Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

ArachNoc : Um processador *manycore* com nós de processamento *multicore* suportando o modelo de programação *IPNoSys*

Laysson Oliveira Luz

Teresina-PI, Setembro de 2016

Laysson Oliveira Luz

ArachNoc : Um processador *manycore* com nós de processamento *multicore* suportando o modelo de programação *IPNoSys*

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ivan Saraiva Silva

Coorientador: Gustavo Girão Barreto da Silva

Teresina-PI

Setembro de 2016

Aos meus pais Francisco Carlos de Moura Luz e Lecimar Oliveira Luz,
ao meu irmão Laécio Oliveira Luz e à todos os demais familiares por sempre estarem comigo em todos os momentos.

Agradecimentos

Agradeço primeiramente a Deus, por tudo que ele tem me proporcionado.

Agradeço aos meus pais, Carlos e Lecimar, pelo auxílio ímpar e incondicional, confiança e conselhos em todos os momentos que precisei.

Aos meu irmão Laécio, por estar sempre presente, acompanhando minha caminhada.

Agradeço ao meu orientador, Ivan Saraiva Silva, por todos os conselhos, pela paciência e ajuda nesse período. Pelas conversas sobre carreira profissional, sobre futebol, política e muito mais. Agradeço, principalmente, pela franqueza presente em seu diálogo, seja em conversas corriqueiras ou em reuniões de trabalho.

Aos meus amigos de laboratório, pelos conselhos e o conhecimento compartilhado em conversas que iam desde implementação de código até discussões políticas e sociais. Agradeço aos meus demais amigos pelos momentos de descontração após horas a fio na frente do computador.

Aos professores, agradeço por tudo, pelo conhecimento transmitido, pela paciência para tirar dúvidas, pelos conselhos e pela amizade construída.

À FAPEPI pelo apoio financeiro para realização deste trabalho de pesquisa.

Muito obrigado!

“Eu sou o caminho,
a verdade,
e a vida,
ninguém vem ao Pai senão por mim”
(João 14.6)

Resumo

Multiprocessadores são arquiteturas com mais de um núcleo de processamento que exploram o paralelismo para fornecer maior desempenho. Eles têm sido pesquisados e implementados na construção de computadores de alto desempenho (em inglês, HPC - *High Perfomance Computers*).

Com muitos núcleos de processamento, a comunicação entre os mesmos é fator crucial para obtenção de alto desempenho. É com base nela que são determinadas a velocidade de processamento, o consumo de energia, o espaço em chip e a escalabilidade. A fim de reduzir a latência na comunicação e aproximar processadores e memórias, pesquisadores têm investido na conexão das unidades de processamento por meio de redes em chip (em inglês, NoC), os chamados MP-SoCs (*Multiprocessor Systems-on-Chip*).

Este trabalho apresenta um MP-SoC com um sistema de comunicação e sincronização entre processos que tem por base o padrão de paralelismo *Fork-Join*. Tal arquitetura, chamada ArachNoc, é composta por nós *multicore* de processamento. Esses nós possuem oito núcleos de processamento cuja microarquitetura é baseada na ISA MIPS, e conferem suporte à programação paralela por meio da inserção das instruções de sincronização.

Para a validação e avaliação de desempenho do sistema proposto, foram desenvolvidas aplicações paralelas. Para a produção destas aplicações foram utilizadas a linguagem C de programação e a biblioteca OpenCL, e para gerar os códigos binários foram desenvolvidos dois compiladores cruzados (do inglês, *cross-compiler*): um compilador *GNU GCC to ArachNoc* e outro *LLVM to ArachNoc*.

O *benchmark* desenvolvido conta com aplicações dos mais diversos níveis de avaliação de desempenho, desde aplicações para apenas validar o sistema em chip, até aplicações para exaurí-lo, isto é, extrair o desempenho máximo do sistema. Algumas delas são: Multiplicação de Matrizes (com diferentes formas de paralelismo), Dijkstra, Algoritmos Genéticos Simples e Algoritmos para processamento de imagens, como o Filtro Laplaciano. Os resultados mostraram que ArachNoC é capaz de executar eficientemente aplicações MIMD e SIMD(OpenCL). Além disso, alguns resultados provaram a flexibilidade do sistema para executar aplicações diferentes ao mesmo tempo, com três diferentes modelos de programação paralela.

Palavras-chaves: Rede em chip. Sistema em Chip. Sistema em chip multiprocessado. Programação Paralela.

Abstract

Multiprocessors are architectures with more than one processing core that exploit parallelism to provide higher performance. They have been researched and implemented in the construction of high-performance computers (HPC).

With many processing cores, the communication between them is crucial to obtain high performance. That is the base to determine the processing speed, the power consumption, the space in chip and the scalability. In order to reduce the communication latency and approach processors and memories, researchers have invested in the connection of the processing units throught networks on chip (NoC), so-called MP-SoCs (*Multiprocessor Systems-on-Chip*).

This paper presents a MP-SoC with a system of communication and synchronization between processes that have as base the Fork-Join parallel pattern. This architecture, so-called ArachNoc, is made up by multicore processing nodes. That nodes have eight processing cores whose microarchitecture is based on the MIPS's ISA, and garantee support to parallel programming by the insertion of synchronization instructions.

For the validation and performance evaluation of the proposed system were developed parallel applications. For the production of these applications were used the C programming language and the OpenCL library, and to generate the binary codes were developed two cross-compilers: one compiler GNU GCC to ArachNoc and other LLVM to ArachNoc.

The benchmark developed have applications for differents levels of performance evaluation, from applications to just validate the system on chip, until applications to doplete it, this is, take out the system maximum performance. Some of them are: Matrix Multiplication (with differents ways to parallelize), Dijkstra, Genetic Algorithms and Algorithms to image processing, like the laplacian filter. The results show that ArachNoC can eficiently run MIMD and SIMD(OpenCL) applications. Besides that, some results asset the system flexibility to execute different applications at the same time, with three different parallel programming models.

Keywords: Network-on-Chip. System-on-Chip. Multiprocessor System-on-Chip. Parallel Programming.

Lista de ilustrações

Figura 1 – Arquitetura de von Neumann. 8
Figura 2 – Gráfico ilustrativo da evolução dos processadores no decorrer do tempo. 8
Figura 3 – Hierarquia de Memória. 10
Figura 4 – Problema de coerência de dados compartilhados. 12
Figura 5 – Representação da rede em chip malha-2D. 18
Figura 6 – Representação da configuração de um Intel Core 2 Duo. 28
Figura 7 – Representação da configuração de um AMD Athlon 64 X2. 29
Figura 8 – Representação da configuração de memória do Intel Nehalem. 30
Figura 9 – Representação de um processador baseado em Sandy Bridge. 31
Figura 10 – Representação em diagrama de blocos de um sistema de processamento utilizando o processador SRP. 32
Figura 11 – Representação em diagrama de blocos da plataforma STORM. 33
Figura 12 – Arquitetura IPNoSys. 34
Figura 13 – Representação esquemática de uma aplicação para IPNoSys. 35
Figura 14 – Representação gráfica de *ArachNid*. 39
Figura 15 – Representação gráfica geral de ArachNoc. 42
Figura 16 – Representação gráfica de um nó escravo de *ArachNoc*. 44
Figura 17 – Representação gráfica de um nó mestre de *ArachNoc*. 46
Figura 18 – Simulação do processo de distribuição de tarefas entre o *buffer* de entrada e os *buffers* de cada escravo do MGTI de um nó escravo. . . . 48
Figura 19 – Representação esquemática do descritor de aplicações. 50
Figura 20 – Representação do formato da instrução Exec. 51
Figura 21 – Representação do formato da instrução Sync. 51
Figura 22 – Representação do formato da instrução SynExec. 52
Figura 23 – Representação do formato das instruções *Send* e *Receive*. 52
Figura 24 – Representação do padrão de programação para ArachNoC. 57
Figura 25 – Gráfico da multiplicação de matrizes de dimensões 16x16. 61
Figura 26 – Aproximações do filtro Laplaciano. 62
Figura 27 – Gráfico da aplicação com Algoritmo de Dijkstra e Filtro Laplaciano. . . 63
Figura 28 – Gráfico do algoritmo genético aplicado ao problema das N rainhas. . . 64
Figura 29 – Gráfico da simulação do algoritmo de *Dijkstra* com o compilador *Ocean*. 65
Figura 30 – ArachNid em módulos VHDL. 67
Figura 31 – Gráfico da simulação de multiplicação de matrizes. 69
Figura 32 – Gráfico da simulação de Smith-Walterman. 69
Figura 33 – Compilação Cruzada 89

Lista de tabelas

Tabela 1 – Tabela de mapeamento de tarefas do MGTI de um nó escravo 53
Tabela 2 – Tabela de controle de pacotes do MGTI do nó mestre após receber SEND. 53
Tabela 3 – Tabela de controle de pacotes do MGTI do nó mestre após receber o RECEIVE. 54
Tabela 4 – Tabela de mapeamento de tarefas do MGTI do nó escravo 1. 55

Lista de abreviaturas e siglas

API *Application Programming Interface*

ArachNoC *ArachNid Network-on-Chip*

ARM *Acorn RISC Machine*

ASIC *Application-Specific Integrated Circuits*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

ccNUMA *cache-coherency Non-Uniform Memory Access*

EDP *Energy-Delay Product*

GCC *GNU Compiler Collection*

GCD *Grand Central Dispatch*

GPU *Graphics Processing Unit*

GPGPU *General Propose Graphics Processing Unit*

HPF *High Performance Fortran*

ILP *Instruction Leve Parallelism*

ISA *Instruction Set Architecture*

Intel ArBB *Intel Array Building Blocks*

MIPS *Microprocessor without Interlocked Pipeline Stages*

MPI *Message Passing Interface*

MPSoC *Multiprocessor System-on-Chip*

NUMA *Non-Uniform Memory Access*

NoC *Network-on-Chip*

MAU *Memory Access Unit*

MIMD *Multiple Instruction, Multiple Data*

MISD	*Multiple Instruction, Single Data*
MGI	Módulo de Gerenciamento de Instruções
MGT	Módulo de Gerenciamento de Tarefas
Ocean	*OpenCL Compiler to ArachNoc*
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
PPI	*Parallel Programming Interfaces*
RLE	*Run Length Encoding algorithm*
RPU	*Routing and Processing Unit*
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
SM	*Streamming Multiprocessors*
SRP	*Speech Recognition Processor*
TCP	Tabela de Controle de Pacotes
TLP	*Thread Level Parallelism*
TBB	*Threading building Blocks*
ULA	*Unidade Lógico-Aritmética*
UMA	*Uniform Memory Access*
VLSI	*Very-Large-Stage Integration*

Lista de símbolos

∂	Letra grega Sigma

Sumário

1 INTRODUÇÃO **1**

Introdução **1**

1.1 Objetivos **3**

Objetivos **3**

1.1.1 Geral 3

1.1.2 Específicos 4

1.2 Considerações Finais **5**

2 FUNDAMENTAÇÃO TEÓRICA **7**

Fundamentação Teórica **7**

2.1 Arquitetura de Computadores **7**

2.2 Hierarquia de Memória **9**

2.3 Modelos de Programação **12**

2.3.1 Programação em Memória Compartilhada 14

2.3.2 Programação em Memória Distribuída 14

2.3.3 Programação em Memória Distribuída Compartilhada 15

2.4 Arquiteturas Paralelas **15**

2.5 Rede em chip **17**

2.6 Considerações Finais **18**

3 ESTADO DA ARTE **21**

Estado da Arte **21**

3.1 Modelos de Programação Paralela **21**

3.1.1 Memória Distribuída 22

3.1.2 Memória Compartilhada 23

3.1.3 Memória Híbrida (Distribuída Compartilhada) 27

3.2 Arquiteturas com suporte a programação paralela **28**

3.3 Modelo de Programação IPNoSys **33**

3.4 Considerações Finais **36**

4 A PLATAFORMA *ARACHNOC* **39**

4.1 O Processador *Multicore ArachNid* **39**

4.2 O *manycore* ArachNoc **41**

4.2.1 Nós de Processamento 42

4.2.1.1 Nó Escravo 43

4.2.1.2 Nó Mestre . . . 45
4.3 Hierarquia de Memória . . . 46
4.3.1 Memória Compartilhada . . . 46
4.4 Gerenciamento de Tarefas . . . 47
4.5 Sistema de Comunicação e Sincronização . . . 48
4.5.1 Descritor de Aplicações . . . 50
4.5.2 Adaptação das instruções de comunicação e sincronização . . . 50
4.5.2.1 Exec . . . 50
4.5.2.2 Sync . . . 51
4.5.2.3 SynExec . . . 52
4.5.2.4 Send e Receive . . . 52
4.6 Ambiente de Desenvolvimento . . . 55
4.6.1 Compilação Cruzada . . . 56
4.6.2 O compilador *Ocean* . . . 57
4.7 Considerações Finais . . . 58

5 RESULTADOS . . . 59
5.1 Experimentos com Compilador Cruzado . . . 59
5.1.1 Multiplicação de Matrizes . . . 60
5.1.2 Algoritmo de *Dijkstra* e Filtro Laplaciano . . . 61
5.1.3 Algoritmo Genético . . . 63
5.1.4 Experimentos com Compilador *Ocean* . . . 64
5.2 Implementação VHDL . . . 66
5.2.1 Experimentos com Compilação Cruzada . . . 66

Conclusão e Trabalhos Futuros . . . 71

REFERÊNCIAS . . . 73

APÊNDICES 79

APÊNDICE A – MULTIPLICAÇÃO DE MATRIZES EM C . . . 81

ANEXOS 87

ANEXO A – COMPILADOR CRUZADO GCC . . . 89

ANEXO B – *LINKER SCRIPT* . . . 93

1 Introdução

O desenvolvimento de processadores de alto desempenho tem sido uma exigência desde o surgimento do primeiro microprocessador integrado em um *chip* (ASPRAY, 1997). O aumento de desempenho era obtido, principalmente, por meio do aumento da frequência de operação, cuja obtenção implica na inserção de transistores no *chip*. O que exige a redução das dimensões físicas dos transistores.

