizando topologias que favorecem a migração de um número grande de indivíduos. Também observou-se que, em cinco dos nove casos mostrados pelas Tabelas 23, 24 e 25, o intervalo de migração é 2, o que pode estar contribuindo para o desenvolvimento de boas características nas populações do AGPE. Finalmente foi observado que, em dois dos nove casos mostrados por essas tabelas, a taxa de

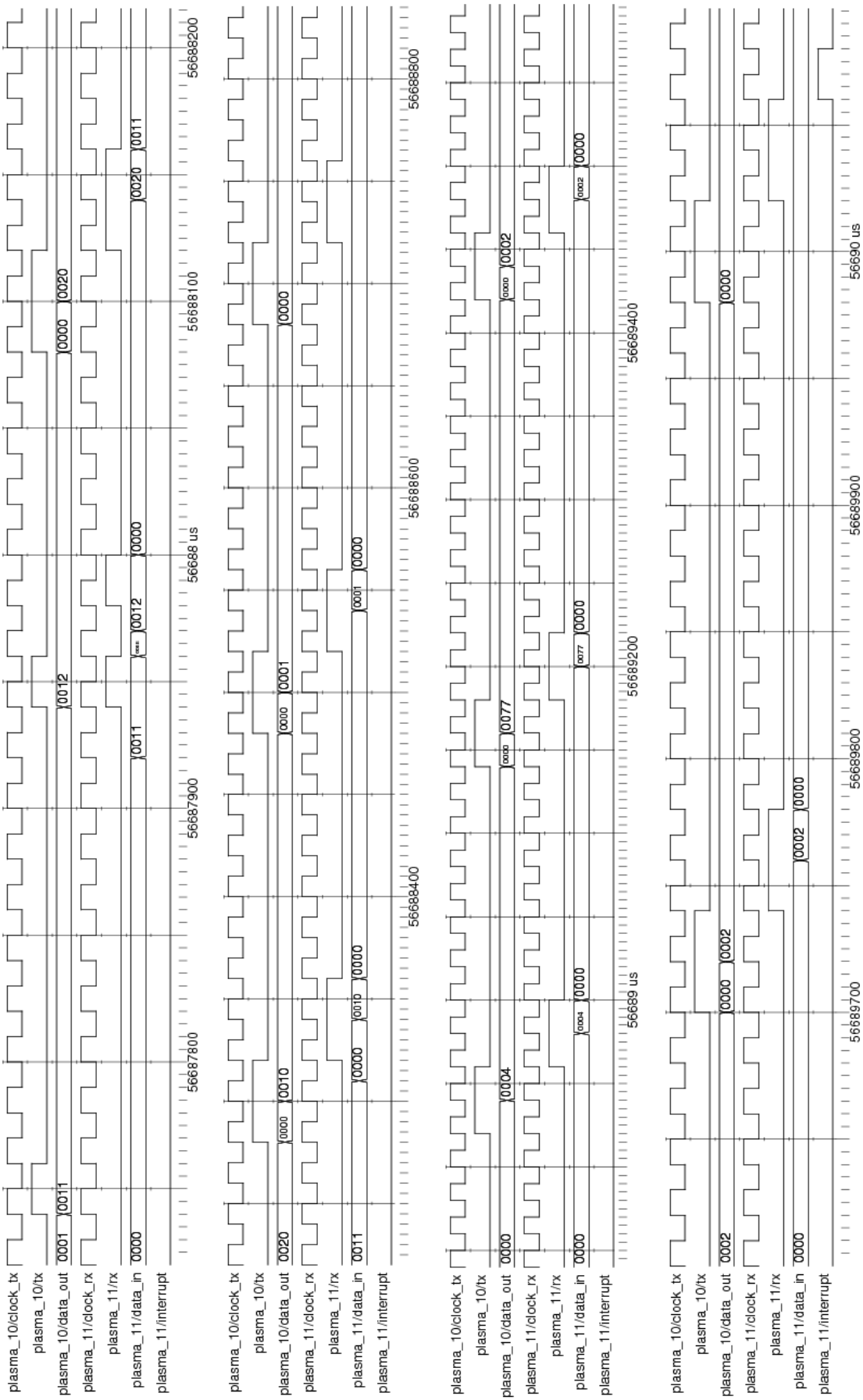


Figura 77: Migração de indivíduo do processador 10 para o 11 utilizando a comunicação em broadcast

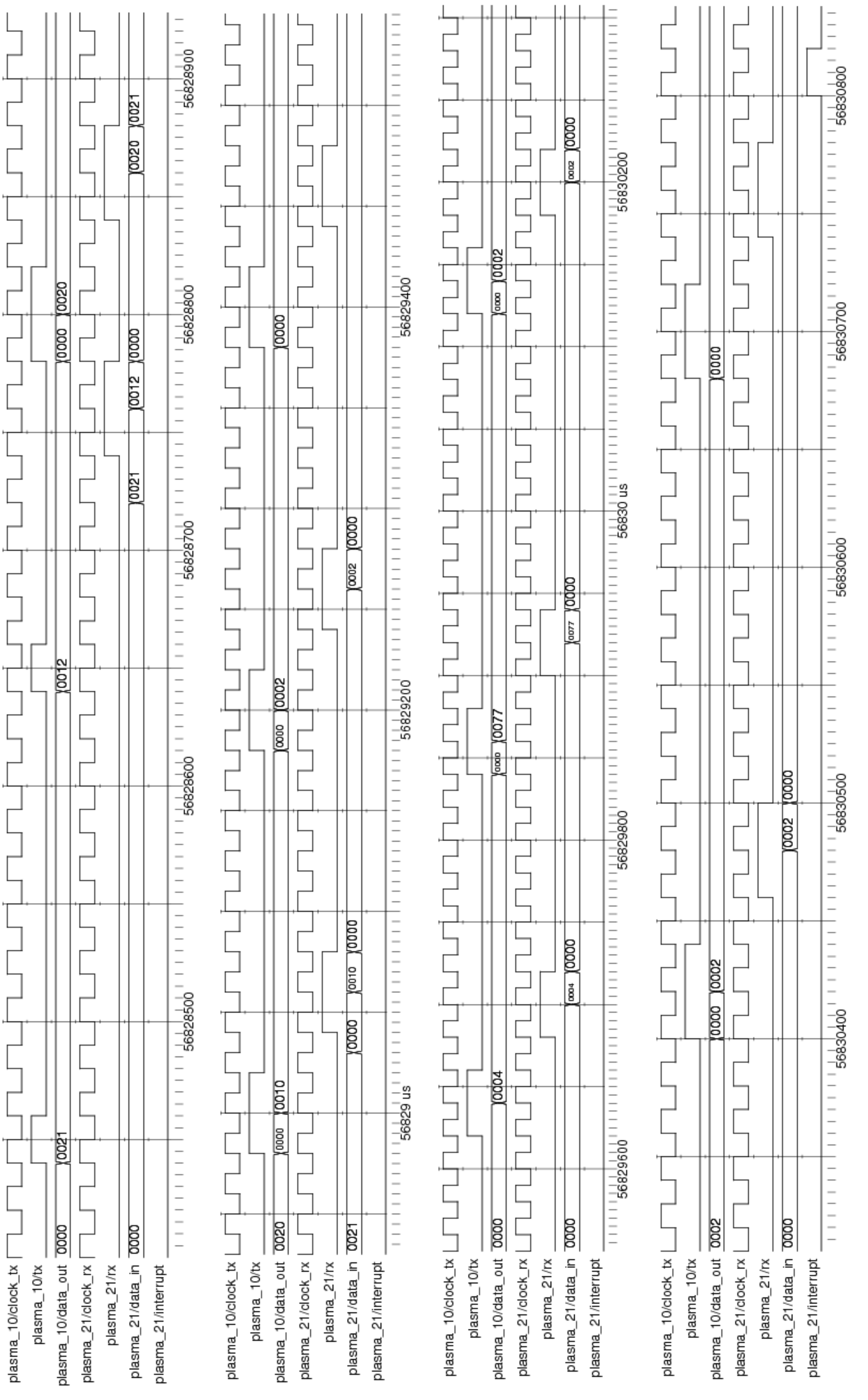


Figura 78: Migração de indivíduo do processador 10 para o 21 utilizando a comunicação em broadcast

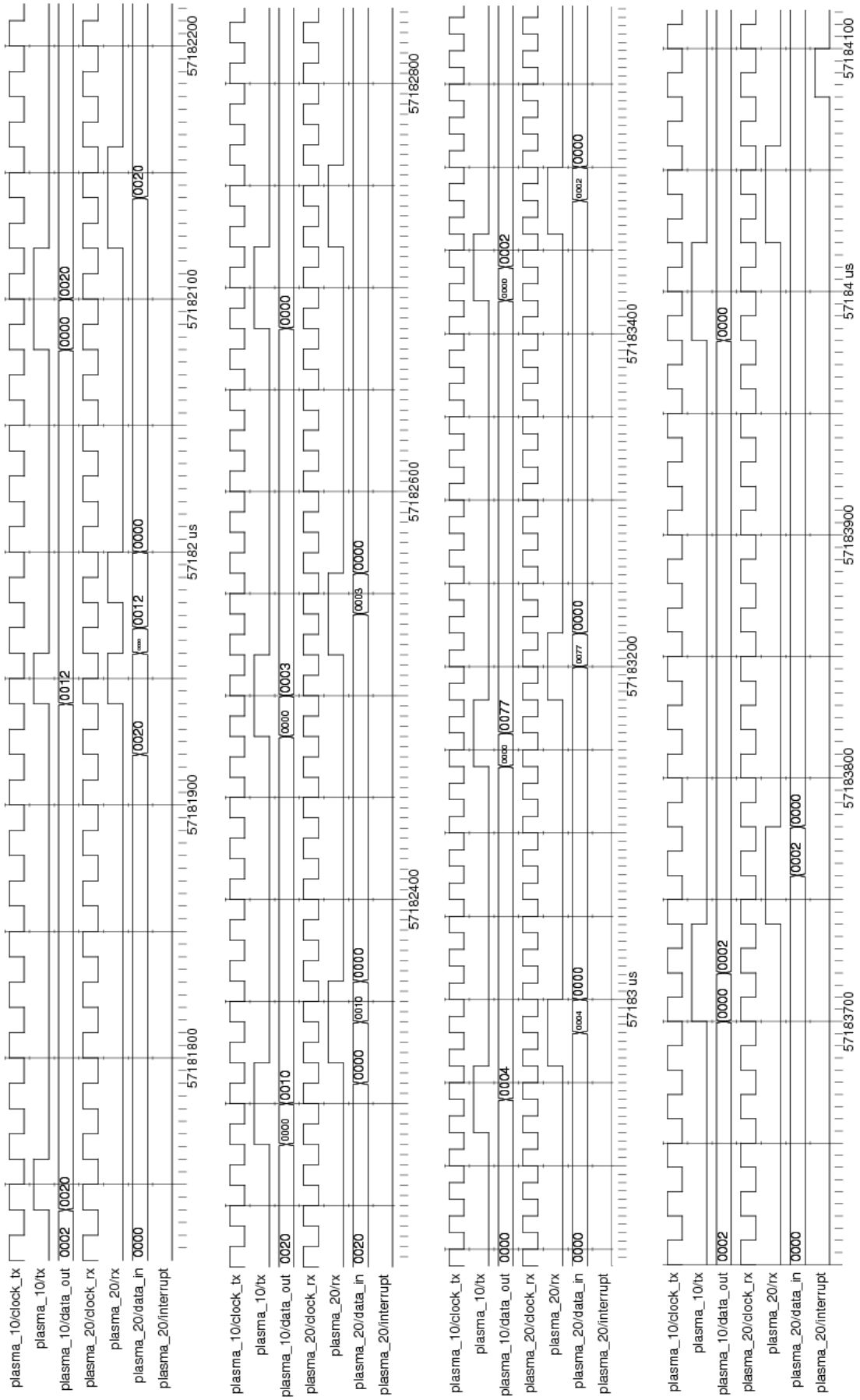
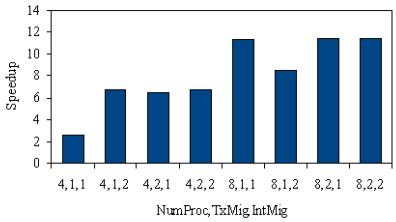
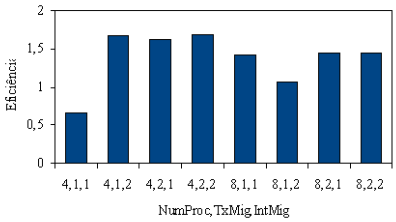