Em busca de desempenhos cada vez melhores, foram desenvolvidos *chips* com frequências cada vez maiores. Com o aumento da frequência de operação, foi verificado aumento da dissipação de potência e do consumo de energia, o chamado *Power Wall*.

Quando se tornou impossível continuar aumentando a frequência de operação dos processadores, o foco passou a ser a integração de mais de um núcleo de processamento em um único *chip*. Essas novas arquiteturas ficaram conhecidas como processadores *multicore*, cujos núcleos operam em menor frequência, e o ganho de desempenho é obtido por meio da exploração do paralelismo.

Os processadores *multicore* necessitam de sistemas para conectar os núcleos de processamento, e para coordenar o funcionamento dos mesmos. Um sistema de conexão simples consiste em conectar todos os núcleos de processamento por meio de barramentos. Essa é uma configuração de baixa escalabilidade, na qual a inserção de cada vez mais núcleos provoca um gargalo no sistema caso a largura de banda do barramento não seja ampliada proporcionalmente.

Um dos primeiros *multicore* de que se tem conhecimento tinha 64 unidades lógico-aritméticas (ULAs) controladas por quatro unidades de controle (BOUKNIGHT et al., 1972). As unidades de controle podiam também realizar operações escalares em paralelo com o vetor de ULAs. Ele realizava operações em vetores, e *arrays*, em paralelo.

A necessidade por maior poder de processamento e maior paralelismo de tarefas, estimulou o desenvolvimento de sistemas cujas unidades de processamento não são apenas ULAs, mas CPUs conectadas à memórias, isto é, computadores completos. São os chamados sistemas multiprocessados em *chip* ou MPSoCs (*Multiprocessors Systems-on-Chip*), que são infraestruturas VLSI (*Very Large Scale Integration*) compostas por múltiplos processadores programáveis (WOLF; JERRAYA; MARTIN, 2008).

Por muito tempo os barramentos foram utilizados como solução de interconexão. No entanto, com o aumento do número de núcleos no *chip* e o surgimento dos chamados *manycore*, os barramentos passaram a ser insuficientes (ZEFERINO, 2003). Para solucionar esse problema surge o conceito de **rede em *chip*** (BENINI; MICHELI, 2002), a qual,

permite a escalabilidade do sistema e suporta o paralelismo necessário.

As redes em *chip* são formadas por um conjunto de roteadores conectados entre si e a nós de processamento. Dessa forma, cada roteador compartilha dois canais físicos com cada um dos outros roteadores adjacentes a ele, e possui um canal de comunicação com o nó de processamento. É a chamada **comunicação ponto-a-ponto**. Um roteador é capaz de transmitir, ao mesmo tempo, dados para todos os outros roteadores adjacentes a este, pois existem portas de comunicação e *buffers* de mensagens, entre este e cada um dos seus vizinhos. Isso permite o **paralelismo de comunicação**.

Na maioria dessas redes é comum que os roteadores estejam limitados a função de encaminhar pacotes, como em (ZEFERINO; SUSIN, 2003), ou seja, eles apenas guiam os pacotes até os nós de processamento. Os nós de processamento são mais robustos que os roteadores, portanto ocupam mais espaço em *chip*. De acordo com o modelo de programação e processamento, algumas arquiteturas possuem roteadores capazes de processar instruções. Araujo (2012) apresenta uma arquitetura não-convencional chamada IPNoSys, na qual, não há processadores, apenas memórias e roteadores. Nessa plataforma, as tarefas são processadas ao longo da rede em *chip*, instrução por instrução, a medida em que viajam na rede através dos roteadores. Dessa forma, além de encaminhar pacotes, os roteadores são responsáveis por executar instruções.

Embora as redes em *chip* proporcionem desempenho elevado e boa reusabilidade, elas precisam ser bem projetadas para que os nós da rede se comuniquem de forma eficiente, evitando perda de dados. Um exemplo de rede em *chip* com protocolo de comunicação que busca evitar gargalos é apresentada por Zeferino e Susin (2003). A rede não pode permitir perda de dados, deve reduzir os atrasos de comunicação e evitar o bloqueio do sistema. Por isso a comunicação entre os nós da rede e a memória tem sido objeto de muitos estudos (SCHAUER, 2008).

No que diz respeito ao projeto das redes em *chip*, em particular, a funcionalidade está consolidada. Entretanto, com relação ao projeto das unidades de processamento, discutem-se aspectos relacionados à complexidade do caminho de dados, reconfiguração e heterogeneidade. Programabilidade de arquiteturas multiprocessadas e hierarquia de memória também são assuntos que merecem muita atenção e esforço de pesquisa.

Dentre os sistemas multiprocessados existem dois modelos baseados na configuração da memória, compartilhada ou distribuída. O modelo de memória determina também a programabilidade do sistema, pois sistemas cuja a memória é compartilhada permitem que os processos compartilhem dados. Se a memória é distribuída, os processos precisam trocar dados diretamente de um para outro. No caso dos sistemas com memória compartilhada, problemas de consistência de dados são frequentes, é necessário garantir que um dado alterado por qualquer um dos núcleos de processamento seja sempre corretamente atualizado antes de ser acessado novamente.

A crescente disparidade entre as frequências de *clock* do processador e da memória, *memory-wall* (MCKEE, 2004), motiva a inserção de memórias com tecnologia que as aproximam do processador e agilizam o acesso aos dados.

Nos sistemas multiprocessados com memória compartilhada, como existem vários processadores que necessitam acessar a uma mesma memória, algumas tecnologias de acesso são adotadas para solucionar o problema do atraso no acesso à memória: UMA (*Uniform Access Memory*) e NUMA (*Non-Uniform Access Memory*). Arquiteturas UMA possuem processadores iguais, com tempos de acesso à memória, iguais. Arquiteturas NUMA definem uma memória logicamente compartilhada, porém fisicamente distribuída. Isto é, cada processador possui uma porção da memória compartilhada conectada a si. Ainda que cada processador possua seu próprio espaço de endereçamento, todo processador tem acesso às memórias de todos os outros (MANCHANDA; ANAND, 2012).

Muitos MPSoCs são compostos por unidades de processamento mononucleadas, e, quando submetidos a aplicações com mais tarefas que a quantidade de núcleos de processamento, provocam alto tráfego na rede. Apesar disso, são processadores de alto desempenho, porém é possível aumentar o poder de processamento desses sistemas por meio da inserção de núcleos às unidades de processamento do mesmo, constituindo assim *manycores* cujas unidades de processamento são *multicores*.

Tendo isso em vista, este trabalho apresenta um processador *manycore* com nós de processamento *multicore*, isto é, cada nó é composto por 8 núcleos *MIPS-based* - um *MIPS* multiciclo adaptado para comunicação entre processos - e duas memórias compartilhadas, uma de dados e uma de instruções. Esta arquitetura confere suporte a sistema operacional e a três modelos de programação paralela (compartilhamento de variáveis, troca de mensagens e *OpenCL*). Dessa forma, esse *manycore* é capaz de executar, eficientemente, aplicações com diferentes modelos de paralelismo.

1.1 Objetivos

1.1.1 Geral

O desenvolvimento da programação paralela ganhou força com o desenvolvimento de processadores *manycore*. Existe uma variedade de *manycore*, cada um com suas especificidades. Porém, todos eles executam multitarefas, e o sincronismo entre os processos exige que os núcleos de processamento se comuniquem.

Em processadores com memória distribuída, cada núcleo possui um espaço de endereçamento privado, e portanto, caso um núcleo necessite de dados que estejam armazenados na memória de outro núcleo, ele precisa requisitar o envio desses. Dessa forma, os núcleos só podem se comunicar através da **troca direta de mensagens**.

Processadores com memória compartilhada permitem que os núcleos de processamento se comuniquem através da mesma. Isto é, os núcleos de processamento podem trocar dados por meio de leitura e escrita na memória. Esse sistema de comunicação é conhecido como **compartilhamento de variáveis**.

Portanto, com um *manycore* com unidades de processamento *multicore*, há um elevado ganho de desempenho em relação a processadores com unidades de processamento mononucleadas. Além disso, para se explorar toda a capacidade desse processador é apresentada uma metodologia de programação.

Este trabalho apresenta um sistema em *chip* multiprocessado, chamado *ArachNoc*, no qual, cada nó de processamento possui uma memória privada. Porém, cada nó é um processador *multicore*, no qual, os núcleos compartilham esse espaço de endereçamento. Esse trabalho apresenta, também, uma metodologia de programação paralela.

1.1.2 Específicos

A plataforma IPNoSys possui um modelo de programação paralela baseado na troca direta de mensagens entre os processos. Para essa finalidade, a plataforma dispõe de um subconjunto de instruções, de comunicação e sincronização, que permite que os processos enviem e recebam dados uns dos outros, e que se comuniquem de modo a manter um sincronismo na execução do programa como um todo.

O modelo de programação de IPNoSys é oriundo de uma plataforma não-convencional, isto é, um sistema de processamento que não segue o modelo de Von Neumman, desenvolvida por um membro do grupo de pesquisa. Porém, *AracNoc* é um processador convencional, portanto, a adoção de tal modelo de programação exigiu algumas modificações.

Para validação e avaliação de desempenho do sistema em *chip* foi desenvolvido um ambiente de programação para *ArachNoc*. Esse ambiente é composto por: um compilador cruzado *GCC to MIPS*, bibliotecas de sincronização, e uma metodologia de programação, todos desenvolvidos no escopo deste trabalho.

Além das ferramentas anteriores, para a execução de aplicações SIMD, foi utilizado um compilador *OpenCL* to *MIPS*, desenvolvido por outro membro do grupo de pesquisa, Jônatas, em seu escopo de trabalho de mestrado.

Uma vez apresentados os meios de experimentação do sistema proposto, este trabalho visa apresentar os resultados obtidos com a construção do sistema em *chip*, alguns deles são: o grande poder de processamento, o aumento na quantidade de aplicações executadas por unidade de tempo e a eficiência do mecanismo de comunicação e sincronização adotado.

1.2 Considerações Finais

Maior poder de processamento é uma ambição comum a pesquisadores e indústrias. Quanto ao *hardware*, a solução mais simples aparente tem sido adicionar cada vez mais núcleos de processamento, porém, quanto mais núcleos se adiciona, mais complexo se torna coordenar todos eles. Quanto a *software*, a programação paralela ainda é um ponto fraco, que segue em constante desenvolvimento a fim de se obter o melhor uso da memória e a melhor distribuição de carga de trabalho entre os núcleos de processamento.

Portanto, *ArachNoc* possui núcleos *multicores* coordenados sob um regime mestre-escravo e um sistema de interconexão que confere escalabilidade ao sistema. Esse, com um subconjunto de instruções de comunicação e sincronização, confere ainda, suporte à programação paralela por meio do compartilhamento de variáveis.

2 Fundamentação Teórica

Sistemas em *chip* multiprocessados (MPSoC) são fundamentalmente compostos de elementos como: nós de processamento, que podem ser compostos por um ou mais processadores ou outros elementos de processamento; um subsistema de comunicação, através do qual os nós trocam dados; e uma hierarquia de memória, onde os dados são armazenados.

Tendo isso em vista, a seguir passa-se à exploração de alguns conceitos fundamentais para se conhecer todos os componentes de um MPSoC, a fim de se compreender como se constrói um sistema multiprocessado.

2.1 Arquitetura de Computadores

Em 1945 von Neumann definiu o que viria a se tornar a estrutura básica de um computador (NEUMANN, 1993). Computadores são constituídos de dispositivos de entrada, de saída, uma memória, que armazena instruções e dados; e uma unidade central de processamento que recebe os dados de entrada e gera os dados de saída. Na Figura 1 temos uma ilustração da estrutura básica de um computador.

Um processador possui uma estrutura definida por elementos de lógica sequencial e elementos de lógica combinacional. Tais elementos, dependendo da organização, compõem uma microarquitetura específica. Toda microarquitetura possui uma ISA, isto é, um conjunto de instruções que a arquitetura desenvolvida deverá executar (HENNESSY; PATTERSON, 2011).

A microarquitetura dos processadores é composta por uma **unidade operativa** e uma **unidade de controle**. O funcionamento do processador é determinado pela unidade de controle, que gera sinais que coordenam o funcionamento da unidade operativa. Inicialmente, somente os projetistas conseguiam manusear os processadores, por meio da microprogramação, isto é, definindo uma sequência de passos que, a cada ciclo de operação, determinam os estados dos sinais de controle do processador.

Posteriormente, a programação dos processadores se tornou mais fácil com a idéia de programa armazenado em memória, difundida por von Neumann. O programa é um conjunto de instruções que são lidas e executadas pelo processador em uma sequência de passos definida pela microarquitetura e o conjunto de instruções. Dessa forma, a programação pôde ser desenvolvida e foi ganhando popularidade, até alcançar as linguagens de alto nível, facilmente acessíveis atualmente.

Processadores desenvolvidos para atender necessidades amplas, são denominados

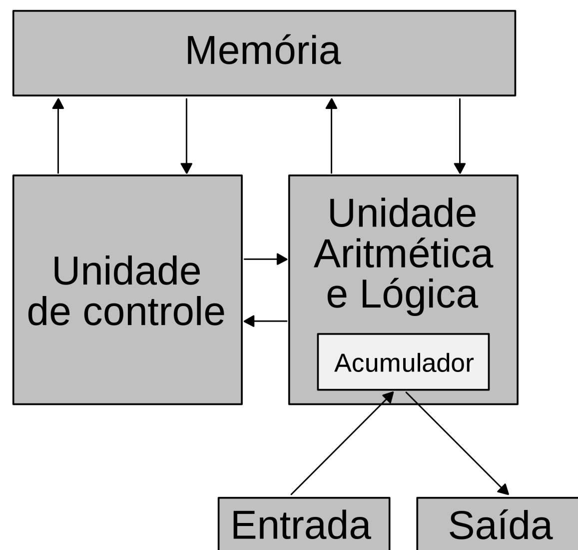

Figura 1 – Arquitetura de von Neumann.

processadores de propósito geral. Tais dispositivos dissipam muita potência e consomem muita energia. Moore, analisando em 1965 os processadores já produzidos, concluiu que, a cada 18 meses, o número de transistores disponível para implementação dos circuitos dobraria. O desenvolvimento de processadores tem comprovado esta previsão (MOORE, 1965).

As tecnologias de fabricação de circuitos integrados foram sendo aprimoradas a fim de diminuir o tamanho dos transistores, no intuito de inserir cada vez mais transistores no *chip*. A frequência de operação tornou-se cada vez maior, possibilitando o desenvolvimento de unidades de processamento de melhor desempenho. A observação de Moore ficou conhecida como **Lei de Moore** e seu resultado pode ser visto na Figura 2.

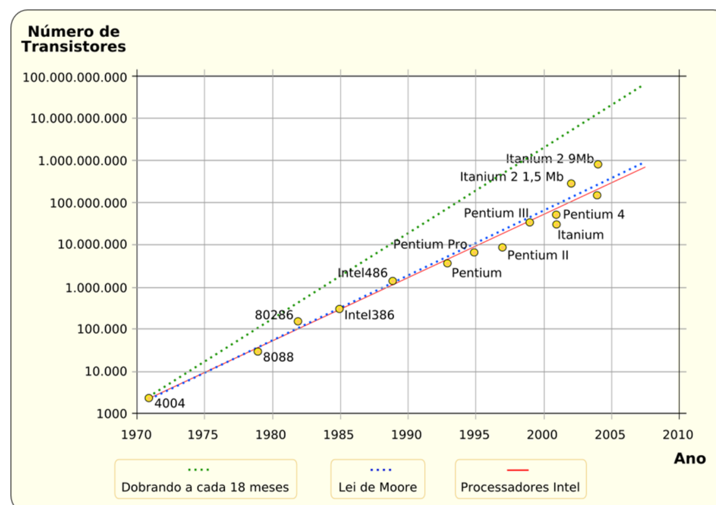

Figura 2 – Gráfico ilustrativo da evolução dos processadores no decorrer do tempo.

A fim de aumentar a capacidade dos processadores com um único núcleo, foram pro-

postas evoluções nas microarquiteturas dos processadores, tal como a microarquitetura com *pipeline* (HENNESSY; PATTERSON, 2011). Técnica por intermédio da qual a execução de uma instrução é dividida em estágios: busca, decodificação, execução, escrita na memória e registro de resultado. Assim, o processador estruturado em unidades independentes, pode executar as partes de várias instruções em paralelo. Isto é, enquanto o processador está executando uma soma na ALU, o decodificador já verifica os operadores de uma instrução seguinte e o processador já busca uma terceira instrução na memória. Esta técnica assemelha-se à filosofia de trabalho bastante difundida por Henry Ford, o *fordismo* (DRUCK, 2005), onde cada operário adiciona uma parte do carro ao que seu antecedente montou, e com um conjunto de operários em sequência, vários carros vão sendo montados simultaneamente.

Além da técnica de *pipeline*, pesquisadores passaram a investir em processadores com caminho de dados múltiplos, onde há mais de uma ALU (*Arithmetic Logic Unit*), mais de um decodificador e mais de um banco de registradores, pois assim é possível decodificar e executar mais de uma instrução por vez. São os chamados processadores super-escalares (SMITH; SOHI, 1995). Replicar o caminho de dados aumenta o espaço ocupado em chip, entretanto, o número de instruções disponíveis no código, para execução paralela, é limitado.

O ganho de desempenho dos processadores *pipeline* e super-escalares foi bastante expressivo, mas a demanda por maior velocidade de processamento permanecia. Percebeu-se que o consumo de energia e a dissipação de potência, principalmente, tinham atingido níveis proibitivos. Dessa forma, não era mais possível continuar aumentando a frequência. A solução encontrada foi limitar a frequência de operação dos núcleos e inserir mais de um núcleo por *chip*. A melhoria de desempenho foi então possibilitada pelo paralelismo e não pelo aumento da frequência de operação, como comentado por Blake, Dreslinski e Mudge (2009), Schauer (2008).

2.2 Hierarquia de Memória

O modelo de von Neumann lançou o conceito de programa armazenado. Assim, espera-se que as instruções executadas por um processador, bem como os dados, estejam armazenados na memória.

As memórias foram desenvolvidas para armazenar a maior quantidade de dados possível e fornecê-los em tempo hábil ao processador. Ocorre entretanto, que essas possuem atraso entre a requisição de determinado dado e a entrega do mesmo, que é muito grande quando comparado com a frequência do processador.

As memórias podem ser classificadas quanto a sua velocidade de acesso e sua capacidade de armazenamento, de acordo com a tecnologia utilizada para implementar

(HENNESSY; PATTERSON, 2011). Existem diferentes tecnologias para implementação de memórias. As tecnologias mais rápidas são mais caras, por unidade de armazenamento, e as mais lentas, são mais baratas. O objetivo é ter-se memórias com acesso muito rápido e grande capacidade de armazenamento, e isso se consegue com uma **hierarquia de memória**. É através da combinação de memórias com diferentes tempos de acesso, para tecnologias diferentes, com custos diferentes e políticas de uso (mapeamento, política de substituição, política de escrita) que se consegue a impressão de alta capacidade, alta velocidade e baixo custo para o usuário.

As diferentes tecnologias de memória são representadas em uma pirâmide. No topo da pirâmide encontram-se as memórias cuja tecnologia permite maior velocidade de acesso, portanto, mais caras e, devido a isso, menor capacidade de armazenamento deste tipo de memória é inserido no sistema (registradores). Na base da pirâmide encontram-se as memórias com tecnologia de menor velocidade de acesso, portanto mais baratas e, devido a isso, maior capacidade de armazenamento desse tipo de memória, é inserido no sistema (disco-rígido, fitas magnéticas, mídias removíveis), como representado na Figura 3.

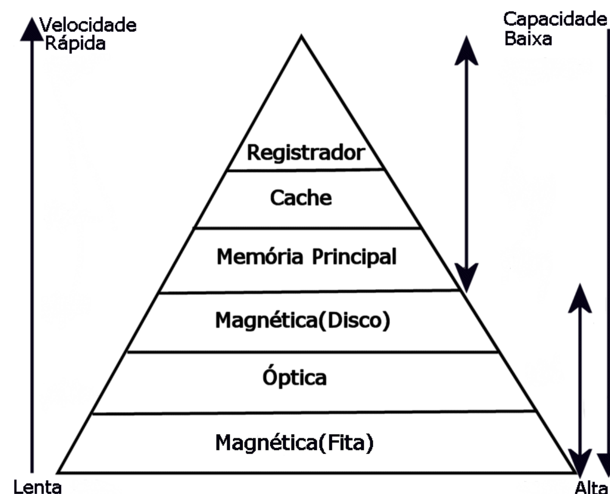

Figura 3 – Hierarquia de Memória.

O custo em tempo de acesso do processador às memórias RAMs é muito alto e limita o desempenho do núcleo. Durante a execução de programas, se o processador acessa um dado, ele tende a voltar a acessar esse dado (laços de repetição, por exemplo). Tal afirmação é conhecida como **princípio da localidade temporal**. Se um programa, no acesso a uma região do espaço endereçável, referenciar um dado, dados cujos endereços são próximos a este, tenderão a ser referenciados (dados em um *array*, por exemplo). Essa máxima é conhecida como **princípio da localidade espacial**.

Diante da confirmação dos princípios da localidade e mediante a necessidade de maior velocidade de acesso aos dados guardados na memória principal, foi proposta a memória cache (IBM, 1964). A cache é uma memória menor que a memória principal, porém de maior velocidade de acesso. A cache tem a função de armazenar parte dos dados

da memória principal que o processador está acessando, isto é, os dados da região de memória que representa a localidade de acesso à memória, em um dado instante.

O espaço de endereçamento da cache, bem como o da memória principal, é organizado em porções de dados (dados de um conjunto de endereços contíguos) que também tentam representar a localidade. Esta porção de dados é conhecida como bloco ou linha. Assim, quando um dado vai ser substituído, todos os dados do bloco ao qual o dado pertence são substituídos por todos os dados do bloco que inclui o dado referenciado. Neste caso, a cache executará um algoritmo para escolher o dado que será substituído pelo novo dado referenciado, chamado de **algoritmo de substituição de páginas**.

Para um bom funcionamento da hierarquia de memória, faz-se necessário definir a política de escrita na cache (*write-through* ou *write-back*). Se *write-through*, o dado alterado na cache deve ser imediatamente atualizado na memória principal; se *write-back*, o bloco alterado na cache só será atualizado na memória quando da substituição do mesmo. Além da política de escrita, o algoritmo de substituição e o tipo de mapeamento de blocos da memória principal para os blocos da memória cache são parâmetros fundamentais para o bom desempenho do modelo de memória (GIRAO, 2009).

Em sistemas multiprocessados a organização da memória cache é fundamental para o desempenho do sistema. Imagine-se uma cache L2 como um espaço de endereçamento compartilhado. Nesse caso, os processadores terão que competir pelo acesso à essa memória, o que faz com que essa memória seja um “gargalo” para o sistema. Agora, considere-se que essa mesma cache possua espaço de endereçamento distribuído. Nesse caso, apesar de não haver concorrência no acesso, necessita-se de maior fluxo de dados entre memória principal e os blocos de cache, pois todo dado alterado em uma instância da cache deve, em algum momento, ser atualizado na memória principal e, consequentemente, nos outros blocos de cache.