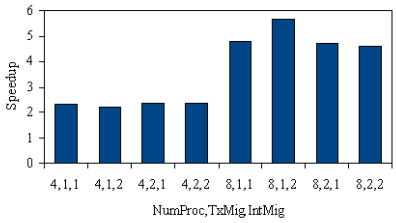
Figura 79: Migração de indivíduo do processador 10 para o 20 utilizando a comunicação em broadcast



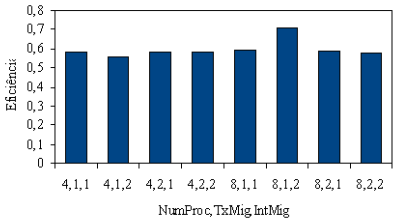
(a) *Speedup* de $f_1(x)$



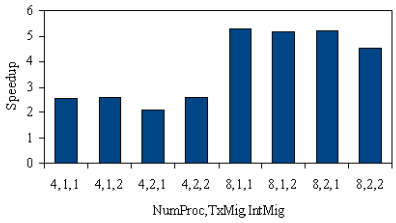
(b) Eficiência de $f_1(x)$



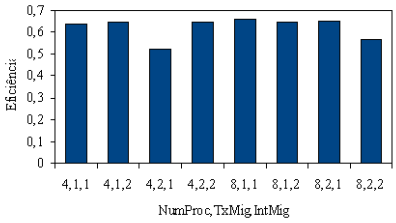
(c) *Speedup* de $f_2(x, y)$



(d) Eficiência de $f_2(x, y)$



(e) *Speedup* de $f_3(x, y)$



(f) Eficiência de $f_3(x, y)$

Figura 80: Impacto da taxa e intervalo de migração no *speedup* e eficiência considerando a topologia de migração em *broadcast*

migração é 1, o que pode estar impedindo uma convergência prematura em um valor de ótimo local.

A Figura 81(a) mostra que o tempo consumido pelas simulações das funções $f_1(x)$, $f_2(x, y)$ e $f_3(x, y)$ diminui à medida que o número de processadores aumenta, o que já era esperado. A Figura 81(b) mostra o *speedup* dessas funções. Valores de *speedup* acima do *speedup* linear, que é igual ao número p de processadores utilizados, podem ser resultado da utilização do paralelismo (CHIWIACOWSKY et al., 1980). Valores de *speedup* abaixo do *speedup*

Tabela 23: Melhores resultados obtidos na otimização da função $f_1(x)$

Número de processadores	Tempo (ms)	S_p	E_p	Topologia de migração	Taxa de migração	Intervalo de migração
6	167,56867	6,72901	1,68225	*broadcast*	2	2
9	98,03011	11,50230	1,43778	*broadcast*	2	2
16	71,09218	15,86070	1,05738	vizinhança	2	1

Tabela 24: Melhores resultados obtidos da na otimização da função $f_2(x, y)$

Número de processadores	Tempo (ms)	S_p	E_p	Topologia de migração	Taxa de migração	Intervalo de migração
6	1448,84485	4,15787	1,03946	anel	2	2
9	1063,52475	5,66428	0,70803	*broadcast*	1	2
16	574,84260	10,47958	0,69863	vizinhança	2	1

Tabela 25: Melhores resultados obtidos na otimização da função $f_3(x, y)$

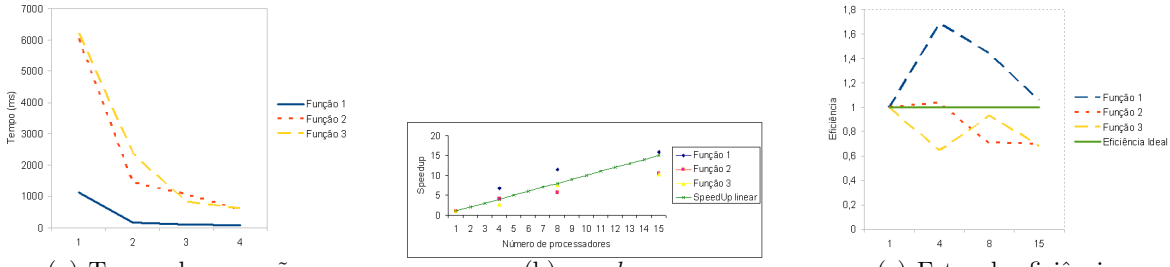
Número de processadores	Tempo (ms)	S_p	E_p	Topologia de migração	Taxa de migração	Intervalo de migração
6	2401,95947	2,58518	0,64629	*broadcast*	1	2
9	830,546335	7,47640	0,93455	vizinhança	2	1
16	607,38299	10,22336	0,68155	anel	2	1

linear podem ser resultado do esforço de comunicação e sincronização entre os processadores (CHIWIACOWSKY et al., 1980). A Figura 81(c) mostra que os valores de eficiência da função $f_3(x, y)$ são menores do que os da função $f_2(x, y)$, que, por sua vez, são menores que os valores da função $f_1(x)$. Vale lembrar que o esforço computacional exigido por $f_3(x, y)$ é maior do que o exigido por $f_2(x, y)$, que, por sua vez, é maior do que o de $f_1(x, y)$.

Os dados da Figura 82, Figura 83 e Figura 84 são organizados em duplas formadas pela pela taxa de migração e pelo intervalo de migração. Para a simulação da função $f_1(x)$, utilizando 2×3 processadores, foi observado que o menor tempo consumido pela simulação foi utilizando a topologia *broadcast* com taxa de migração igual ao intervalo de migração, cujo o valor é 2. Utilizando 3×3, o menor tempo consumido foi também utilizando a topologia *broadcast*, com taxa de migração migração igual ao intervalo de migração, cujo o valor é 2. Utilizando 4×4, o menor tempo consumido foi utilizando a topologia vizinhança, com taxa de migração de 2 e intervalo de migração de 1.

Para a simulação da função $f_2(x, y)$, utilizando 2×3 processadores, foi observado que o menor tempo consumido pela simulação foi utilizando a topologia anel, com taxa de migração igual ao intervalo de migração, cujo o valor é 2. Utilizando 3×3, o menor tempo consumido foi também utilizando a topologia *broadcast*, com taxa de migração de 1 e intervalo de migração, de 2. Utilizando 4×4, o menor tempo consumido foi utilizando a topologia vizinhança, com taxa de migração de 2 e intervalo de migração de 1.

Para a simulação da função $f_3(x, y)$, utilizando 2×3 processadores, o menor tempo consumido foi utilizando a topologia *broadcast*, com taxa de migração de 1 e intervalo de



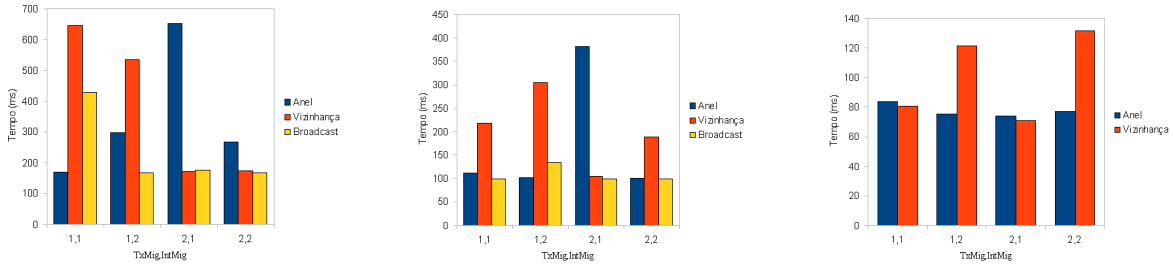
(a) Tempo de execução (b) *speedup* (c) Fator de eficiência

Figura 81: Impacto do número de processadores

migração de 2. Utilizando 3×3, o menor tempo consumido foi utilizando a topologia vizinhança, com taxa de migração de 2 e intervalo de migração de 1. Utilizando 4×4, o menor tempo consumido foi utilizando a topologia anel, com taxa de migração de 1 e intervalo de migração de 1.

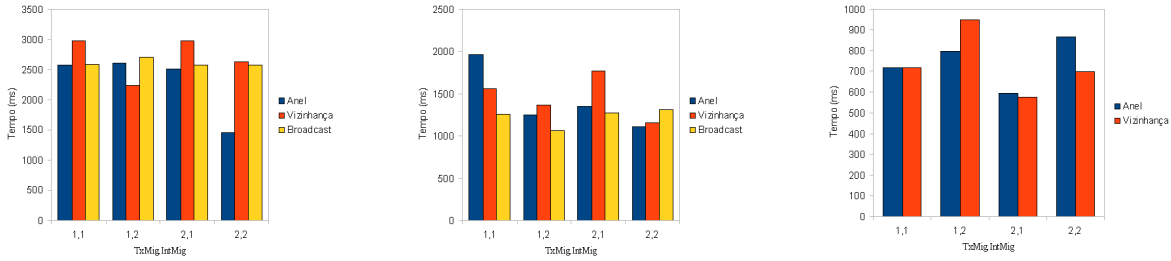
A migração é um instrumento importante para garantir a diversidade genética das populações, como, também, para acelerar mudanças evolucionárias. Entretanto, um intervalo de migração muito pequeno, um número grande de indivíduos migrantes por geração e uma topologia de migração que permita um número grande de indivíduos migrantes por vez pode levar a uma convergência prematura não desejada em um valor de ótimo local (CHIWIACOWSKY et al., 1980).

Apesar dos parâmetros de migração terem sido intensamente estudados (CANTU-PAZ, 1995) (HUE, 1997) (CHIWIACOWSKY et al., 1980), a intuição para ajustá-los ainda é mais utilizada que a análise (CHIWIACOWSKY et al., 1980). A escolha do momento em que deve ser realizada a migração, que indivíduos devem migrar e que topologia de migração deve ser utilizada é bem difícil. Populações pequenas tendem a evoluir rapidamente. Entretanto, as migrações devem ocorrer em um tempo longo o suficiente para permitir o desenvolvimento de boas características em cada população. Além disso, os melhores indivíduos de uma população substituem os piores da(s) populações vizinhas. Por último, a topologia de migração utilizada não deve facilitar a convergência prematura. Com os parâmetros ajustados, os resultados obtidos foram considerados satisfatórios, confirmando que os algoritmos genéticos paralelos são uma ferramenta poderosa para a solução de problemas computacionais difíceis e que o AGPE é uma ferramenta com potencial para ser utilizado na plataforma escolhida ou em derivadas desta.