Em sistemas multiprocessados existem quatro diferentes formas de implementar a memória cache. São elas:

- Cache fisicamente compartilhada com espaço de endereçamento compartilhado;
- Cache fisicamente compartilhada com espaço de endereçamento distribuído;
- Cache fisicamente distribuída com espaço de endereçamento compartilhado;
- Cache fisicamente distribuída com espaço de endereçamento distribuído;

Uma cache L2 fisicamente distribuída com espaço de endereçamento compartilhado, com blocos instanciados em diferentes nós da rede, reduz o congestionamento da rede durante acessos á memória, desde que os núcleos de processamento possuam caches de nível inferior. Com caches L1, tais núcleos são capazes de armazenar blocos da cache L2 e,

portanto, reduzir o fluxo de dados entre memórias e agilizar o acesso aos dados, pois estes encontram-se mais próximos dos processadores.

Em sistemas multiprocessados, onde cada núcleo de processamento possui uma cache privada, o problema de coerência de cache é bastante comum. Nesse caso, faz-se necessário atualizar os dados da cache L1 e os da cache L2, e isso não pode ser garantido apenas pela política de escrita.

A política de escrita na cache não sana o problema de coerência de cache, pois, seja ela *write-through* ou *write-back*, em ambos os casos, poderá ocorrer incoerência de cache uma vez que ela pode ocorrer não só entre a cache e a memória como também entre as caches dos diferentes processadores.

Por exemplo, no caso do uso de dados compartilhados, o problema da incoerência pode ocorrer da seguinte maneira: considerando-se dois processadores, cada um com sua própria cache e ambos compartilhando a memória principal, supõe-se que exista uma estrutura de dados X que é referenciada por ambos os processadores e ambas as cópias estão coerentes (Figura 4(a)). Em caso de uma escrita por um dos processadores, se a cache utilizar a política de *write-through*, a memória terá a atualização do bloco (modificando a estrutura para X', por exemplo), entretanto a outra cache que contém este dado não terá sido atualizada (Figura 4(b)). Caso a cache utilize uma política de *write-back*, tanto a cache do outro processador quanto a própria memória ficarão desatualizadas (apesar de que esta última será atualizada quando a cache substituir este bloco) (Figura 4(c)).

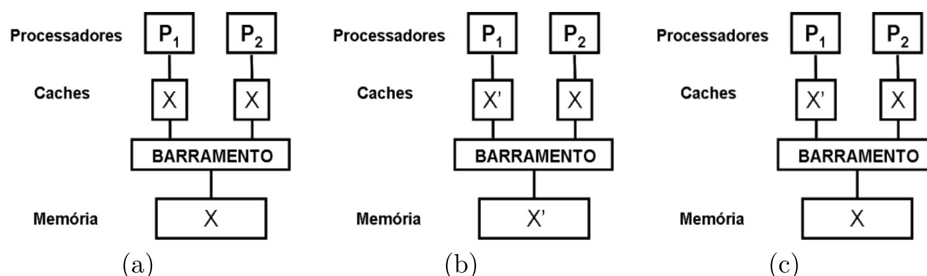

Figura 4 – Problema de coerência de dados compartilhados.

Fonte – (GIRAO, 2009)

Para se garantir a coerência da cache em sistemas multiprocessados existem outros modos, como *snoop* e diretório (GIRAO, 2009).

2.3 Modelos de Programação

A programação para processadores *multicore* é fortemente influenciada pela arquitetura dos mesmos. Isto é, pela forma como a memória está organizada e pelo sistema de comunicação entre processos.

A programação paralela se desenvolveu de forma mais lenta que o desenvolvimento de novas arquiteturas. A busca pela programação paralela mais cômoda e mais eficiente possível se inicia nos modelos de programação e vai até o compilador, passando pelas linguagens, bibliotecas e outros recursos.

Para a obtenção de códigos paralelos há duas opções: paralelizar códigos lineares ou desenvolvê-los por meio de linguagens paralelas. Ainda que a paralelização de códigos lineares satisfaça a programação multicore, a mesma está limitada às restrições das linguagens, que podem limitar o nível de paralelismo das aplicações desenvolvidas. Dessa forma, muitos *frameworks* e bibliotecas têm sido desenvolvidos para permitir a programação paralela, alguns deles são discutidos em (DIAZ; MUNOZ-CARO; NINO, 2012; SANCHEZ et al., 2012). Além disso, diversas linguagens de programação paralela como *OpenCL* (STONE; GOHARA; SHI, 2010) e HPF (*High Perfomance Fortran*) (KENNEDY; KOELBEL; ZIMA, 2007), têm sido desenvolvidas para que a paralelização seja a melhor possível e possa-se explorar o máximo do hardware.

Os recursos para a paralelização da aplicação ficam a cargo da linguagem de programação, muito por determinismo do *framework* ou biblioteca utilizada (DIAZ; MUNOZ-CARO; NINO, 2012). São os chamados **padrões de paralelização**, cujos exemplos mais conhecidos são:

i) SPMD (*Single Program Multiple Data*): o mesmo programa é designado para diferentes processadores executarem-no com diferentes dados. Ex: OpenCL e CUDA;

ii) Mestre/Escravo: um processo mestre ordena a execução de vários processos escravos em paralelo e possui uma “bolsa de tarefas”;

iii) Paralelismo de repetição: iterações de um ou mais laços de repetição são executadas em paralelo;

iv) *Fork-Join*: um processo inicial se divide em processos concorrentes que posteriormente se unem em um processo finalizador.

Cada padrão de programação é melhor adaptado a determinado modelo de programação.

Em sistemas *multicore* e multiprocessados a configuração da memória do sistema é determinante na escolha do modelo de programação, pois este é uma consequência direta da estrutura de memória do sistema. Caso os procesadores compartilhem um espaço de endereçamento, o sistema deverá ser programado por meio de um modelo que explore a comunicação por **compartilhamento de variáveis**, porém, se os processadores possuírem memórias privadas, o modelo de programação a ser adotado explorará a capacidade dos mesmos de se comunicarem diretamente, por meio de **troca de mensagens**.

2.3.1 Programação em Memória Compartilhada

Arquiteturas com modelo de memória compartilhada, no qual os núcleos de processamento competem pelo mesmo espaço de endereçamento, são também conhecidas como multiprocessadores e conferem suporte à troca de informações entre os processos por meio do compartilhamento de variáveis, isto é, os processadores escrevem e lêem dados compartilhados na memória.

Para sistemas de comunicação entre processadores por meio do compartilhamento de variáveis, um dos modelos de execução paralela mais conhecido e estabilizado é o *POSIX Threads* (BUTENHOF, 1997a). No Pthreads, há várias *threads* executando em paralelo que podem ser controladas independentemente. Tal modelo é viabilizado por uma biblioteca (**pthread.h**) que possui um conjunto de tipos e chamadas de procedimentos que permitem criar, destruir e coordenar as atividades das *threads* por meio de construções que garantem o acesso exclusivo a regiões de memória selecionadas. Este modelo é apropriado ao padrão de programação *Fork-Join*.

Há ainda o modelo de programação OpenMP (DAGUM; MENON, 1998), o qual não é apenas uma biblioteca, mas um framework para as linguagens C, C++ e Fortran. Esse modelo possui um conjunto de diretrizes para o compilador, chamadas **pragmas**, e fornece gerenciamento de uma *thread pool*, que é, um conjunto de *threads* a serem liberadas para execução. Pragmas são palavras-chave que indicam ao compilador que o trecho de código seguinte ao pragma, deve ser tratado de maneira específica, como declarações de variáveis globais, locais, ou como um processo indivisível, ou mesmo um processo a ser paralelizado. Apesar de *OpenMP* disponibilizar *threads* em um nível mais abstrato que *POSIX Threads*, *OpenMP* também é apropriado ao padrão de programação *Fork-Join*.

Muitas *APIs* são bastante conhecidas, e por funcionarem com o suporte de linguagens de programação consolidadas, se popularizaram. Sanchez et al. (2012) apresenta e avalia o desempenho de algumas delas.

2.3.2 Programação em Memória Distribuída

Plataformas com modelo de memória distribuída, as quais também é dado o nome de multicomputadores, não conseguem compartilhar variáveis, pois cada processador possui seu próprio espaço de endereçamento. Dessa forma, os processos se comunicam por meio da troca direta de mensagens, para tal, os mesmos contam com as chamadas **redes de interconexão**.

Para arquiteturas com memória distribuída, a programação de aplicações baseia-se nessa troca direta de mensagens, e o modelo de programação mais conhecido para estes sistemas é o *Message Passing*. Para esse modelo, consolidou-se um padrão de interface definido, chamado *MPI* (*Message Passing Interface*).

MPI (SNIR et al., 1998) não é uma linguagem, mas uma biblioteca, a qual especifica nomes, sequências de chamada e resultados de sub-rotinas ou funções a serem chamadas a partir de programas Fortran, C ou C++. Os programas podem, portanto, serem compilados com os compiladores originais das linguagens, mas precisam ser ligados com a biblioteca MPI.

2.3.3 Programação em Memória Distribuída Compartilhada

Em arquiteturas híbridas, como o Intel Sandy Bridge (MOLKA; HACKENBERG; SCHöNE, 2014) e o AMD Fusion (BRANOVER; FOLEY; STEINMAN, 2012), há suporte aos dois meios de comunicação entre processos, anteriormente citados. Tais arquiteturas possuem um modelo de memória híbrida, com a qual busca-se combinar as qualidades de ambos modelos, as variáveis em memória e a facilidade de programar, de memória compartilhada, com a escalabilidade de modelos com memória distribuída.

Entretanto, além do desenvolvimento de novas bibliotecas e linguagens de programação, é possível confiar na mistura dos modelos e ferramentas de programação já disponíveis. Esta abordagem é conhecida como **programação paralela híbrida** (STERLING; MESSINA; SMITH, 1995). Esse modelo de programação é o mais utilizado pelas arquiteturas que combinam GPU com CPU, para processamento de imagens, principalmente. São as chamadas GPGPU's, que compõem as placas de vídeo. A idéia básica é usar troca de mensagens (*MPI*, por exemplo) através dos nós distribuídos e modelos de memória compartilhada (como *Pthreads e OpenMP*) dentro de um nó do sistema.

Com a finalidade de atender às novas arquiteturas que possuem configuração híbrida, diversas linguagens e *frameworks* foram desenvolvidos, como *CUDA e OpenCL*, por exemplo, apesar de terem sido desenvolvidos para GPUs e GPUs+CPU. Porém, diversas combinações de modelos têm sido implementadas, como *CUDA e MPI* usado em (CHEN; ZHANG, 2009), *CUDA e OpenMP* aplicado em (WANG et al., 2009), *MPI e OpenMP* (SMITH; BULL, 2001) e *MPI e Pthreads* (JOHNSON, 2007), buscando sempre o melhor desempenho da arquitetura híbrida alvo.

Devido ao crescente interesse nas arquiteturas com modelo de memória híbrida, principalmente por atender a um mercado como é o de processadores para placas-de-vídeo, pesquisadores têm buscado a melhor combinação de *APIs* para programação paralela híbrida. Diaz, Munoz-Caro e Nino (2012) apresentam e discute sobre algumas dessas *APIs* de programação paralela para arquiteturas híbridas.

2.4 Arquiteturas Paralelas

Arquiteturas paralelas são sistemas computacionais constituídos por diversas unidades de processamento. Tais arquiteturas são regidas pela ideologia de "dividir para

conquistar", isto é, dividir uma aplicação em partes que serão executadas paralelamente. O resultado final da aplicação é obtido pela união dos resultados parciais, ou seja, dos resultados de cada execução paralela.

Em 1966, Michael Flynn propôs uma classificação de arquiteturas de computadores, que se tornou conhecida como taxonomia de Flynn (FLYNN, 1972). A taxonomia de Flynn se baseia no fluxo de instruções e de dados processados pelas arquiteturas. As classes definidas por Flynn são:

- SISD - *Single Instruction Single Data*: nesta classe estão as arquiteturas tradicionais, as quais realizam uma única operação sobre um único dado, ou um par, como em uma operação de adição. Ex: Máquina de von Neumann;
- SIMD - *Single Instruction Multiple Data*: arquiteturas dessa classe executam um única operação sobre um conjunto vasto de dados. Ex: Arquiteturas vetoriais;
- MISD - *Multiple Instruction Single Data*: arquiteturas dessa classe executariam múltiplas instruções sobre um único dado ou um conjunto resumido destes. Alguns afirmam que não existe;
- MIMD - *Multiple Instruction Multiple Data*: classificação para a realização de diferentes operações sobre diversos dados. Modelo clássico de paralelismo de tarefas (operações e dados) distintas em núcleos distintos. Ex: Multiprocessador, Multicomputador.