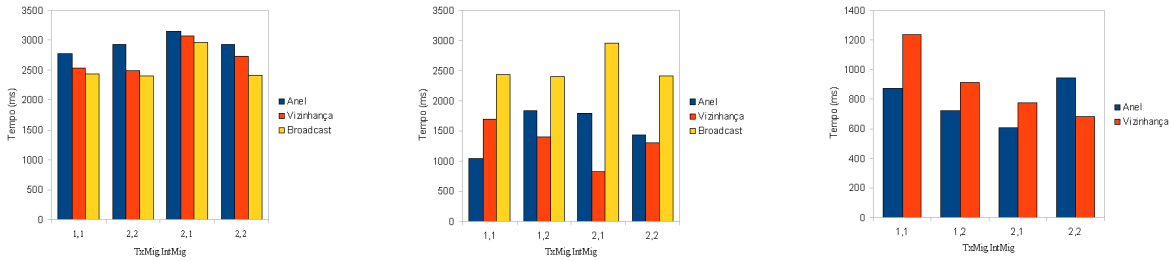
(a) Usando 2×3 processadores (b) Usando 3×3 processadores (c) Usando 4×4 processadores

Figura 82: Impacto da escolha da topologia de migração na otimização de $f_1(x)$



(a) Usando 2×3 processadores (b) Usando 3×3 processadores (c) Usando 4×4 processadores

Figura 83: Impacto da escolha da topologia de migração na otimização de $f_2(x, y)$



(a) Usando 2×3 processadores (b) Usando 3×3 processadores (c) Usando 4×4 processadores

Figura 84: Impacto da escolha da topologia de migração na otimização de $f_3(x, y)$

5.4 Considerações Finais

Este capítulo apresentou a implementação do algoritmo genético paralelo, o AGPE, para a plataforma HMPS. O AGPE provou ser eficiente na busca de soluções para problemas computacionais difíceis. Os resultados encontrados são compatíveis com os esperados. No próximo capítulo são apresentadas as conclusões obtidas nessa dissertação e as direções para trabalhos futuros.

Capítulo 6

CONCLUSÕES E TRABALHOS FUTUROS

ESSA dissertação apresentou um estudo detalhado de sistemas embutidos multiprocessados, sistemas operacionais para esse tipo de plataforma e algoritmos genéticos paralelos. O objetivo desse trabalho foi o desenvolvimento de um algoritmo genético paralelo para uma plataforma baseada em um sistema embutido multiprocessado. Nesse capítulo, introduzimos algumas conclusões alcançadas a partir da análise dos resultados de simulação obtidos e apresentamos algumas propostas para trabalhos futuros.

6.1 Conclusões

Sistemas embutidos multiprocessados são uma tendência no projeto de vários dispositivos eletrônicos, principalmente os da chamada eletrônica de consumo, como telefones celulares, computadores portáteis, televisões digitais. Esses dispositivos são capazes de executar uma variedade aplicações embutidas e são beneficiados pelas vantagens proporcionadas pelo processamento paralelo. Algumas dessas aplicações estão começando a utilizar algoritmos genéticos, o que justifica o desenvolvimento de versões paralelas dos mesmos para sistemas embutidos multiprocessados. Entretanto, quase não existem plataformas de sistemas embutidos multiprocessados completas disponíveis de domínio público para a realização de pesquisas. Por exemplo, há vários trabalhos nas áreas de redes intrachip, processadores embutidos, sistemas operacionais embutidos. Os trabalhos descrevendo um sistema completo são escassos.

A contribuição dessa dissertação está no desenvolvimento de um algoritmo genético paralelo para a plataforma HMPS. Essa plataforma é um sistema embutido multiprocessado completo, composto das chaves, dos processadores e de um *microkernel*. Entretanto possui limitações de *hardware* e de *software* que impedem o desenvolvimento e execução de um algoritmo genético paralelo na mesma tais como o tamanho da rede intrachip e o tamnaho da

página de memória. Inicialmente, o *hardware*, o *software* e o ambiente de desenvolvimento foram modificados para permitir o desenvolvimento e execução do AGPE, como também facilitar o desenvolvimento e execução de futuras aplicações. Várias características foram parametrizadas tais como o tamanho da rede intrachip, com a finalidade de executar AGPE em 4, 9 e 16 processadores; o tamanho da página de memória, com a finalidade de configurar uma página de memória de tamanho suficiente para alojar o AGPE ou outras aplicações; o número de páginas, com a finalidade de executar um número de tarefas por processador maior do que três, que era o valor original. O Apêndice A apresenta os parâmetros de configuração utilizados.

Uma vez a plataforma HMPS melhorada para permitir a implementação da aplicação, o AGPE foi desenvolvido, utilizando a linguagem C e a biblioteca *Newlib*. Através da simulação do modelo da plataforma, alguns resultados foram, então, obtidos. Através desses resultados, concluiu-se que o aumento do número de processadores implicou em um menor tempo de simulação para alcançar o valor de otimização desejado em todas as funções. Foi observado que a topologia que favorece a migração de um número maior de indivíduos pode levar a uma convergência mais rápida. Também foi observado que um intervalo de migração pequeno e uma taxa de migração grande podem levar a uma convergência mais rápida. Entretanto, se a topologia escolhida e os valores do intervalo e taxa de migração estão permitindo um número muito grande de indivíduos migrantes, quando o operador de migração é executado pode levar a uma convergência prematura em um valor de ótimo local. Por outro lado, a escolha de uma topologia que não favoreça a migração de um número grande de indivíduos por vez, um intervalo de migração não muito pequeno, que possa levar ao desenvolvimento de boas características nas populações, e uma taxa de migração não muito alta, podem evitar uma convergência prematura em um valor de ótimo local não desejado

6.2 Trabalhos Futuros

A utilização de algoritmos genéticos paralelos em sistemas embutidos multiprocessados é uma área com várias possibilidades de pesquisa. A seguir, sugerimos alguns temas para trabalhos futuros na área.

A prototipagem da plataforma HMPS, em um dispositivo reconfigurável do tipo FPGA, permitiria avaliar o custo da implementação. O AGPE é um tipo de aplicação que procura explorar o paralelismo oferecido pela plataforma. Dessa forma, poderíamos analisar a relação custo x desempenho entre uma solução sequencial e uma paralela.

Outras topologias de rede, como toroide e hipercubo, poderiam ser exploradas. Isso

implica em modificação das conexões da rede intrachip e do algoritmo de roteamento da chave. Além disso, seria possível pensar em outras topologias de migração e na avaliação do comportamento do AGPE nessas topologias.

Para simplificar o projeto de *hardware*, a chave utilizada pela plataforma HMPS não possui canais virtuais. Com redes grandes, pode haver congestionamento do tráfego na rede, sendo bem vinda a inclusão de canais virtuais na chave, com a finalidade de reduzir esse congestionamento.

O desenvolvimento de uma unidade de ponto flutuante e sua inclusão no Plasma reduziria bastante o tempo consumido por aplicações que fazem uso intenso de cálculos matemáticos complexos, como o AGPE.

A plataforma original oferece uma interface gráfica para o usuário. No entanto, com as modificações feitas, essa interface tornou-se incompatível. Assim sendo, seria interessante o desenvolvimento de uma nova interface.

Uma outra proposta seria o desenvolvimento de um mecanismo de variação dinâmica das taxas de cruzamento e mutação. Isso ajudaria a melhorar a diversidade das subpopulações.

REFERÊNCIAS

ADAMIDIS, P. *Review of parallel genetic algorithms bibliography, Technical report.* Thessaloniki, Greece: [s.n.], 1994.

AGUILERA, L. M. *Ferramenta para geração automática de redes de Petri a partir da especificação de um sistema de software com características tempo real.* Dissertação (Mestrado) — Universidade Estadual de Campinas, Campinas, SP, Brazil, 1989.

ALDEN, H. W. Genetic algorithms for real parameter optimization. In: RAWLINS, G. J. (Ed.). *Foundations of Genetic Algorithms.* [S.l.]: Morgan Kaufmann, 1991. p. 205–218.

ALLAN, R. J.; ANDREWS, S. J.; GUEST, M. F. *High performance computing and Beowulf clusters.* http://www.ukhec.ac.uk/publications/reports/beowulf_paper.pdf: [s.n.], 2009.

AMARAL, D. M. *Análise de desempenho de topologias de redes em chip (NoC).* Dissertação (Mestrado) — Universidade de Brasília, Brasília, DF, Brazil, 2008.

ARM. *ARM 1136JF-S processor.* http://www.arm.com: [s.n.], 2008.

BAKER, J. Reducing bias and inefficiency in the selection algorithm. In: *Proceedings of the Third International Conference on Genetic Algorithms.* Hillsdale, NJ, USA: Lawrence Erlbaum Associates, 1987. p. 14–21.

BARBOSA, A. *Algoritmos genéticos paralelos.* Salvador, CE, Brazil: [s.n.], out 1998.

BATISTA, M. *Algoritmos genéticos em ambientes paralelos.* São José dos Campos, SP, Brazil: [s.n.], 2005.

BLELLOCH. *Vector models for data-parallel computing.* Cambridge, MA, USA: The MIT Press, 1990. 255 p.

BUTENHOF, D. R. *Programming with POSIX threads.* Reading, MA, USA: Addison-Wesley Professional, 1997. 400 p.

CANTU-PAZ, E. A summary of research on parallel genetic algorithms. 1995.

CHAMBERLAIN, S. *Cygwin*. http://www.sourceware.org/cygwin: [s.n.], 2009.

CHAPMAN, B. *Using OpenMP: portable shared memory parallel programming*. Cambridge, MA, USA: The MIT Press, 2007. 353 p.

CHIVERS, I. *Introduction to programming with Fortran: with coverage of Fortran 90, 95, 2003 and 77*. New York, NY, USA: Springer, 2008. 592 p.

CHIWIACOWSKY, L. D. et al. Identifying initial conduction in heat conduction transfer by a genetic algorithm: a parallel aproach. In: . [S.l.: s.n.], 1980. v. 28, n. 4, p. 180–195.

COSTA, R. *Análise de desempenho de um algoritmo Paralelo Implementado em um cluster Beowulf*. Lages, SC, Brazil: [s.n.], nov 2002.

DAVIS, L. *Handbook of genetic algorithms*. New York, NY, USA: Van Nostrand Reinhold Company, 1991. 385 p.

DEJONG, K. *The analysis and behaviour of a class of genetic adaptive systems*. Tese (Doutorado) — University of Michigan, MI, USA, 1975.

ECOSCENTRIC. *eCos*. http://ecos.sourceware.org: [s.n.], 2008.

ESHELMAN L. J.; SHAFFER, D. J. Real-coded genetic algorithms and interval-schemata. In: *Foundation of Genetic Algorithms 3*. [S.l.]: Morgan Kaufmann, 1992. p. 187–202.

GAPH. *Hardware Design Support Group*. http://www.inf.pucrs.br/ gaph: [s.n.], 2006.

GEIST A., B. *PVM : parallel virtual machine*. Cambridge, MA, USA: The MIT Press, 1994. 299 p.

GIRãO, G. Implementação de uma plataforma mp-soc baseada em noc com solução de diretório para manutenção da coerência de cache. In: *PublICa III (2007)*. [S.l.: s.n.], 2007. p. 9–17.

GNU. *Binutils*. http://www.gnu.org/software/binutils: [s.n.], 2009.

GNU. *Gcc*. http://www.gnu.org/software/gcc: [s.n.], 2009.

GNU. *Projeto GNU*. http://www.gnu.org: [s.n.], 2009.

GOLDBERG, D. E. *Genetic algorithms in search, optimization, and machine learning.* Reading, MA, USA: Addison-Wesley Professional, 1989. 432 p.

GOMES, J. L. S. *Paralelização de algoritmo de simulação de Monte Carlo para a adsorção em superfícies heterogêneas bidimensionais.* Dissertação (Mestrado) — Universidade Estadual de Maringá, Maringá, PR, Brazil, 2009.

HAUSEN, A. C. *ValiMPI: uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem.* Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, PR, Brazil, 2005.

HOLLAND, J. H. *Adaptation in natural and artificial systems.* Cambridge, MA, USA: The MIT Press, 1975. 228 p.

HOMAYOUNFAR, H.; AREIBI, S.; WANG, F. An island based ga for static/dynamic optimization problems. In: . [S.l.]: Watam Press, 2003.

HUE, X. *Genetic algorithms for optimization – background and applications.* 1997.

IBM. *PowerPC 440 processor.* http://www.ibm.com: [s.n.], 2008.

JOHNSTON, J. *Newlib.* http://www.sourceware.org/newlib: [s.n.], 2009.

KARAIVAZOGLOU, E.; SPIRAKIS, P. G.; TRIANTAFILOU, V. Wormhole versus deflection routing: a case study on the mesh. In: *COCOON '96: Proceedings of the Second Annual International Conference on Computing and Combinatorics.* London, UK: Springer-Verlag, 1996. p. 31–40.

KERNIGHAN, B. W. R. D. M. *C programming language.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. 274 p.

KOELBEL C., H. *The High performance Fortran handbook.* Cambridge, MA, USA: The MIT Press, 1993. 345 p.

LACERDA, S.; CARVALHO, A. Introdução aos algoritmos genéticos. In: *Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais.* Porto Alegre, RS, Brazil: Editora da Universidade Federal do Rio Grande do Sul (UFRGS), 1999. p. p99–150.

LIN, G. et al. Parallel genetic algorithms on pvm. In: *Proceedings of the International Conference on Parallel Algorithms (ICPA'95).* [S.l.: s.n.], 1995. p. page.

LONGHIN, G. C. *Implementação paralela do método de resolução formal de sistemas de equações*. Dissertação (Mestrado) — Universidade Estadual de Campinas, Campinas, SP, Brazil, set 2001.

MATHWORKS, I. *Function peaks*. http://www.mathworks.com: [s.n.], 2007.

MELLO, A. et al. Multinoc: a multiprocessing system enabled by a network on chip. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005. p. 234–239.

MELLO, A. M. *Arquitetura multiprocessada em SoCs: estudo de diferentes topologias de conexão*. Porto Alegre, RS, Brazil: [s.n.], jun 2003.

MICHALEWICZ, Z. *Genetic algorithms + data structures = evolution programs*. New York, NY, USA: Springer-Verlag, 1994. 387 p.

MICROSOFT. *Microsoft CE*. http://www.microsoft.com: [s.n.], 2008.

MILLBERG, M. et al. The nostrum backbone: a communication protocol stack for networks on chip. In: *In Proc. Int'l Conference on VLSI Design*. [S.l.: s.n.], 2004. p. 693–696.

MIPS. *MIPS32 24Kf processor*. http://www.mips.com: [s.n.], 2008.

MITCHELL, M. *An introduction to genetic algorithms*. Cambridge, MA, USA: The MIT Press, 1988. 206 p.

MORAES, F. et al. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 38, n. 1, p. 69–93, 2004.

NILSSON, E. *Design and implementation of a Hot-potato Switch in a network on chip*. Dissertação (Mestrado) — Department of Microelectronics and Information Technology, Royal Institute of Technology, IMIT/LECS 2002-11, Stockholm, Sweden, jun 2002. Disponível em: <http://www.imit.kth.se/ãxel/papers/2002/nilsson-masters.pdf>.

OPENCORES.ORG. 2006. Disponível em: <http://opencores.org/>.

PACHECO, P. *Parallel programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1996. 418 p.

PACKARD, H. *Parallel programming guide for HP-UX systems*. USA: Hewlett Packard, 1988. 394 p.

PAMUNUWA, D. et al. A study on the implementation of 2-d mesh-based networks-on-chip in the nanometre regime. *Integr. VLSI J.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 38, n. 1, p. 3–17, 2004.

PATTERSON, D.; HENNESSY, J. *Computer organization and design: the hardware/software interface*. 2 sub. ed. [S.l.]: Morgan Kaufmann, 1998.

QNX. *QNX RTOS*. http://www.qnx.com: [s.n.], 2008.

RHOADS, S. *Plasma microprocessor*. http://www.opencores.org: [s.n.], 2006.

RIJPKEMA, E.; GOOSSENS, K.; WIELAGE, P. A router architecture for networks on silicon. In: *In Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*. [S.l.: s.n.], 2001. p. 181–188.

RUIZ, P. M.; ANTONIO. Using genetic algorithms to optimize the behavior of adaptive multimedia applications in wireless and mobile scenarios. In: *IEEE Wireless Communications and Networking Conf. (WCNC'2003)*. [S.l.]: IEEE Press, 2003. p. 2064–2068.

SALES, P. S. B. *Avaliação de desempenho de ferramentas de renderização de imagens em clusters openMosix e arquiteturas multicore*. Recife, Pe, Brazil: [s.n.], jun 2008.

SIGUENZA-TORTOSA, D. *VHDL-based simulation environmente for Proteo NoC*. Tampere, Finland: [s.n.], 2002.

SILBERCHATZ, A. *Applied operating system concepts*. New York, NY, USA: John Wiley and Sons, 2000. 840 p.

SILVA, A. J. M. *Implementação de um algoritmo genético utilizando O modelo de ilhas*. Dissertação (Mestrado) — COPPE, UFRJ, Rio de Janeiro, RJ, Brazil, ago 2005.

SWEETMAN, D. *See MIPS run*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

TANENBAUM, A. *Operating systems: design and implementation*. NJ, USA: Prentice-Hall, 1997. 939 p.

TORVALDS, L. B. *Embedded Linux*. http://www.linuxdevices.com: [s.n.], 2008.

TOTA, S.; CASU, M. R.; MACCHIARULO, L. Implementation analysis of noc: a mpsoc trace-driven approach. In: *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2006. p. 204–209.

UFSC. *EPOS: embedded parallel operating system*. http://epos.lisha.ufsc.br: [s.n.], 2008.

WHITLEY, D. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In: *Proceedings of the Third International Conference on Genetic Algorithms*. [S.l.]: Morgan Kaufmann, 1989. p. 116–121.

WIKLUND, D. *Development and performance evaluation of networks on chip*. Tese (Doutorado) — Linköping University, Linköping, Sweden, 2005.

WOSZEZENKI, C. *Alocação de tarefas e comunicação entre tarefas em MPSoCs*. Dissertação (Mestrado) — Faculdade de Informática, PUCRS, Porto Alegre, RS, Brazil, jun 2007.

ZEFERINO, C. A. *Redes-em-Chip: arquiteturas e modelos para avaliação de área e desempenho*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brasil, 2003.

ZHANG, Q.; LEUNG, Y.-W. An orthogonal genetic algorithm for multimedia multicast routing. *IEEE Trans. Evolutionary Computation*, IEEE Press, v. 3, n. 1, p. 53–62, april 1999.

ZIMMERMANN, H. Osi reference model: the is0 model of architecture for open systems interconnection. *IEEE TRANSACTIONS ON COMMUNICATIONS*, IEEE Press, v. 28, n. 4, p. 425–432, April 1980.

APÊNDICE A – Configuração da Plataforma

As modificações introduzidas na plataforma permitem agora a fácil modificação de vários parâmetros que eram de muito difícil alteração ou não podiam ser alterados na versão original da plataforma. Esses parâmetros ajustáveis são apresentados a seguir.

A.1 Modificação do tamanho do *flit* da chave

O tamanho do *flit* da chave pode ser modificado no arquivo Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant TAM_FLIT : integer range 1 to 32 := 32;  -- pode ser 16 ou 32
```

A.2 Modificação do tamanho da fila da chave

O tamanho da fila da chave pode ser modificado no arquivo Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant TAM_BUFFER : integer := 4;
```

A.3 Modificação do tamanho do *flit* da interface de rede

O tamanho do *flit* da interface de rede pode ser modificado no arquivo Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant TAM_NI_FLIT : integer range 1 to 32 := 32;  -- pode ser 16 ou 32
```

A.4 Modificação do tamanho da fila da interface de rede

O tamanho da fila da interface de rede pode ser modificado no Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant TAM_NI_BUFFER : integer range 1 to 32 := 32;  -- pode ser 16 ou 32
```

A.5 Modificação do tamanho da rede intrachip

O tamanho da rede intrachip pode ser modificado no arquivo Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant MAX_X : integer range 1 to 15 := 2;
constant MAX_Y : integer range 1 to 15 := 2;
--- os enderecos dos switches vao de (0,0) ate (MAX_X,MAX_Y) ---
--- valores para uma noc 3x3 ---
```

A.6 Modificação do tamanho da página

O tamanho da página pode ser modificado nos arquivos Hermes_package.vhd e ids_kernel-slave.h

```
--- Hermes_package.vhd ---
constant TAM_PAGINA : integer range 14 to 27 := 18; -- pode variar de 14 ate 27
--- o tamanho de pagina e dado por 2 ^ TAM_PAGINA ---
--- o numero de paginas e dado por 2 ^ (28 - TAM_PAGINA) ---

--- ids_kernel-slave.h ---
define PAGESIZE 0x40000
define MAXLOCALTASKS 7
```

A.7 Modificação do tamanho da memória

O tamanho da memória pode ser modificado no arquivo Hermes_package.vhd

```
--- Hermes_package.vhd ---
constant TAM_MEMORIA : integer range 16 to 27 := 21; -- pode variar de 14 ate 27
--- o tamanho da memoria ram interna e dado por 2 ^ (TAM_MEMORIA) ---
```

A.8 Modificação do endereço do processador mestre

O endereço do processador mestre pode ser modificado nos arquivos mpsoc.vhd e ids_kernel-slave.h

```
--- mpsoc.vhd ---
plasma_master: if (col = 0) and (row = 0) generate
plasma_slave: if not((col = 0) and (row = 0)) generate