As arquiteturas SISD, são aquelas que seguem o modelo de von Neumann. Constituídas por uma memória para armazenamento de instruções e dados, uma Unidade Central de Processamento (UCP) e dispositivos de entrada/saída, todos conectados por um subsistema de comunicação. Em arquiteturas monoprocessadas, com poucos dispositivos de E/S, o subsistema de comunicação é o barramento. A UCP, por sua vez, é constituída de um bloco de controle e um caminho de dados ou parte operativa.

Os computadores paralelos, que possuem múltiplos fluxos de instruções operando sobre múltiplos fluxos de dados, pertecem a categoria de Flynn chamada MIMD. Esses computadores vão desde sistemas com dois processadores conectados via barramento até supercomputadores com milhares de processadores interligados por complexas redes de interconexão.

Existem dois níveis de paralelização de aplicações que se tornaram mais populares, são eles: ILP (*Instruction Level Paralelism*) e TLP (*Thread Level Paralelism*). Todo programa consiste em um conjunto de instruções. No paralelismo a nível de *thread*, TLP, o programa pode ser dividido em tarefas que podem ser paralelizadas, com ou sem comunicação ou sincronização. Ao passo que, no paralelismo a nível de instrução, ILP,

não há tarefas, de fato, a serem paralelizadas, há apenas instruções individuais que não apresentam dependência e por esta razão podem ser paralelizadas.

O nível de paralelismo é característica chave no desenvolvimento de aplicações para arquiteturas paralelas, exatamente porque é a técnica de paralelização da aplicação que define como os núcleos de processamento serão utilizados (SANCHEZ et al., 2012; SHEKHAR et al., 2011). Porém, a arquitetura alvo define a forma como a aplicação paralela será implementada.

As máquinas mononucleadas com mais de um caminho de dados (parte operativa), como os processadores superescalares, favorecem o paralelismo a nível de instruções (ILP). As máquinas paralelas MIMD, como processadores *multicore* e *manycore*, favorecem o paralelismo a nível de *threads* (TLP).

2.5 Rede em chip

Inicialmente, os MPSoCs e processadores *multicore* tinham seus núcleos conectados por meio de barramentos. Barramentos são canais de trocas de dados, utilizados para comunicar processadores, memórias e outros dispositivos. Os barramentos transportam dados de todos os dispositivos para todos os dispositivos, e por isso, são necessários árbitros associados em cada dispositivo para selecionar os dados a serem enviados e recebidos do barramento.

Barramentos são indicados para a construção de sistemas *multicore* com poucos núcleos de processamento. Com o desenvolvimento de processadores com dezenas a centenas de núcleos, os barramentos não atendiam mais ao paralelismo exigido, e passaram a dar lugar às **redes em *chip***.

Também conhecidas como redes de interconexão, as redes em *chip*, além de suportarem grande fluxo de dados em paralelo, proporcionam escalabilidade e portabilidade aos sistemas multiprocessados. Um exemplo de rede em chip é apresentado na Figura 5.

As redes em *chip* são constituídas por roteadores conectados entre si. Esses roteadores são responsáveis por encaminhar pacotes na rede. Cada roteador é composto por *buffers* e árbitros. Todo pacote que chega ao roteador é enfileirado em um *buffer* de entrada, onde irá aguardar ser arbitrado, isto é, o roteador definirá seu destino de saída. Uma vez definido a rota de destino do pacote, esse é enfileirado em um *buffer* de saída do roteador, onde aguardará ser enviado.

A conexão dos roteadores pode resultar em diferentes topologias, mas algumas delas são mais conhecidas, como a malha-2D, *Torus*(também chamada toróide) ou hipercubo. A topologia da rede determina quantas portas de comunicação os roteadores necessitam, e quais roteadores se comunicam diretamente.

A topologia de uma rede de interconexão é importante para a definição e seleção de uma rede em *chip* para conectar elementos de um sistema em *chip*. Ela também influencia o desempenho do sistema construído, por exemplo, uma rede em anel, onde os núcleos de chaveamento estão conectados em linha, limita a capacidade de expansão do sistema, mesmo fornecendo uma conectividade simples entre os nós.

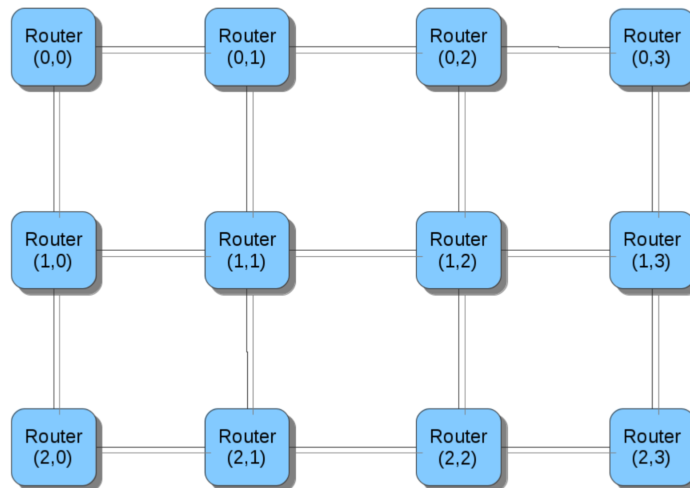

Figura 5 – Representação da rede em chip malha-2D.

O desenvolvimento de sistemas multiprocessados baseados em rede em *chip* se popularizou rapidamente. Muito devido ao paralelismo suportado e à escalabilidade que o subsistema de interconexão confere ao sistema. Ele permite que núcleos de processamento, memórias ou dispositivos de E/S sejam adicionados ou removidos do sistema sem grandes alterações na estrutura do subsistema.

2.6 Considerações Finais

A alternativa encontrada para se continuar aumentando o desempenho dos processadores, sem aumentar a potência dissipada foi limitar a frequência de operação dos elementos de processamento, porém adicionar mais elementos de processamento aos circuitos integrados. Isto é, passou-se a investir em multi-tarefas, paralelismo.

Com o crescente número de elementos de processamento nos *chips*, surgem definições de projeto, como: modelo de memória, subsistema de interconexão e subsistema de comunicação e sincronização.

A arquitetura dos sistemas multiprocessados é definida pelo modelo de memória (distribuída ou compartilhada), o subsistema de comunicação (barramentos ou redes em *chip*) e a ISA dos núcleos de processamento, dentre outros aspectos.

Com base nessas características, a programação paralela evolui de forma a gerenciar tais recursos e facilitar o uso de tais sistemas em seu máximo desempenho. Dependendo do modelo de memória, define-se um conjunto de diretrizes de programação. Sistemas com memória distribuída suportam programação baseada em troca direta de mensagens. Sistemas com memória compartilhada suportam programação baseada em compartilhamento de variáveis.

3 Estado da Arte

Este Capítulo apresenta trabalhos relacionados a esta dissertação, isto é, pesquisas a respeito dos modelos e metodologias de programação paralela, e trabalhos que apresentam alguns exemplos de *manycores*.

De posse dos conceitos relatados no capítulo anterior, são apresentados alguns trabalhos que propõem modelos de comunicação entre processos, trabalhos que avaliam e comparam diferentes modelos de programação para arquiteturas *manycore* e até mesmo, modelos de arquiteturas com configuração de memória específica.

3.1 Modelos de Programação Paralela

O desenvolvimento de aplicações em alto nível de abstração é mais cômodo para o programador, porém, um microprocessador só interpreta código binário. Portanto, para verificar o correto funcionamento da aplicação, é necessário traduzir o código de alto nível para binário. Dessa forma, o projeto de uma nova arquitetura abrange processos de desenvolvimento que vão desde a implementação da microarquitetura até ao desenvolvimento de aplicações para avaliar o funcionamento da mesma.

Com o surgimento dos *manycores*, nota-se a necessidade de se coordenar o funcionamento dos núcleos de processamento. Faz-se necessário desenvolver programas a serem executados pelos núcleos, em paralelo. Existem duas maneiras principais de se paralelizar aplicações: **autoparalelização e programação paralela**. A autoparalelização de códigos lineares pode se dar por intermédio da exploração de instruções independentes (ILP (*Instruction Level Paralelism*)), ou pela paralelização de iterações de trechos que se repetem. A programação paralela consiste no desenvolvimento de aplicações particionando a carga de trabalho, isto é, definindo tarefas a serem executadas pelos núcleos de processamento (DIAZ; MUNOZ-CARO; NINO, 2012).

Shekhar et al. (2011) relatam a dificuldade que os programadores enfrentavam para extrair o paralelismo dos programas sequenciais. A fim de facilitar a extração de paralelismo, algumas ferramentas foram desenvolvidas. Tais ferramentas, são capazes de extrair *threads* (fluxos de execução), mapear as dependências existentes entre *threads* e gerenciar tais programas em tempo de execução. Essas ferramentas possuem a vantagem de evitar erros de paralelismo e permitem que o programador se preocupe apenas com a solução do problema, ou seja, com o código.

As ferramentas de extração de paralelismo são eficientes, porém limitadas. Essas, não fornecem ao programador a liberdade para definir o nível de paralelismo (ILP ou TLP),

ou mesmo o padrão de paralelismo, isto é, a forma como as *threads* irão se comunicar. Tais características são fundamentais para o desenvolvimento de programas completamente adaptados ao hardware, pois definem como, e quando, cada núcleo do processador será utilizado, além de definir como ocorrerá a comunicação entre os núcleos, e entre núcleos e memória.

Um modelo de programação paralela possui características dependentes da arquitetura para a qual se pretende descrever aplicações, isto é, o modelo de memória da arquitetura, a forma como os núcleos estão interconectados e os protocolos e meios de comunicação implementados determinam que modelo de programação é mais adequado. Por exemplo, uma arquitetura paralela, cujos núcleos de execução estão interconectados por intermédio de um barramento e que compartilham um mesmo espaço de endereçamento, implementado em uma única memória física, adequa-se melhor à programação paralela com comunicação usando variáveis compartilhadas. Por outro lado, uma arquitetura cujos núcleos de execução possuem sua própria memória privada e se interconectam por meio de uma rede em *chip*, adequa-se melhor à programação com troca de mensagens.

Quanto à configuração de memória do processador os modelos de programação podem ser classificados em três categorias: memória distribuída, memória compartilhada e memória compartilhada distribuída.

3.1.1 Memória Distribuída

Diaz, Munoz-Caro e Nino (2012) realizam uma apresentação dos mais modernos e estáveis modelos de programação paralela desenvolvidos para as mais diferentes arquiteturas, seja com modelo de memória compartilhada, seja com memória distribuída e até mesmo os modelos híbridos.

Alguns modelos de programação paralela são destinados a arquiteturas com hierarquia de memória pura, isto é, memória compartilhada ou memória distribuída. São os chamados **modelos de programação paralela pura**.

Quanto ao desenvolvimento de aplicações para arquiteturas com modelo de memória distribuída, Pacheco (1996 apud DIAZ; MUNOZ-CARO; NINO, 2012) apresentam o ***Message Passing***.

MPI (*Message Passing Interface*) é uma biblioteca que permite o desenvolvimento de aplicações paralelas utilizando o modelo *Message Passing*. Essa biblioteca especifica chamadas de procedimentos, sub-rotinas e funções. A mesma, é compatível com as linguagens Fortran, C e C++.

Esse modelo de programação favorece padrões de paralelismo SPMD e, em menor extensão, o padrão *Master/Worker*. Nesse modelo, os processos executados em paralelo têm espaços de memória separados. A comunicação ocorre por meio da **troca direta de**

messagens.

3.1.2 Memória Compartilhada

Em processadores *manycore* com memória compartilhada, os núcleos não necessitam enviar dados diretamente de um para outro, pois possuem um espaço de endereçamento para compartilharem.

No trabalho de Shekhar et al. (2011), são analisados três modelos populares de programação paralela, OpenMP (CHAPMAN; JOST; PAS, 2007), GCD (*Grand Central Dispatch*) (APPLE, 2009) e Pthreads (BUTENHOF, 1997b), aplicando estes modelos para paralelizar dois diferentes algoritmos, Detecção de Face e Reconhecimento Automático de Voz.