--- ids_kernel-slave.h ---
#define MASTERADDRESS 0x00
```

APÊNDICE B – Instruções de Uso da Plataforma

Para utilizar a plataforma HMPS, é necessária a ferramenta de simulação ModelSim, o sistema operacional Windows com o Cygwin instalado ou o sistema operacional Linux (Fedora 9 de preferência). As Seções seguintes descrevem os arquivos utilizados pela plataforma HMPS, sua instalação, o processo de Desenvolvimento de aplicações e o processo de compilação de tarefas. Os arquivos utilizados pela plataforma HMPS são descritos na Tabela 26.

B.2 Instalação da Plataforma

B.2.1 Windows

1.Instale o ModelSim

2.Instale o Cygwin

3.Descompacte o arquivo

```
D:\cross-windows\gcc-4.3.0.tar.gz em C:\cygwin\usr\local
```

4.Descompacte o arquivo

```
D:\hmps.tar.gz em C:\
```

B.2.2 Linux

1.Instale o ModelSim

2.Instale os pacotes gcc, glibc-devel, gmp, gmp-devel, mpfr, mpfr-devel.

3.Descompacte o arquivo

```
/media/cdrom/cross-linux/gcc-4.3.0.tar.gz em /usr/local
```

Tabela 26: Arquivos da Plataforma HMPS

Diretório	Conteúdo	Descrição
cross-windows	gccmips-tar.gz	Compilador cruzado para o Windows
cross-linux	gccmips-tar.gz	Compilador cruzado para o Linux
simulation	subdiretório plasma	Contém o modelo VHDL do processador Plasma
	subdiretório noc	Contém o modelo VHDL da rede intrachip Hermes
	subdiretório repository	Contém o arquivo fonte do repositório de tarefas
	testbench.vhd	*Testbench* da plataforma
	compile.win	*Script* para a compilação do modelo da plataforma no Windows
	compile.lin	*Script* para a compilação do modelo da plataforma no Linux
	code_master.txt	*Microkernel* que é executado no processador mestre
	code_slave.txt	*Microkernel* que é executado nos processadores escravos
	output_master.txt	Relatório de execução no processador mestre
	output_slave_xx.txt	Relatório de execução nos processadores escravos
software	subdiretório *applications*	Cada aplicação deve possuir um subdiretório aqui com o nome da mesma
	subdiretório *build*	Contém os arquivos de *include* e o *makefile* para a compilação da aplicação
	subdiretório include	Contém os arquivos de *include* utilizados pelas tarefas e pelo *microkernel*
	subdiretório kernel16	Contém o *microkernel* dos processadores mestre e escravo para ser utilizado com *flit* de 16 *bits*
	subdiretório kernel32	Contém o *microkernel* dos processadores mestre e escravo para ser utilizado com *flit* de 32 *bits*
tools	*convert*	Ferramenta utilizada para a geração de código objeto
	rom_loader	Ferramenta que carrega os códigos objetos das tarefas para o repositório de tarefas

4.Descompacte o arquivo

```
/media/cdrom/hmps.tar.gz em /home/<usuario>
```

5.Edite o arquivo

```
/etc/profile
```

6.Inclua no final a linha:

```
Export PATH=\$PATH:/usr/local/gccmips/bin
```

B.3 Desenvolvimento de aplicações

As aplicações são desenvolvidas em linguagem C e são localizadas no diretório applications. Cada aplicação desenvolvida deve possuir um subdiretório com o seu nome. Em muitos casos, as aplicações executam mais de uma instância da mesma tarefa, como no caso do MicroAGP, ou possuem várias tarefas diferentes como no caso da aplicação throughput. No primeiro caso, o subdiretório com o nome da tarefa contém o código fonte da mesma. No último caso, o subdiretório com o nome da tarefa contém vários subdiretórios que contém os códigos das tarefas da aplicação. O subdiretório com o nome da tarefa deve também conter vários arquivos utilizados para a sua compilação e que são descritos na Tabela 27.

Tabela 27: Arquivos utilizados para a compilação da tarefa

Arquivo	Descrição
ids-<nome_aplicacao>.h	Utilizado para mapear os nomes das tarefas em números de identificação
ids_kernel-slave.h	Utilizado para definir o tamanho de página, o numero de páginas por processador, o número máximo de tarefas por processador, o número máximo de tarefas da platatorma e o endereço do processador mestre
ids_kernel-master.h	Utilizado para definir os endereços dos processadores escravos, o número de processadores utilizados, os processadores que serão inicializados e que tarefa da aplicação será executada por qual processador
makefile	É o script de compilação da aplicação

O *microkernel* da plataforma HMPS possui várias primitivas desenvolvidas para serem utilizadas pelas aplicações. Essas primitivas são descritas na Tabela 28.

Tabela 28: Primitivas da plataforma HMPS

Primitiva	Descrição
putchar(caracter)	Exibe um caracter
puts(mensagem)	Exibe uma mensagem no processador local (somente caracteres)
echo(mensagem)	Exibe uma mensagem no processador mestre (somente caracteres)
itoa(numero)	Converte um número inteiro em caracteres decimais (0-9)
itoh(numero)	Converte um número inteiro em caracteres hexadecimais (0-9,A-F)
getprocessorid()	Retorna o identificador do processador que está executando a tarefa corrente
getprocessid()	Retorna o identificador da tarefa que está sendo executada
GetTick()	Retorna o número de ciclos de relógio consumidos desde o início da execução da plataforma
WritePipe(mensagem,tarefa)	Envia uma mensagem para uma tarefa
ReadPipe(mensagem,tarefa)	lê uma mensagem de uma tarefa

B.3.1 Exemplo de aplicação 1 - AGPE

Arquivo applications/ga/task.c: Esse é código fonte da tarefa 1 da aplicação. Serão executadas várias instâncias dessa tarefa.

Arquivo applications/ga/ids_ga.h:

```
#define TASKA 0
#define TASKB 1
#define TASKC 2
#define TASKD 3
#define TASKE 4
#define TASKF 5
#define TASKG 6
#define TASKH 7
```

Arquivo applications/ga/ids_kernel-slave.h:

```
#define PAGESIZE 0x40000
#define MAXLOCALTASKS 3
#define MAXGLOBALTASKS 24
#define INITIALTASKS 8
```

```
#define MASTERADDRESS 0x00
```

Arquivo applications/ga/ids_kernel-master.h (alocação estática):

```
#define SLAVE0 0x00000001
#define SLAVE1 0x00000002
#define SLAVE2 0x00000012
#define SLAVE3 0x00000022
#define SLAVE4 0x00000021
#define SLAVE5 0x00000020
#define SLAVE6 0x00000010
#define SLAVE7 0x00000011
#define MAXLOCALTASKS 3
#define MAXGLOBALTASKS 24
#define MAXPROCESSORS 8
void InitializeProcessors(){
InsertProc(SLAVE0);
InsertProc(SLAVE1);
InsertProc(SLAVE2);
InsertProc(SLAVE3);
InsertProc(SLAVE4);
InsertProc(SLAVE5);
InsertProc(SLAVE6);
InsertProc(SLAVE7); }
void InitializeTasks() {
InsertTaskLoc(0,SLAVE0);
OccupedPage(SLAVE0);
InsertTaskLoc(1,SLAVE1);
OccupedPage(SLAVE1);
InsertTaskLoc(2,SLAVE2);
OccupedPage(SLAVE2);
InsertTaskLoc(3,SLAVE3);
OccupedPage(SLAVE3);
InsertTaskLoc(4,SLAVE4);
OccupedPage(SLAVE4);
```

```
InsertTaskLoc(5,SLAVE5);
OccupedPage(SLAVE5);
InsertTaskLoc(6,SLAVE6);
OccupedPage(SLAVE6);
InsertTaskLoc(7,SLAVE7);
OccupedPage(SLAVE7); }
```

Arquivo applications/ga/ids_kernel-master.h (alocação dinâmica):

```
#define SLAVE0 0x00000001
#define SLAVE1 0x00000002
#define SLAVE2 0x00000012
#define SLAVE3 0x00000022
#define SLAVE4 0x00000021
#define SLAVE5 0x00000020
#define SLAVE6 0x00000010
#define SLAVE7 0x00000011
#define MAXLOCALTASKS 3
#define MAXGLOBALTASKS 24
#define MAXPROCESSORS 8
void InitializeProcessors(){
InsertProc(SLAVE0);
InsertProc(SLAVE1);
InsertProc(SLAVE2);
InsertProc(SLAVE3);
InsertProc(SLAVE4);
InsertProc(SLAVE5);
InsertProc(SLAVE6);
InsertProc(SLAVE7); }
void InitializeTasks() {
InsertTaskLoc(0,SLAVE0);
OccupedPage(SLAVE0); }
```

Arquivo applications/ga/makefile:

```
CFLAGS = -mips1 -mno-fused-madd -msoft-float -O2 -Wall -c -s
```

```
LDMIPS = -Bstatic \
-L/usr/local/gccmips/mips-elf/lib/soft-float \
-lc -lcfe -lg -lidt -llsi -lm -lnosys \
-lnullmon -lpmon \
-L /usr/local/gccmips/lib/gcc/mips-elf/4.3.0/soft-float/ \
-lgcc -lgcov \
CC_X86 = gcc
GCC_MIPS = mips-elf-gcc.exe  $(CFLAGS)
AS_MIPS = mips-elf-as.exe
LD_MIPS = mips-elf-ld.exe
DUMP_MIPS = mips-elf-objdump.exe
16all: convert_bin kernel16_slave kernel16_master task0
       task1 task2 task3 task4 task5 task6 task7 loader
32all: convert_bin kernel32_slave kernel32_master task0
       task1 task2 task3 task4 task5 task6 task7 loader
convert_bin: $(CC_X86) -o ../../tools/convert_bin ../../tools/convert.c
task0:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task0.o
          --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task0.map -s -N -o test.exe
          bootTask.o common.o task0.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task0.o > task0.asm
@$(DUMP_MIPS) --disassemble test.exe > task0.lst
../../tools/convert_bin.exe
mv code.txt code0.txt
rm *.o *.bin test.exe
task1:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task1.o
       --include ids_ag_b.h
```

```
$(LD_MIPS) -Ttext 0 -eentry -Map task1.map -s -N -o test.exe
       bootTask.o common.o task1.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task1.o > task1.asm
@$(DUMP_MIPS) --disassemble test.exe > task1.lst
../../tools/convert_bin.exe
mv code.txt code1.txt
rm *.o *.bin test.exe
task2:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task2.o
        --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task2.map -s -N -o test.exe
        bootTask.o common.o task2.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task2.o > task2.asm
@$(DUMP_MIPS) --disassemble test.exe > task2.lst
../../tools/convert_bin.exe
mv code.txt code2.txt
rm *.o *.bin test.exe
task3:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task3.o
          --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task3.map -s -N -o test.exe
          bootTask.o common.o task3.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task3.o > task3.asm
@$(DUMP_MIPS) --disassemble test.exe > task3.lst
../../tools/convert_bin.exe
mv code.txt code3.txt
rm *.o *.bin test.exe
task4:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
```

```
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task4.o
        --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task4.map -s -N -o test.exe
        bootTask.o common.o task4.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task4.o > task4.asm
@$(DUMP_MIPS) --disassemble test.exe > task4.lst
../../tools/convert_bin.exe
mv code.txt code4.txt
rm *.o *.bin test.exe
task5:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task5.o
        --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task5.map -s -N -o test.exe
        bootTask.o common.o task5.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task5.o > task5.asm
@$(DUMP_MIPS) --disassemble test.exe > task5.lst
../../tools/convert_bin.exe
mv code.txt code5.txt
rm *.o *.bin test.exe
task6:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task6.o
        --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task6.map -s -N -o test.exe
        bootTask.o common.o task6.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task6.o > task6.asm
@$(DUMP_MIPS) --disassemble test.exe > task6.lst
../../tools/convert_bin.exe
mv code.txt code6.txt
```