O artigo compara os modelos de programação paralela, destinados a arquiteturas com memória compartilhada, quanto a gerência de memória, controle de execução das *threads* e velocidade de execução. O objetivo de utilizar dois algoritmos tão distintos é confrontar as peculiaridades de cada um. Detecção de face é um algoritmo regular, isto é, o paralelismo é o mesmo do início ao fim da execução do programa, enquanto que o Reconhecimento Automático de Voz, é um algoritmo irregular, ou seja, a quantidade de processos paralelos muda durante a execução. Dessa forma, é possível verificar a eficiência de cada modelo de programação paralela para algoritmos com paralelismo regular e irregular.

Inicialmente, Shekhar et al. (2011) apresentam o modelo de programação paralela baseado em *Pthreads*, que são definidas como um conjunto de tipos e chamadas de procedimentos da linguagem de programação C, implementados com as bibliotecas **pthread.h** e **thread.h**. Pthreads fornecem interfaces para gerenciamento de *threads*, mutexes (variáveis de exlusão mútua usadas para o gerenciemento de regiões críticas entre processos) e variáveis de compartilhamento e sincronização.

Os autores apresentam ainda o modelo de programação baseado em *OpenMP* (*Open Multi-Processing*), que é uma API que suporta programação paralela para multiplataformas de memória compartilhada. OpenMP pode ser utilizada em C, C++ e FORTRAN, em vários sistemas operacionais, incluindo UNIX e Windows. Tal API consiste de um conjunto de diretivas de compilador, biblioteca de rotinas e variáveis de ambiente que influenciam o comportamento em tempo de execução.

OpenMP inclui uma biblioteca que possui estruturas para criação de *threads*, distribuição de carga de trabalho (*work sharing*), gerenciamento de dados de ambiente, sincronização de *threads*, rotinas de tempo de execução a nível de usuário e variáveis de ambiente. A descrição de aplicações utilizando OpenMP segue o padrão de paralelismo *fork/join*, no qual um processo se subdivide em processos paralelos, e estes por sua vez,

findam unindo-se novamente.

O artigo apresenta ainda o modelo *Grand Central Dispatch (GCD)*, que é um modelo de programação paralela desenvolvido pela Apple, que suporta paralelismo baseado em *threads* e é baseado no padrão *pool* de *threads*. Nesse padrão, uma grande quantidade de *threads* é inicialmente armazenada e, dessa porção, as *threads* vão sendo removidas e entregues para execução. *GCD* é capaz de administrar a quantidade de *threads* de acordo com o número de núcleos disponíveis no processador, e as demandas por tarefas podem ser atendidas a qualquer momento da execução.

As *APIs* de programação apresentaram resultados semelhantes quanto ao desempenho. Pthreads, porém, exige uma reestruturação do código fonte para adequar funções ***thread*** para os núcleos da arquitetutura alvo. Também é necessário configurar vários mecanismos de sinalização para sincronização, o que não é necessário em OpenMP e GCD. Portanto, OpenMP e GCD facilitam a programação, porém reduzem o controle do programador sobre o paralelismo e sincronização, diferentemente de Pthreads.

O trabalho de Sanchez et al. (2012), tem o objetivo de apresentar modelos de programação paralela e avaliar os desempenhos dos mesmos para arquiteturas com memória compartilhada. Dentre os modelos existentes, o artigo tem OpenMP, TBB (*Threading Building Blocks*), Intel ArBB e CUDA, como objeto de estudo e avaliação. O objetivo de comparar os quatro modelos é avaliar o desempenho decorrente de suas peculiaridades. Por um lado, OpenMP e TBB exploram *threads* paralelas executando em sistemas *multicore*. ArBB, por sua vez, combina *threads* paralelas com características *multicore* do modelo SIMD, utilizando um modelo de programação mais simples. Enquanto que CUDA, explora características SIMD das GPUs.

O artigo apresenta duas abordagens principais a serem consideradas. Primeiro, são plataformas *multithread* de propósito geral, nas quais é possível executar várias *threads* em um único núcleo (*hyperthreading*), ou executar *threads* em núcleos diferentes de um mesmo *chip*, que compartilham um espaço de endereçamento. A segunda consiste em aplicar um mesmo código para diferentes unidades SIMD.

O trabalho apresenta e avalia algumas *APIs* de programação paralela, são elas:

- **OpenMP**: um conjunto de diretivas de compilador, chamadas de rotinas e marcações para programação multi-processo, utilizando linguagens estáveis e famosas como FORTRAN, C e C++. OpenMP é portável sobre arquiteturas de memória compartilhada, porém exige um compilador específico;

- **Intel TBB**: *Intel Threading Building Block* é um modelo de programação paralela baseado em tempo de execução para códigos C++ que usam *threads*. Consiste em uma biblioteca de tempo de execução e contém uma série de estruturas de

dados e algoritmos que facilitam a programação quando comparado com o uso de *threads* nativas. TBB representa um paralelismo de alto nível, baseado em tarefas que abstraem detalhes da plataforma e mecanismos de *thread* para desempenho e escalabilidade. Não necessita de um compilador especial. A filosofia por trás de TBB é aplicar poucas restrições e tornar possível um estilo de programação genérico;

- **Intel ArBB**: *Intel Array Building Block* (ArBB) é uma biblioteca C++ desenvolvida para exploração de problemas de dados paralelos para usufruir da vantagem de ter vários tipos de processadores. Esta biblioteca oferece um modelo de programação baseado em uma composição dinâmica de padrões estruturados de computação paralela. Ela suporta um conjunto de padrões paralelos determinísticos, por meio dos quais programas com ArBB evitam condição de corrida e *deadlocks* em suas construções. ArBB possui uma camada intermediária de paralelismo que mapeia a abstração das operações paralelas, expressas em semânticas da máquina virtual sobre a camada de hardware, melhorando assim, o desempenho. Isso faz com que ArBB seja portável e de fácil manutenção;

- **CUDA**: *Compute Unified Device Architectures* é um *framework* de computação paralela desenvolvido pela Nvidia, para GPUs. É acessível através da indústria padrão de linguagens de programação, como a linguagem C. O modelo de programação exige um conhecimento amplo da arquitetura da GPU, como por exemplo como a memória é distribuída. CUDA suporta um conjunto de interfaces computacionais incluindo *OpenGL* e *Direct Compute*. Além disso, exige um compilador específico e possui três abstrações-chaves: uma hierarquia de grupos de *threads*, memórias compartilhadas e “barreiras” de sincronização;

Mediante as avaliações sobre os modelos de programação e experimentos realizados, o trabalho conclui que ArBB é o melhor modelo de programação paralela para obter o melhor desempenho de CPU, quando as aplicações fazem uso intensivo e regular da memória. TBB e OpenMP possuem desempenhos similares em diferentes plataformas, porém OpenMP demonstra desempenho um pouco melhor na maioria dos casos.

A exploração do paralelismo a nível de *thread* tem sido um desafio para os desenvolvedores de software. Isto acontece porque, além de extraírem o máximo de desempenho do sistema embarcado, precisam reduzir o consumo de energia. Para agilizar o processo de desenvolvimento e torná-lo o mais transparente possível, foram desenvolvidas interfaces de programação paralela, do inglês *Parallel Programming Interfaces* (PPI) (LORENZON; CERA; BECK, 2015).

Lorenzon, Cera e Beck (2015) mostram que, cada PPI implementa diferentes formas de trocar dados usando regiões de memória compartilhada, influenciando, desempenho, consumo de energia e o produto energia-atraso (*Energy-Delay Product (EDP)*), os quais

variam em diferentes processadores embarcados. Por meio da avaliação de 4 PPIs e 3 processadores multicore (ARM A8, A9 e Intel Atom), os autores demonstram que simplesmente mudando a PPI é possível garantir 59% menos consumo de energia e atingir até 85% de melhorias em EDP, nos casos mais significativos. Além disso, a eficiência (o melhor uso possível dos recursos disponíveis) diminui com o aumento do número de *threads* em quase todos os casos, mas em diferentes proporções.

Para acelerar o processo de desenvolvimento e promover o maior conforto aos programadores, as interfaces de programação paralela, OpenMP, Pthreads e MPI, têm sido desenvolvidas e aprimoradas. Porém, cada uma delas possui características particulares com respeito ao gerenciamento de processos, distribuição de carga de trabalho e sincronização.

Independentemente da PPI utilizada, a comunicação entre os processos ocorre por meio da memória compartilhada. Estas regiões de memória são, comumente, mais distantes do processador (cache L3 e memória principal) e proporcionam um grande atraso e alto consumo de energia, quando comparadas à memórias mais próximas do processador como (caches L1 e L2, e registradores). Estas PPIs proporcionam diferentes formas de trocar dados, através do compartilhamento de variáveis ou primitivas de *send* e *receive*.

No cenário da programação paralela para arquiteturas com memória compartilhada, quanto mais comunicação a aplicação paralela possui, mais energia irá consumir. Por outro lado, o aumento de desempenho pode provocar redução de consumo de energia. Porém, essa melhoria não é linear, e algumas vezes não segue uma escala em relação ao número de *threads* (devido à sincronização e largura de banda do barramento de comunicação).

Lorenzon, Cera e Beck (2015) investigam qual PPI, das quatro selecionadas para avaliação (OpenMP, Pthreads, MPI-1 e MPI-2), otimiza o uso dos recursos disponíveis, considerando diferentes processadores embarcados, número de núcleos e aplicações com comportamentos distintos. Os autores consideram o número de acessos à memória e instruções executadas, desempenho, consumo de energia, EDP, escalabilidade e eficiência, e como cada um influencia os outros.

Alguns resultados quanto a desempenho e consumo de energia mostraram que para programas CPU-B, os desempenhos das PPIs são similares e muito próximos do ideal. Porém, para programas MEM-B, o modo como cada PPI realiza comunicação entre processos, bem como a hierarquia de memória de cada processador, impacta fortemente os resultados. No processador Atom, Pthreads mostraram mais uma vez ser a melhor alternativa, com grande vantagem sobre os outras PPIs.

No geral, os resultados mostram dois diferentes cenários. Pthreads é a melhor escolha para processador Atom. No caso mais significativo, é 53% mais rápido, economiza 46% de energia e melhora EDP em 75%. Por outro lado, para o processador ARM, OpenMP é a alternativa correta, sendo acima de 64% mais rápido, trazendo economias de energia

acima de 59% e melhorias de EDP acima de 85%.

Para arquiteturas com modelo de memória compartilhada Diaz, Munoz-Caro e Nino (2012) também analisam, *POSIX Threads* e OpenMP e observa que o uso de OpenMP possui um pequeno, mas contínuo aumento. Isso, devido ao uso, cada vez mais frequente, do modelo de programação paralela de memória compartilhada.

3.1.3 Memória Híbrida (Distribuída Compartilhada)

Nas atuais arquiteturas *manycore*, os *clusters* de processadores têm múltiplos nós, e cada nó é um *multicore*. Entre os nós a memória é distribuída, e portanto, os nós se comunicam por meio da troca de mensagens. Dentro de um nó, os núcleos compartilham memória, e portanto, se comunicam por meio do compartilhamento de variáveis. Dessa forma, entre os nós a programação é baseada na troca de mensagens e dentro dos nós a programação se baseia no compartilhamento de variáveis. É a chamada **programação híbrida**, que tem se tornado popular em função do desenvolvimento crescente de sistemas envolvendo GPUs.

Com o surgimento de arquiteturas com tal modelo de memória surgem os modelos **heterogêneos** de programação paralela. Diaz, Munoz-Caro e Nino (2012) apresentam, além de CUDA, algumas *APIs* adaptadas para esse modelo de programação híbrida:

- **OpenCL (*Open Computing Language*)**: é uma norma aberta para programação paralela de propósito geral para CPUs, GPUs e outros processadores. OpenCL essencialmente distingue entre os dispositivos (GPUs e CPUs) e o host (CPU). A ideia fundamental é escrever *kernels* (funções que executam em dispositivos OpenCL) e APIs para criar e destruir esses *kernels*. As diferentes execuções de um programa são enfileiradas e controladas pela aplicação *host*, que configura o contexto no qual os *kernels* executam, incluindo alocação de memória, transferência de dados entre objetos de memória e criação de filas de comandos. Esse modelo é melhor adaptado ao padrão de paralelismo SPMD, assim como CUDA;

- ***Microsoft's DirectCompute***: é uma abordagem para programação de GPUs. É parte da coleção de APIs Microsoft DirectX, adequado para GPGPUs. Tal modelo possui uma restrição: só funciona em plataformas Windows, o que limita sua usabilidade e popularidade;

- ***Intel's Array Building Blocks* (ArBB)**: fornece uma solução generalizada de programação paralela vetorial para computação matemática intensiva de dados. Como *runtime*, ArBB usa *Intel's Building Blocks*, que contribui para abstrair detalhes da plataforma e mecanismos de *threads* para fornecer escalabilidade e melhorar o desempenho. ArBB pode executar computação vetorial de dados paralelos em um

sistema possivelmente heterogêneo e é capaz também de prever condições de corrida e *deadlocks*;

- ***Partitioned Global Address Space*:** Programas paralelos de memória compartilhada são considerados mais fáceis de desenvolver que programas baseados em troca de mensagens, porém, troca de mensagens proporciona maior escalabilidade e portabilidade. Isto porque modelos de programação paralela baseado em memória compartilhada não exploram localidade de dados na cache eficientemente, enquanto que o modelo de memória compartilhada distribuída tenta combinar as vantagens de ambas abordagens, memória compartilhada e memória distribuída;

O modelo de memória híbrida combina a flexibilidade de definição do paralelismo com um balanço apropriado das cargas de trabalho, e de comunicação, para obter eficiência máxima.