```
rm *.o *.bin test.exe
task7:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/ag/task.c" -o task7.o
        --include ids_ag_b.h
$(LD_MIPS) -Ttext 0 -eentry -Map task7.map -s -N -o test.exe
        bootTask.o common.o task7.o $(LDMIPS)
@$(DUMP_MIPS) --disassemble task7.o > task7.asm
@$(DUMP_MIPS) --disassemble test.exe > task7.lst
../../tools/convert_bin.exe
mv code.txt code7.txt
rm *.o *.bin test.exe
kernel16_master:
$(AS_MIPS) -o bootKernel.o ../kernel16/master/bootKernel.asm
$(GCC_MIPS) ../kernel16/master/kernel.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_master.map -s -N -o test.exe
         bootKernel.o kernel.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_master.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_master.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_master.txt
mv code.txt code_master.txt
rm *.o *.bin test.exe
kernel16_slave:
$(AS_MIPS) -o bootKernel.o ../kernel16/slave/bootKernel.asm
$(GCC_MIPS) ../kernel16/slave/kernel.c
$(GCC_MIPS) ../include/common.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_slave.map -s -N -o test.exe
        bootKernel.o kernel.o common.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_slave.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_slave.lst
../../tools/convert_bin.exe
```

```
cp code.txt ../../simulation/code_slave.txt
mv code.txt code_slave.txt
rm *.o *.bin test.exe
kernel32_master:
$(AS_MIPS) -o bootKernel.o ../kernel32/master/bootKernel.asm
$(GCC_MIPS) ../kernel32/master/kernel.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_master.map -s -N -o test.exe
         bootKernel.o kernel.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_master.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_master.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_master.txt
mv code.txt code_master.txt
rm *.o *.bin test.exe
kernel32_slave:
$(AS_MIPS) -o bootKernel.o ../kernel32/slave/bootKernel.asm
$(GCC_MIPS) ../kernel32/slave/kernel.c
$(GCC_MIPS) ../include/common.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_slave.map -s -N -o test.exe
        bootKernel.o kernel.o common.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_slave.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_slave.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_slave.txt
mv code.txt code_slave.txt
rm *.o *.bin test.exe
loader:
../../tools/rom_loader.exe code0.txt code1.txt code2.txt code3.txt
         code4.txt code5.txt code6.txt code7.txt
cp extern_memory.vhd ../../simulation/repository/
```

B.3.2 Exemplo de aplicação 2 - Troughput

Arquivo applications/troughput/taskA/task.c: Esse é código fonte da tarefa 1 da aplicação. Será executada apenas uma instância dessa tarefa.

Arquivo applications/troughput/taskB/task.c: esse é código fonte da tarefa 2 da aplicação. Será executada apenas uma instância dessa tarefa.

Arquivo applications/troughput/ids_troughput.h:

```
#define TASKA 0
#define TASKB 1
```

Arquivo applications/troughput/ids_kernel-slave.h:

```
#define PAGESIZE 0x40000
#define MAXLOCALTASKS 3
#define MAXGLOBALTASKS 24
#define INITIALTASKS 2
#define MASTERADDRESS 0x00
```

Arquivo applications/troughput/ids_kernel-master.h (alocação estática):

```
#define SLAVE0 0x00000001
#define SLAVE1 0x00000020
#define MAXLOCALTASKS 3
#define MAXGLOBALTASKS 24
#define MAXPROCESSORS 2
void InitializeProcessors(){
InsertProc(SLAVE0);
InsertProc(SLAVE1); }
void InitializeTasks() {
InsertTaskLoc(0,SLAVE0);
OccupedPage(SLAVE0);
InsertTaskLoc(1,SLAVE1);
OccupedPage(SLAVE1); }
```

Arquivo applications/troughput/makefile:

```
CFLAGS = -O2 -Wall -c -s
CC_X86 = gcc
GCC_MIPS = mips-elf-gcc.exe  $(CFLAGS)
AS_MIPS = mips-elf-as.exe
LD_MIPS = mips-elf-ld.exe
DUMP_MIPS = mips-elf-objdump.exe
16all: convert_bin kernel16_slave kernel16_master task0 task1 loader
32all: convert_bin kernel32_slave kernel32_master task0 task1 loader
convert_bin:
$(CC_X86) -o ../../tools/convert_bin ../../tools/convert.c
task0:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/throughput/taskA/task.c"
        -o task0.o --include ids_throughput.h
$(LD_MIPS) -Ttext 0 -eentry -Map task0.map -s -N -o test.exe
        bootTask.o common.o task0.o
@$(DUMP_MIPS) --disassemble test.exe > task0.lst
../../tools/convert_bin.exe
mv code.txt code0.txt
rm *.o *.bin test.exe
task1:
$(AS_MIPS) -o bootTask.o ../include/bootTask.asm
$(GCC_MIPS) ../include/common.c
$(GCC_MIPS) "C:/hmps/hmps/software/applications/throughput/taskB/task.c"
         -o task1.o --include ids_throughput.h
$(LD_MIPS) -Ttext 0 -eentry -Map task1.map -s -N -o test.exe
         bootTask.o common.o task1.o
@$(DUMP_MIPS) --disassemble test.exe > task1.lst
../../tools/convert_bin.exe
mv code.txt code1.txt
rm *.o *.bin test.exe
kernel16_master:
```

```
$(AS_MIPS) -o bootKernel.o ../kernel16/master/bootKernel.asm
$(GCC_MIPS) ../kernel16/master/kernel.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_master.map -s -N
         -o test.exe bootKernel.o kernel.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_master.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_master.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_master.txt
mv code.txt code_master.txt
rm *.o *.bin test.exe
kernel16_slave:
$(AS_MIPS) -o bootKernel.o ../kernel16/slave/bootKernel.asm
$(GCC_MIPS) ../kernel16/slave/kernel.c
$(GCC_MIPS) ../include/common.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_slave.map -s -N
         -o test.exe bootKernel.o kernel.o common.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_slave.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_slave.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_slave.txt
mv code.txt code_slave.txt
rm *.o *.bin test.exe
kernel32_master:
$(AS_MIPS) -o bootKernel.o ../kernel32/master/bootKernel.asm
$(GCC_MIPS) ../kernel32/master/kernel.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_master.map -s -N
         -o test.exe bootKernel.o kernel.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_master.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_master.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_master.txt
mv code.txt code_master.txt
rm *.o *.bin test.exe
```

```
kernel32_slave:
$(AS_MIPS) -o bootKernel.o ../kernel32/slave/bootKernel.asm
$(GCC_MIPS) ../kernel32/slave/kernel.c
$(GCC_MIPS) ../include/common.c
$(LD_MIPS) -Ttext 0 -eentry -Map kernel_slave.map -s -N
         -o test.exe bootKernel.o kernel.o common.o
@$(DUMP_MIPS) --disassemble bootKernel.o > bootkernel_slave.lst
@$(DUMP_MIPS) --disassemble test.exe > kernel_slave.lst
../../tools/convert_bin.exe
cp code.txt ../../simulation/code_slave.txt
mv code.txt code_slave.txt
rm *.o *.bin test.exe
loader:
../../tools/rom_loader.exe code0.txt code1.txt
cp extern_memory.vhd ../../simulation/repository/
```

B.4 Compilando a aplicação

B.4.1 Windows

1. Copie os arquivos ids_<nome_aplicacao>.h, ids_kernel-master.h, ids_kernel-slave e makefile do diretório

   ```
   C:\hmps\hmps\software\applications\nome_aplicacao
   ```

   para o diretório

   ```
   C:\hmps\hmps\software\build
   ```

2. Execute o Cygwin e dentro do ambiente do cygwin execute os comandos

   ```
   Source /cygdrive/c/hmps/hmps/bashrc
   cd /cygdrive/c/hmps/hmps/software/build
   make -B 16all (se o tamanho do flit utilizado for 16 bits)
   make -B 32all (se o tamanho do flit utilizado for 32 bits)
   ```

B.4.2 Linux

1. Copie os arquivos ids_<nome_aplicacao>.h, ids-kernel-master.h, ids-kernel-slave e makefile do diretório

   ```
   /home/<usuario>/hmps/hmps/software/applications/nome_aplicacao
   ```

   para o diretório

   ```
   /home/<usuario>/hmps/hmps/software/build
   ```

2. Execute os commandos:

   ```
   cd /home/usuario/hmps/hmps/software/build
   make -B 16all (se o tamanho do flit utilizado for 16 bits)
   make -B 32all (se o tamanho do flit utilizado for 32 bits)
   ```

B.5 Simulando a aplicação

B.5.1 Windows

1. Execute o ModelSim e crie o projeto hmps

2. Execute os comandos

   ```
   do compile.win; run xx ms
   ```

3. Os resultados da simulação são gravados nos arquivos output_master.txt, output_slave_1.txt, output_slave_2.txt, ... e output_slave_n.txt do diretório:

   ```
   C:\hmps\hmps\simulation
   ```

B.5.2 Linux

1. Execute o ModelSim e crie o projeto hmps

2. Execute os comandos

   ```
   do compile.lin; run xx ms
   ```

3. Os resultados da simulação são gravados nos arquivos output_master.txt, output_slave_1.txt, output_slave_2.txt, ... e output_slave_n.txt do diretório:

   ```
   /home/<usuario>/hmps/hmps/simulation
   ```

APÊNDICE C – Modelo VHDL da Chave

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.HermesPackage.all;
entity switch is
generic(col, row: integer);
port(
    clock:     in  std_logic;
    reset:     in  std_logic;
    clock_rx:  in  regNport;
    rx:        in  regNport;
    credit_i:  in  regNport;
    data_in:   in  arrayNport_regflit;
    clock_tx:  out regNport;
    tx:        out regNport;
    credit_o:  out regNport;
    data_out:  out arrayNport_regflit);
end switch;
architecture switch of switch is
signal h, ack_h, data_av, sender, data_ack: regNport := (others=>'0');
signal data: arrayNport_regflit := (others=>(others=>'0'));
signal mux_in, mux_out: arrayNport_reg3 := (others=>(others=>'0'));
signal free: regNport := (others=>'0');
begin
```

```
switch_type_1: if col = 0 and row = 0 generate
 FEast : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(0),
     rx => rx(0),
     h => h(0),
     ack_h => ack_h(0),
     data_av => data_av(0),
     data => data(0),
     sender=>sender(0),
     clock_rx => clock_rx(0),
     data_ack => data_ack(0),
     credit_o => credit_o(0));
 FNorth : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(2),
     rx => rx(2),
     h => h(2),
     ack_h => ack_h(2),
     data_av => data_av(2),
     data => data(2),
     sender=>sender(2),
     clock_rx => clock_rx(2),
     data_ack => data_ack(2),
     credit_o => credit_o(2));
 FLocal : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
```

```
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
        data_ack => data_ack(4),
        credit_o => credit_o(4));
    --aterrando os sinais de entrada do buffer 1 removido
    h(1)<='0';
    data_av(1)<='0';
    data(1)<=(others=>'0');
    sender(1)<='0';
    credit_o(1)<='0';
    --aterrando os sinais de entrada do buffer 3 removido
    h(3)<='0';
    data_av(3)<='0';
    data(3)<=(others=>'0');
    sender(3)<='0';
    credit_o(3)<='0';
  end generate switch_type_1;
  switch_type_2: if col = 0 and row > 0 and row < MAX_Y generate
    FEast : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(0),
        rx => rx(0),
        h => h(0),
        ack_h => ack_h(0),
        data_av => data_av(0),
```

```
        data => data(0),
        sender=>sender(0),
        clock_rx => clock_rx(0),
        data_ack => data_ack(0),
        credit_o => credit_o(0));
    FNorth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(2),
        rx => rx(2),
        h => h(2),
        ack_h => ack_h(2),
        data_av => data_av(2),
        data => data(2),
        sender=>sender(2),
        clock_rx => clock_rx(2),
        data_ack => data_ack(2),
        credit_o => credit_o(2));
    FSouth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(3),
        rx => rx(3),
        h => h(3),
        ack_h => ack_h(3),
        data_av => data_av(3),
        data => data(3),
        sender=>sender(3),
        clock_rx => clock_rx(3),
        data_ack => data_ack(3),
        credit_o => credit_o(3));
```

```
 FLocal : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(4),
     rx => rx(4),
     h => h(4),
     ack_h => ack_h(4),
     data_av => data_av(4),
     data => data(4),
     sender=>sender(4),
     clock_rx => clock_rx(4),
     data_ack => data_ack(4),
     credit_o => credit_o(4));
 --aterrando os sinais de entrada do buffer 1 removido
 h(1)<='0';
 data_av(1)<='0';
 data(1)<=(others=>'0');
 sender(1)<='0';
 credit_o(1)<='0';
end generate switch_type_2;
switch_type_3: if col = 0 and row = MAX_Y generate
 FEast : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(0),
     rx => rx(0),
     h => h(0),
     ack_h => ack_h(0),
     data_av => data_av(0),
     data => data(0),
     sender=>sender(0),
```

```
        clock_rx => clock_rx(0),
        data_ack => data_ack(0),
        credit_o => credit_o(0));
    FSouth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(3),
        rx => rx(3),
        h => h(3),
        ack_h => ack_h(3),
        data_av => data_av(3),
        data => data(3),
        sender=>sender(3),
        clock_rx => clock_rx(3),
        data_ack => data_ack(3),
        credit_o => credit_o(3));
    FLocal : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
        data_ack => data_ack(4),
        credit_o => credit_o(4));
    --aterrando os sinais de entrada do buffer 1 removido
    h(1)<='0';
```

```
 data_av(1)<='0';
 data(1)<=(others=>'0');
 sender(1)<='0';
 credit_o(1)<='0';
 --aterrando os sinais de entrada do buffer 2 removido
 h(2)<='0';
 data_av(2)<='0';
 data(2)<=(others=>'0');
 sender(2)<='0';
 credit_o(2)<='0';
end generate switch_type_3;
switch_type_4: if col > 0 and col < MAX_X and row = 0 generate
 FEast : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(0),
     rx => rx(0),
     h => h(0),
     ack_h => ack_h(0),
     data_av => data_av(0),
     data => data(0),
     sender=>sender(0),
     clock_rx => clock_rx(0),
     data_ack => data_ack(0),
     credit_o => credit_o(0));
 FWest : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(1),
     rx => rx(1),
     h => h(1),
```

```
        ack_h => ack_h(1),
        data_av => data_av(1),
        data => data(1),
        sender=>sender(1),
        clock_rx => clock_rx(1),
        data_ack => data_ack(1),
        credit_o => credit_o(1));
    FNorth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(2),
        rx => rx(2),
        h => h(2),
        ack_h => ack_h(2),
        data_av => data_av(2),
        data => data(2),
        sender=>sender(2),
        clock_rx => clock_rx(2),
        data_ack => data_ack(2),
        credit_o => credit_o(2));
    FLocal : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
```

```
        data_ack => data_ack(4),
        credit_o => credit_o(4));
    --aterrando os sinais de entrada do buffer 3 removido
    h(3)<='0';
    data_av(3)<='0';
    data(3)<=(others=>'0');
    sender(3)<='0';
    credit_o(3)<='0';
   end generate switch_type_4;
   switch_type_5: if col > 0 and col < MAX_X and row > 0 and row < MAX_Y generate
    FEast : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(0),
        rx => rx(0),
        h => h(0),
        ack_h => ack_h(0),
        data_av => data_av(0),
        data => data(0),
        sender=>sender(0),
        clock_rx => clock_rx(0),
        data_ack => data_ack(0),
        credit_o => credit_o(0));
    FWest : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(1),
        rx => rx(1),
        h => h(1),
        ack_h => ack_h(1),
        data_av => data_av(1),
```

```
        data => data(1),
        sender=>sender(1),
        clock_rx => clock_rx(1),
        data_ack => data_ack(1),
        credit_o => credit_o(1));
    FNorth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(2),
        rx => rx(2),
        h => h(2),
        ack_h => ack_h(2),
        data_av => data_av(2),
        data => data(2),
        sender=>sender(2),
        clock_rx => clock_rx(2),
        data_ack => data_ack(2),
        credit_o => credit_o(2));
    FSouth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(3),
        rx => rx(3),
        h => h(3),
        ack_h => ack_h(3),
        data_av => data_av(3),
        data => data(3),
        sender=>sender(3),
        clock_rx => clock_rx(3),
        data_ack => data_ack(3),
        credit_o => credit_o(3));
```

```
 FLocal : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(4),
     rx => rx(4),
     h => h(4),
     ack_h => ack_h(4),
     data_av => data_av(4),
     data => data(4),
     sender=>sender(4),
     clock_rx => clock_rx(4),
     data_ack => data_ack(4),
     credit_o => credit_o(4));
end generate switch_type_5;
switch_type_6: if col > 0 and col < MAX_X and row = MAX_Y generate
 FEast : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(0),
     rx => rx(0),
     h => h(0),
     ack_h => ack_h(0),
     data_av => data_av(0),
     data => data(0),
     sender=>sender(0),
     clock_rx => clock_rx(0),
     data_ack => data_ack(0),
     credit_o => credit_o(0));
 FWest : Entity work.Hermes_buffer
 port map(
     clock => clock,
```

```
        reset => reset,
        data_in => data_in(1),
        rx => rx(1),
        h => h(1),
        ack_h => ack_h(1),
        data_av => data_av(1),
        data => data(1),
        sender=>sender(1),
        clock_rx => clock_rx(1),
        data_ack => data_ack(1),
        credit_o => credit_o(1));
    FSouth : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(3),
        rx => rx(3),
        h => h(3),
        ack_h => ack_h(3),
        data_av => data_av(3),
        data => data(3),
        sender=>sender(3),
        clock_rx => clock_rx(3),
        data_ack => data_ack(3),
        credit_o => credit_o(3));
    FLocal : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
```

```
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
        data_ack => data_ack(4),
        credit_o => credit_o(4));
 --aterrando os sinais de entrada do buffer 2 removido
 h(2)<='0';
 data_av(2)<='0';
 data(2)<=(others=>'0');
 sender(2)<='0';
 credit_o(2)<='0';
end generate switch_type_6;
switch_type_7: if col = MAX_X and row = 0 generate
 FWest : Entity work.Hermes_buffer
 port map(
        clock => clock,
        reset => reset,
        data_in => data_in(1),
        rx => rx(1),
        h => h(1),
        ack_h => ack_h(1),
        data_av => data_av(1),
        data => data(1),
        sender=>sender(1),
        clock_rx => clock_rx(1),
        data_ack => data_ack(1),
        credit_o => credit_o(1));
 FNorth : Entity work.Hermes_buffer
 port map(
        clock => clock,
        reset => reset,
        data_in => data_in(2),
```

```
        rx => rx(2),
        h => h(2),
        ack_h => ack_h(2),
        data_av => data_av(2),
        data => data(2),
        sender=>sender(2),
        clock_rx => clock_rx(2),
        data_ack => data_ack(2),
        credit_o => credit_o(2));
    FLocal : Entity work.Hermes_buffer
    port map(
        clock => clock,
        reset => reset,
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
        data_ack => data_ack(4),
        credit_o => credit_o(4));
    --aterrando os sinais de entrada do buffer 0 removido
    h(0)<='0';
    data_av(0)<='0';
    data(0)<=(others=>'0');
    sender(0)<='0';
    credit_o(0)<='0';
    --aterrando os sinais de entrada do buffer 3 removido
    h(3)<='0';
    data_av(3)<='0';
    data(3)<=(others=>'0');
```

```
 sender(3)<='0';
 credit_o(3)<='0';
end generate switch_type_7;
switch_type_8: if col = MAX_X and row > 0 and row < MAX_Y generate
 FWest : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(1),
     rx => rx(1),
     h => h(1),
     ack_h => ack_h(1),
     data_av => data_av(1),
     data => data(1),
     sender=>sender(1),
     clock_rx => clock_rx(1),
     data_ack => data_ack(1),
     credit_o => credit_o(1));
 FSouth : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(3),
     rx => rx(3),
     h => h(3),
     ack_h => ack_h(3),
     data_av => data_av(3),
     data => data(3),
     sender=>sender(3),
     clock_rx => clock_rx(3),
     data_ack => data_ack(3),
     credit_o => credit_o(3));
 FNorth : Entity work.Hermes_buffer
```

```
port map(
    clock => clock,
    reset => reset,
    data_in => data_in(2),
    rx => rx(2),
    h => h(2),
    ack_h => ack_h(2),
    data_av => data_av(2),
    data => data(2),
    sender=>sender(2),
    clock_rx => clock_rx(2),
    data_ack => data_ack(2),
    credit_o => credit_o(2));
FLocal : Entity work.Hermes_buffer
port map(
    clock => clock,
    reset => reset,
    data_in => data_in(4),
    rx => rx(4),
    h => h(4),
    ack_h => ack_h(4),
    data_av => data_av(4),
    data => data(4),
    sender=>sender(4),
    clock_rx => clock_rx(4),
    data_ack => data_ack(4),
    credit_o => credit_o(4));
--aterrando os sinais de entrada do buffer 0 removido
h(0)<='0';
data_av(0)<='0';
data(0)<=(others=>'0');
sender(0)<='0';
credit_o(0)<='0';
```

```
end generate switch_type_8;
switch_type_9: if col = MAX_X and row = MAX_Y generate
 FWest : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(1),
     rx => rx(1),
     h => h(1),
     ack_h => ack_h(1),
     data_av => data_av(1),
     data => data(1),
     sender=>sender(1),
     clock_rx => clock_rx(1),
     data_ack => data_ack(1),
     credit_o => credit_o(1));
 FSouth : Entity work.Hermes_buffer
 port map(
     clock => clock,
     reset => reset,
     data_in => data_in(3),
     rx => rx(3),
     h => h(3),
     ack_h => ack_h(3),
     data_av => data_av(3),
     data => data(3),
     sender=>sender(3),
     clock_rx => clock_rx(3),
     data_ack => data_ack(3),
     credit_o => credit_o(3));
 FLocal : Entity work.Hermes_buffer
 port map(
     clock => clock,
```

```
        reset => reset,
        data_in => data_in(4),
        rx => rx(4),
        h => h(4),
        ack_h => ack_h(4),
        data_av => data_av(4),
        data => data(4),
        sender=>sender(4),
        clock_rx => clock_rx(4),
        data_ack => data_ack(4),
        credit_o => credit_o(4));
    --aterrando os sinais de entrada do buffer 0 removido
    h(0)<='0';
    data_av(0)<='0';
    data(0)<=(others=>'0');
    sender(0)<='0';
    credit_o(0)<='0';
    --aterrando os sinais de entrada do buffer 2 removido
    h(2)<='0';
    data_av(2)<='0';
    data(2)<=(others=>'0');
    sender(2)<='0';
    credit_o(2)<='0';
  end generate switch_type_9;
    SwitchControl : Entity work.SwitchControl
        generic map(col => col,    row => row)
    port map(
    clock => clock,
    reset => reset,
    h => h,
    ack_h => ack_h,
    data => data,
    sender => sender,
```

```
    free => free,
    mux_in => mux_in,
    mux_out => mux_out);
    CrossBar : Entity work.Hermes_crossbar
    port map(
    data_av => data_av,
    data_in => data,
    data_ack => data_ack,
    sender => sender,
    free => free,
    tab_in => mux_in,
    tab_out => mux_out,
    tx => tx,
    data_out => data_out,
    credit_i => credit_i);
    CLK_TX : for i in 0 to(NPORT-1) generate
    clock_tx(i) <= clock;
    end generate CLK_TX;
end switch;
```