3.2 Arquiteturas com suporte a programação paralela

Muitas são as arquiteturas desenvolvidas com configurações de memória distintas. Algumas delas, porém, se tornaram estereótipos bastante conhecidos e utilizados com base no desenvolvimento de computadores de alto desempenho.

O Intel Core 2 Duo, presente em *laptops* e *desktops* comerciais, é um processador *dual-core* (seu processador possui dois núcleos de execução) homogêneo, e fornece processamentos simultâneos para aplicações *multithreaded* e ambientes multi-tarefas, como descrito por Dowdeck (2006).

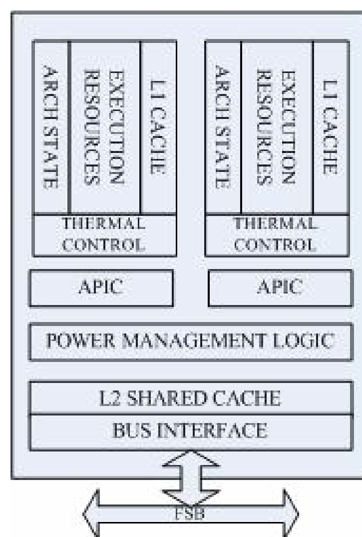

Figura 6 – Representação da configuração de um Intel Core 2 Duo.

Fonte – (SCHAUER, 2008)

A Figura 6 mostra um diagrama de blocos para o Core 2 Duo. Um processador com modelo de memória compartilhada, onde os núcleos possuem caches L1 privadas e compartilham uma cache L2 que fornece um pico de taxa de transferência de 96 GB/sec. Com tal configuração de memória, a programação para o mesmo é facilitada, pois os processos a serem executados dispõem de um espaço de endereçamento comum, e podem se comunicar através do compartilhamento de variáveis.

No Core 2 Duo, se ocorrer um *miss* na cache L1, isto é, a ausência do dado buscado, tanto a cache L2 quanto a cache L1 do segundo núcleo, são atualizadas paralelamente antes de que seja feita uma nova requisição à memória principal.

Um outro modelo de processador multicore é o Athlon 64 X2, da AMD, que também está presente em *laptops* e *desktops*, mas que não dispõem de memória compartilhada, como é possível observar na Figura 7.

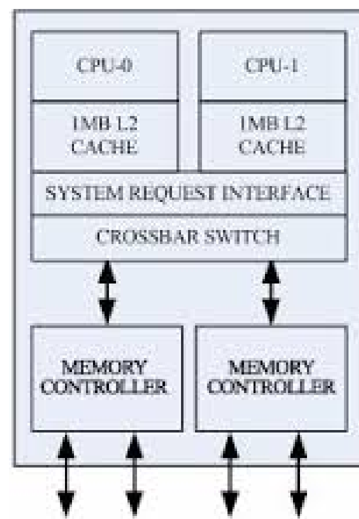

Figura 7 – Representação da configuração de um AMD Athlon 64 X2.

Fonte – (SCHAUER, 2008)

Ao contrário do Intel Core 2 Duo, o Athlon possui um modelo de memória com caches L2 privadas. Dessa forma, os processos não possuem um espaço de endereçamento comum, e portanto precisam se comunicar por meio de troca de mensagens.

Alguns outros exemplos de arquitetura com suporte à programação paralela são apresentados por Sanchez et al. (2012). O artigo destaca arquiteturas que utilizam as técnicas de paralelismo apresentadas, como *Hyperthreading* e *Multi-thread* (ÉTIENNE, 2012), e organização de memória *ccNUMA*.

Na hierarquia de memória utilizada na arquitetura Intel Nehalem (SCHELLE et al., 2010), sucessora da Intel Core 2 Duo, usa-se a mesma configuração de memória da CPU Phenom da AMD, caches L1 e L2 individuais para cada núcleo e uma cache L3 compartilhada. Cada memória cache L2 é de 256 KB e a cache L3 é de 8MB. Caches L1

permanecem como no Core 2 Duo, ou seja, 64 KB, 32 KB para instruções e 32 KB para dados.

A arquitetura Intel Nehalem, é uma CPU Intel com uma controladora de memória integrada que compõe os processadores da série Core i7, cuja representação gráfica é apresentada na Figura 8. Esta arquitetura também é utilizada nas CPUs destinadas a servidores (Xeon). CPUs baseadas nesta arquitetura, possuem um controlador de memória para dar suporte a três canais DDR3, três níveis de cache, a tecnologia *Hyperthreading* e um barramento externo chamado *QuickPath*, que permite ainda, que sockets de Nehalems suportem uma configuração ccNUMA. Portanto, esta arquitetura suporta o modelo de programação SIMD, com unidades de 128 *bits* de tamanho. A tecnologia Nehalem, suporta programação paralela baseada no compartilhamento de variáveis, pois os núcleos de processamento compartilham a memória.

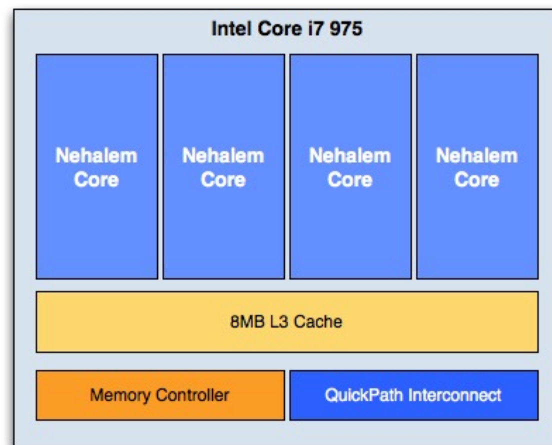

Figura 8 – Representação da configuração de memória do Intel Nehalem.

Sanchez et al. (2012) apresentam também a arquitetura Intel Sandy Bridge, que assim como a Nehalem, também permite a inclusão de até 8 núcleos físicos por *chip* e acrescenta a capacidade para *Hyperthreading*. Além disso, vários *chips* podem ser conectados em uma plataforma ccNUMA. A grande mudança no paralelismo é a inclusão de unidades de SIMD de 256 bits e a inclusão de um conjunto vetorial de instruções.

Sandy Bridge trás um conjunto vetorial de instruções AVX (*Advanced Vector Extensions*). Estas instruções usam o mesmo conceito de paralelismo SIMD. Este conceito consiste em usar um único e grande registrador para armazenar muitos dados e então processar todos eles com uma única instrução, acelerando o processamento.

Em relação ao modelo Nehalem, Sandy Bridge apresenta uma nova organização de memória, com uma cache L0, para armazenar micro-instruções recém decodificadas. A programação paralela, porém, permanece sendo uma programação baseada no compartilhamento de variáveis, pois da cache L1 em diante, na hierarquia de memória, utiliza

espaço de endereçamento compartilhado.

A microarquitetura Sandy Bridge possui uma estrutura em anel para os componentes internos se comunicarem entre si. Quando um componente necessita se comunicar com outro, ele coloca a informação no anel que moverá esta informação até a mesma alcançar seu destino. Nenhum componente nenhum conversa com outro diretamente, eles devem utilizar a estrutura em anel. Os componentes que usam o anel são: os núcleos de processamento, cada memória cache, e agentes de sistema adicionados na construção de processadores e para comunicações externas como, controlador de memória, controladora PCI Express, unidade de controle de energia e *display*, e controladora gráfica.

Cada unidade do último nível da cache não está associada a um núcleo de processamento. Qualquer núcleo pode usar qualquer uma das caches. Por exemplo, na Figura 9, tem-se quatro núcleos de processamento com quatro unidades do último nível de cache. O núcleo 1 não possui a cache 1 como privada; ele pode usar qualquer uma das caches.

Na Figura 9 podemos ver o anel, representado pelas linhas tracejadas que conectam todos os núcleos de processamento entre si.

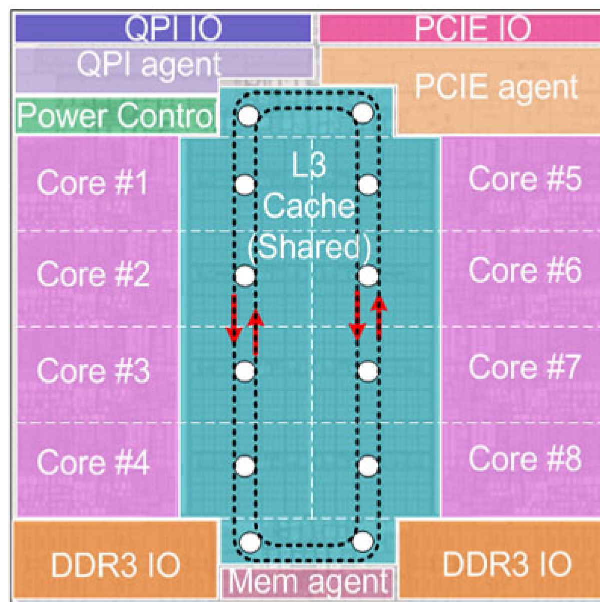

Figura 9 – Representação de um processador baseado em Sandy Bridge.

Há ainda a GPU Nvidia GF100, que é basicamente um array de multiprocessadores projetados para executar as mesmas operações com diferentes conjuntos de dados, SIMD (*Single Instruction, Multiple Data*). A arquitetura, inicialmente com 4 núcleos, pode ser aprimorada para 512 nós CUDA distribuídos, com 16 multiprocessadores de fluxo de dados, (*SM - Streaming Multiprocessors*), com 32 núcleos cada, 16 unidades de ponto flutuante e 4 unidades de função especial, trabalhando em uma frequência de 1.25 a 1.4 GHz. Cada SM possui 16 + 48 KB de memória compartilhada (cache L1), e a memória global varia de 3 a 6 GB.

Portanto, na Nvidia GF100, cada SM é composto por vários processadores *multicore*, que se comunicam por meio de troca de mensagens, enquanto que, a comunicação entre SMs ocorre por compartilhamento de variáveis na memória global. Dessa forma, tal arquitetura suporta programação híbrida, aproveitando a escalabilidade de uma configuração com memória distribuída e a facilidade de programação conferida pelo compartilhamento de memória.

Além dos processadores comerciais apresentados, alguns outros modelos, acadêmicos, também merecem destaque.

Kawakami et al. (1985) apresentam um microprocessador de *chip* único para reconhecimento de voz, o SRP (em inglês, *Speech Recognition Processor*). O SRP é uma arquitetura de múltiplos processadores e possui estrutura *pipeline*. Ele consegue reconhecer até 340 palavras isoladas, ou 40 palavras conectadas, em tempo real. Para o SRP, variações de algoritmos e parâmetros de operação são programáveis pelo usuário.

A Figura 10 apresenta um típico sistema de reconhecimento de voz usando o processador SRP, o qual consiste em um microprocessador padrão (*host CPU*), uma memória principal, um analisador de voz, uma memória de padrão de voz e o SRP. Para agilizar as longas tarefas da *host CPU*, o SRP se comunica diretamente com a memória padrão, que armazena padrões de referência. O tamanho do vocabulário pode ser facilmente expandido por meio do uso de múltiplos SRPs em paralelo, como representado pelas linhas pontilhadas na Figura 10

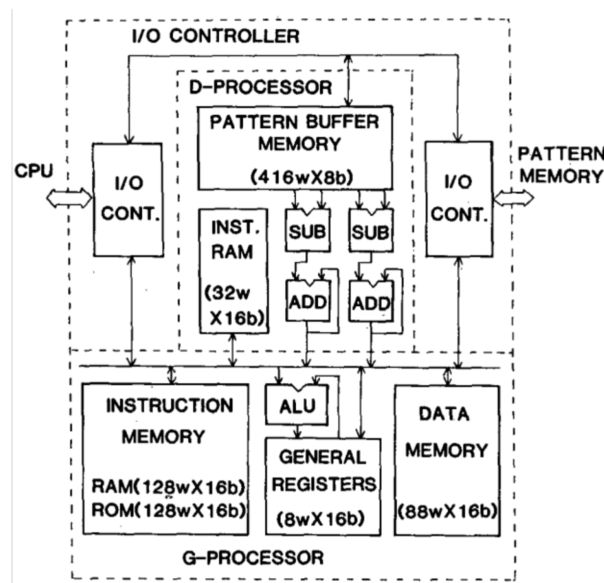

Figura 10 – Representação em diagrama de blocos de um sistema de processamento utilizando o processador SRP.