APÊNDICE D – Modelo VHDL da Rede Intrachip

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.HermesPackage.all;
entity noc is
port(
    clock         : in  regXYlocal;
    reset         : in  std_logic;
    clock_rxLocal : in  regXYlocal;
    rxLocal       : in  regXYlocal;
    data_inLocal  : in  arrayXYlocal_regflit;
    credit_oLocal : out regXYlocal;
    clock_txLocal : out regXYlocal;
    txLocal       : out regXYlocal;
    data_outLocal : out arrayXYlocal_regflit;
    credit_iLocal : in  regXYlocal
    );
end noc;
architecture noc of noc is
signal noc_clock_rx, noc_rx, noc_credit_i: regXYrot;
signal noc_clock_tx, noc_tx, noc_credit_o: regXYrot;
signal noc_data_out, noc_data_in: arrayXYrot_regflit;
begin
  row0 : for row in 0 to MAX_Y generate
  begin
```

```
    col0 : for col in 0 to MAX_X generate
    begin
       inst_switch: entity work.switch
       generic map(col,row)
       port map(
                clock => clock(col)(row),
                reset => reset,
                clock_rx => noc_clock_rx(col+1)(row+1),
                rx => noc_rx(col+1)(row+1),
                credit_i => noc_credit_i(col+1)(row+1),
                data_in => noc_data_in(col+1)(row+1),
                clock_tx => noc_clock_tx(col+1)(row+1),
                tx => noc_tx(col+1)(row+1),
                credit_o => noc_credit_o(col+1)(row+1),
                data_out => noc_data_out(col+1)(row+1)
               );
    end generate col0;
 end generate row0;
 row1 : for row in 1 to MAX_Y+1 generate
 begin
    col1 : for col in 1 to MAX_X+1 generate
    begin
      noc_clock_rx(col)(row)(0)<=noc_clock_tx(col+1)(row)(1);
      noc_rx(col)(row)(0)<=noc_tx(col+1)(row)(1);
      noc_data_in(col)(row)(0)<=noc_data_out(col+1)(row)(1);
      noc_credit_i(col)(row)(0)<=noc_credit_o(col+1)(row)(1);
      noc_clock_rx(col)(row)(1)<=noc_clock_tx(col-1)(row)(0);
      noc_rx(col)(row)(1)<=noc_tx(col-1)(row)(0);
      noc_data_in(col)(row)(1)<=noc_data_out(col-1)(row)(0);
      noc_credit_i(col)(row)(1)<=noc_credit_o(col-1)(row)(0);
      noc_clock_rx(col)(row)(2)<=noc_clock_tx(col)(row+1)(3);
      noc_rx(col)(row)(2)<=noc_tx(col)(row+1)(3);
      noc_data_in(col)(row)(2)<=noc_data_out(col)(row+1)(3);
```

```
      noc_credit_i(col)(row)(2)<=noc_credit_o(col)(row+1)(3);
      noc_clock_rx(col)(row)(3)<=noc_clock_tx(col)(row-1)(2);
      noc_rx(col)(row)(3)<=noc_tx(col)(row-1)(2);
      noc_data_in(col)(row)(3)<=noc_data_out(col)(row-1)(2);
      noc_credit_i(col)(row)(3)<=noc_credit_o(col)(row-1)(2);
      noc_clock_rx(col)(row)(4)<=clock_rxLocal(col-1)(row-1);
      noc_rx(col)(row)(4)<=rxLocal(col-1)(row-1);
      noc_data_in(col)(row)(4)<=data_inLocal(col-1)(row-1);
      noc_credit_i(col)(row)(4)<=credit_iLocal(col-1)(row-1);
      clock_txLocal(col-1)(row-1)<=noc_clock_tx(col)(row)(4);
      txLocal(col-1)(row-1)<=noc_tx(col)(row)(4);
      data_outLocal(col-1)(row-1)<=noc_data_out(col)(row)(4);
      credit_oLocal(col-1)(row-1)<=noc_credit_o(col)(row)(4);
   end generate col1;
end generate row1;
col2 : for col in 1 to MAX_X+1 generate
begin
   noc_clock_rx(col)(0)(2)<='0';
   noc_rx(col)(0)(2)<='0';
   noc_data_in(col)(0)(2)<=(others=>'0');
   noc_credit_i(col)(0)(2)<='0';
   noc_clock_rx(col)(MAX_Y+2)(3)<='0';
   noc_rx(col)(MAX_Y+2)(3)<='0';
   noc_data_in(col)(MAX_Y+2)(3)<=(others=>'0');
   noc_credit_i(col)(MAX_Y+2)(3)<='0';
end generate col2;
row2 : for row in 1 to MAX_Y+1 generate
begin
   noc_clock_rx(0)(row)(0)<='0';
   noc_rx(0)(row)(0)<='0';
   noc_data_in(0)(row)(0)<=(others=>'0');
   noc_credit_i(0)(row)(0)<='0';
   noc_clock_rx(MAX_X+2)(row)(1)<='0';
```

```
    noc_rx(MAX_X+2)(row)(1)<='0';
    noc_data_in(MAX_X+2)(row)(1)<=(others=>'0');
    noc_credit_i(MAX_X+2)(row)(1)<='0';
  end generate row2;
end NOC;
```

APÊNDICE E – Modelo VHDL do Sistema HMPS

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.HermesPackage.all;
entity mpsoc is
   port (
      -- Clock & Reset
      clock_noc : in std_logic;
      reset : in std_logic;
      -- Tasks repository interface
      manager_address : out std_logic_vector(31 downto 2);
      manager_data_read : in std_logic_vector(31 downto 0)
      );
end;
architecture mpsoc of mpsoc is
signal clock : regXYlocal;
signal clock_rx, rx, credit_o: regXYlocal;
signal clock_tx, tx, credit_i: regXYlocal;
signal data_in, data_out : arrayXYlocal_regflit;
begin
   row0 : for row in 0 to MAX_Y generate
   begin
      col0 : for col in 0 to MAX_X generate
      begin
```

```
        clock(col)(row) <= clock_noc;
      end generate col0;
   end generate row0;
   noc: entity work.noc(noc)
   port map(
      clock         => clock,
      reset         => reset,
      clock_rxLocal => clock_rx,
      rxLocal       => rx,
      data_inLocal  => data_in,
      credit_oLocal => credit_o,
      clock_txLocal => clock_tx,
      txLocal       => tx,
      data_outLocal => data_out,
      credit_iLocal => credit_i
      );
 row1 : for row in 0 to MAX_Y generate
 begin
    col1 : for col in 0 to MAX_X generate
    begin
       plasma_master: if (col = 0) and (row = 0) generate
          inst_plasma: entity work.plasma(plasma)
          generic map (
             col                 => col,
             row                 => row,
             memory_type => "TRI_PORT_X",
             processor_type  => "master",
             log_file    => "output_master.txt")
          port map(
             clock               => clock(col)(row),
             reset               => reset,
             clock_tx            => clock_rx(col)(row),
             tx                  => rx(col)(row),
```

```
        data_out            => data_in(col)(row),
        credit_i            => credit_o(col)(row),
        clock_rx            => clock_tx(col)(row),
        rx                  => tx(col)(row),
        data_in             => data_out(col)(row),
        credit_o            => credit_i(col)(row),

        address             => manager_address,
        data_write          => open,
        data_read           => manager_data_read,
        write_byte_enable   => open,
        mem_pause_in        => '0'
        );
end generate plasma_master;
plasma_slave: if not((col = 0) and (row = 0)) generate
    inst_plasma: entity work.plasma(plasma)
     generic map (
        col                 => col,
        row                 => row,
        memory_type => "TRI_PORT_X",
        processor_type  => "slave",
        log_file => "output_slave_" & CONV_HEX(col) &
                    CONV_HEX(row) & ".txt")
    port map(
        clock               => clock(col)(row),
        reset               => reset,
        clock_tx            => clock_rx(col)(row),
        tx                  => rx(col)(row),
        data_out            => data_in(col)(row),
        credit_i            => credit_o(col)(row),
        clock_rx            => clock_tx(col)(row),
        rx                  => tx(col)(row),
        data_in             => data_out(col)(row),
```

```
            credit_o            => credit_i(col)(row),
            address             => open,
            data_write          => open,
            data_read           => (others => '0'),
            write_byte_enable   => open,
            mem_pause_in        => '0'
            );
      end generate plasma_slave;
     end generate col1;
  end generate row1;
end mpsoc;
```

APÊNDICE F – Passos para Construção do Compilador Cruzado

```
export TARGET=mips-elf
export PREFIX=/usr/local/gccmips
export PATH=$PATH:$PREFIX/bin
rm -rf /usr/local/gccmips 2> /dev/null
rm -rf binutils-2.18 2> /dev/null
rm -rf gcc-4.3.0 2> /dev/null
rm -rf newlib-1.16.0 2> /dev/null
rm -rf build-binutils 2> /dev/null
rm -rf build-gcc 2> /dev/null
rm -rf build-newlib 2> /dev/null
tar -xvzf binutils-2.18.tar.gz
tar -xvzf gcc-4.3.0.tar.gz
tar -xvzf newlib-1.16.0.tar.gz
mkdir build-binutils
cd build-binutils
../binutils-2.18/configure --target=$TARGET --prefix=$PREFIX
make MAKEINFO=makeinfo 2>> erro.log
make install
cd ..
mkdir build-gcc
cd build-gcc
 ../gcc-4.3.0/configure --with-newlib --without-headers --enable-languages="c"\
 --target=$TARGET --prefix=$PREFIX --with-gnu-ld --with-gnu-as --disable-libssp
make 2>> erro.log
make install
```

```
cd ..
mkdir build-newlib
cd build-newlib
../newlib-1.16.0/configure --without-fp --target=$TARGET --prefix=$PREFIX
make 2>> erro.log
make install
cd ..
```