Fonte – (KAWAKAMI et al., 1985)

Quando conectados em paralelo, múltiplos *SRPs* se comunicam através da troca direta de mensagens, necessária para atualizar dados a serem atualizados na memória

padrão e nas memórias locais de cada *SRP*.

Além de multiprocessadores voltados para aplicações específicas, alguns modelos de plataforma para construção de MPSoCs de próposito geral são conhecidos. O trabalho desenvolvido por Rego (2006) apresenta uma plataforma MPSoC, chamada STORM (*MPSoC Directory-Based Platform*). Uma plataforma composta por: processador SPARC V8; processador GPOP, desenvolvido pelo próprio autor; módulo de cache, móculo de memória, módulo de diretório e dois diferentes modelos de rede em *chip*, a NoCX4 e a Árvore Obesa. Uma representação gráfica de STORM pode ser vista na Figura 11.

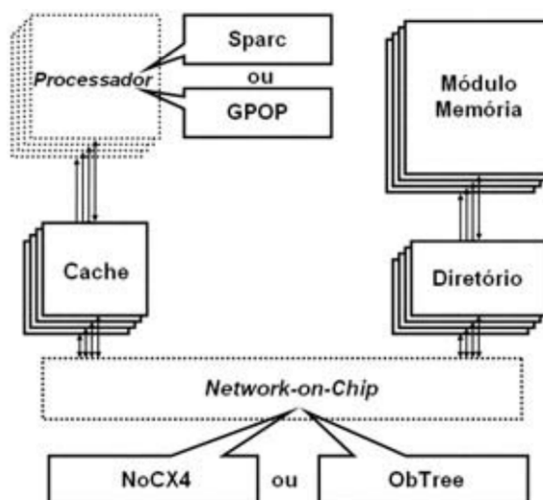

Figura 11 – Representação em diagrama de blocos da plataforma STORM.

Fonte – (REGO, 2006)

Para a programação da plataforma usando a linguagem C foi implementado um montador SPARC, totalmente compatível com o código *assembly* gerado pelo compilador *GCC*. O ambiente de programação para STORM ainda conta com uma biblioteca de funções para gerenciamento de *mutexes*.

Em STORM, as unidades de processamento possuem duas caches privadas, uma de instruções e uma de dados. STORM faz uso de memória compartilhada, onde diversos blocos de memória espalhados pela NoC formam um único espaço de endereçamento, visível por todos os processadores. Apesar do uso de memória compartilhada, a plataforma faz uso de diretório para a manutenção da coerência da cache, o que elimina o gargalo dos protocolos *snoop*: o *broadcast*.

3.3 Modelo de Programação IPNoSys

IPNoSys é um acrônimo para *Integrated Processing NoC System*. IPNoSys é uma arquitetura paralela não-convencional e modelo de programação baseado em características

de rede em *chip*, onde os roteadores, além da função de rotear, podem também executar instruções lógico-aritméticas (ARAUJO, 2012).

Estes roteadores, denominados *Routing and Processing Unit (RPU)*, estão conectados por intermédio de uma topologia em malha-2D e são responsáveis pela execução das aplicações. Além da malha de RPUs, a arquitetura IPNoSys inclui quatro *Memory Access Units (MAU)*, distribuídas nos cantos da rede em *chip* (*NoC - Network-on-Chip*). As MAUs são responsáveis pelo gerenciamento dos acessos à memória e por injetar as aplicações na rede de RPUs.

Uma das MAUs é responsável por entradas e saídas e, portanto, é chamada de IOMAU. A Figura 12 mostra a Arquitetura IPNoSys (ARAUJO, 2012).

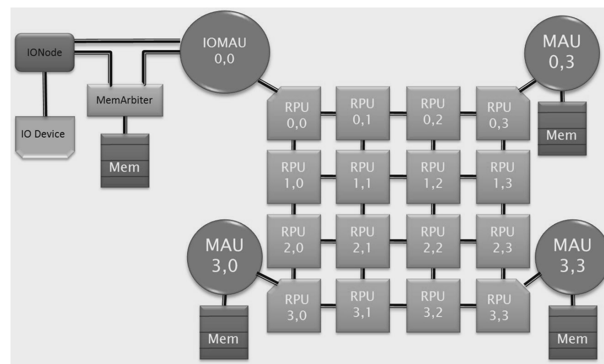

Figura 12 – Arquitetura IPNoSys.

Fonte – (ARAUJO, 2012)

Na arquitetura IPNoSys, as aplicações são compiladas como um conjunto de pacotes. Cada pacote representa um processo de aplicação que deve ser executado. Cada pacote é identificado por um par identificador (Program_ID, Packet_ID) incluído no seu cabeçalho.

A ISA (*Instruction Set Architecture*) de IPNoSys inclui 4 instruções de comunicação e sincronização entre processos (SEND, EXEC, SYNEXEC e SYNC), as quais são executadas somente por MAUs. Portanto, quando RPUs decodificam estas instruções, elas enviam, através de um pacote de controle, para uma MAU alvo. As operações realizadas por tais instruções são:

- SEND: esta instrução é usada para enviar e armazenar dados gerados pelo pacote atualmente sendo executado em um pacote esperando execução;
- EXEC: ordena uma execução assíncrona de um pacote. O pacote deve ser injetado na malha de RPUs tão logo seja possível, pela MAU recebendo a instrução EXEC;
- SYNEXEC: ordena uma execução síncrona de um pacote. A execução do pacote depende de sinais de sincronização que serão enviados por um conjunto de pacotes especificados na instrução SYNEXEC;

- SYNC: sinaliza que um pacote alcançou um ponto de sincronização. O evento de sincronização é enviado para a MAU que está esperando eventos de sincronização para injetar um pacote para execução síncrona.

Com estas instruções é possível desenvolver aplicações onde o programador indica, explicitamente, o paralelismo. A Figura 13 mostra uma aplicação desenvolvida segundo o modelo de programação IPNoSys. A aplicação apresentada na Figura 13 é o processo de decodificação do algoritmo *Run Length Encoding Algorithm (RLE).*

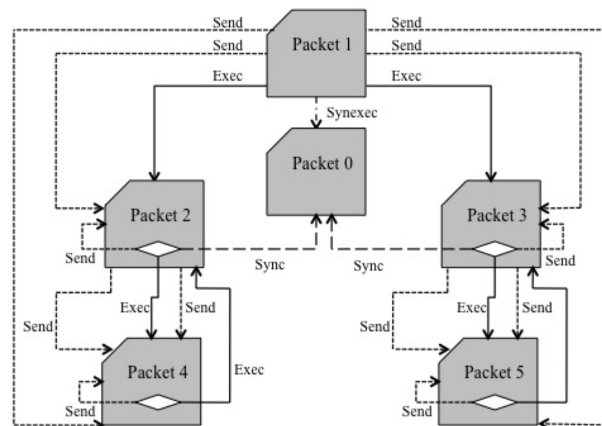

Figura 13 – Representação esquemática de uma aplicação para IPNoSys.

Fonte – (ARAUJO, 2012)

Seis pacotes compõem esta aplicação em sua versão paralela segundo o modelo IPNoSys. O pacote 1 inicializa a aplicação, enviando para os pacotes 2 e 3 o endereço inicial de onde os dados serão lidos para decodificação (instruções *SEND* saindo do pacote 1 para os pacotes 2 e 3). O pacote 1 também é responsável por iniciar a execução paralela. Neste caso dois fluxos de execução paralela são iniciados, por intermédios de duas instruções *EXEC* que ordenam a execução assíncrona dos pacotes 2 e 3. O pacote 1 também envia para os pacotes 4 e 5 o endereço inicial onde os dados decodificados devem ser armazenados (instruções *SEND* saindo do pacote 1 para os pacotes 4 e 5).

A decodificação RLE em si é realizada por um par de pacotes, pacotes 2 e 3, para o primeiro fluxo paralelo, e pacotes 4 e 5 para o segundo fluxo. Os pacotes 2 e 3 buscam os dados codificados e os enviam aos pacotes 4 e 5, respectivamente. Os pacotes 4 e 5 decodificam os dados e armazenam o resultado da decodificação.

A execução consecutiva dos pares de pacotes (2 e 3) e (4 e 5) se repete enquanto houverem dados a serem buscados e decodificados. Assim, estes pacotes enviam para si mesmos um controle de iterações (instruções *SEND* saindo destes pacotes para eles mesmo).

No modelo IPNoSys, o pacote 1 é sempre o iniciador da aplicação, e o pacote 0 é sempre o finalizador. Deste modo, ao iniciar a aplicação, o pacote 1 emite uma ordem de

execução síncrona do pacote 0 (instrução *SYNEXEC* saindo do pacote 1 para o pacote 0). O pacote zero só é executado, entretanto, quando os pacotes 2 e 3, identificarem que não há mais dados a serem decodificados, e enviarem sinais de sincronização (instruções *SYNC*, saindo dos pacotes 2 e 3 para o pacote 0). Ao enviar a ordem de execução síncrona, o pacote 1 especifica que são necessários dois sinais de sincronização, o pacote 0 necessita, portanto, que os dois sinais de sincronização cheguem para que possa ser executado.

A primeira versão da arquitetura IPNoSys é capaz de executar quatro fluxos paralelos. A aplicação RLE para execução paralela com quatro fluxos, bem como outras aplicações, podem ser encontradas em (ARAUJO, 2012). Outras versões da plataforma IPNoSys (SOARES; SILVA; FERNANDES, 2015)(PEREIRA, 2014) suporta mais fluxos de execução paralela.

Portanto, toda a comunicação ocorre através da memória, todo pacote é carregado e gravado em memória, através das MAUs. As RPUs não possuem registradores, todo dado a ser armazenado deve ser enviado para uma MAU.

IPNoSys é uma plataforma não-convencional que possui modelo de memória distribuída, uma em cada canto da rede, às quais somente as MAUs têm acesso. Dessa forma, a mesma confere suporte a programação paralela baseada na **troca de mensagens** entre os processos na rede.

3.4 Considerações Finais

Na última década, MPI foi a ferramenta mais utilizada para programação paralela, isto é, esse modelo de programação paralela é o mais popular (DIAZ; MUNOZ-CARO; NINO, 2012).

O uso de OpenMP possui um pequeno, mas contínuo aumento. Isto, em consequência do atual disseminado uso de sistemas *manycore* com modelo de programação paralela baseado em memória compartilhada.

Muitos dos atuais sistemas multiprocessados têm adotado o modelo de programação paralela de memória compartilhada, pois é um modelo que permite a paralelização automática dos processos. Isto é, o programador não precisa se preocupar com a forma através da qual os processos vão se comunicar ou com a granularidade do paralelismo, deve apenas indicar ao compilador que dados os processos transferem.

IPNoSys é uma plataforma não-convencional, porém, possui um modelo de comunicação entre processos eficiente, que permite que a plataforma confira suporte para a programação paralela.

À vista da eficiência do subconjunto de instruções de comunicação e sincronização de *IPNoSys*, há a possibilidade de extrair o desempenho máximo do mesmo em uma

arquitetura convencional.

Nesta Dissertacão de mestrado, decidiu-se trabalhar com o modelo de programação IPNoSys em uma arquitetura *manycore*, convencional, capaz de suportar um número bastante elevado de fluxos paralelos de execução. Como resultado deste trabalho, foi projetada, descrita e validada por simulação, a plataforma *ArachNoC*. No próximo Capítulo esta plataforma será apresentada em detalhes.

4 A Plataforma *ArachNoc*

ArachNoc é um processador *manycore*, composto por nós *multicore* de processamento, isto é, cada nó de processamento é composto por vários núcleos. *ArachNoc* surgiu a partir de um processador *multicore* chamado *ArachNid*, também desenvolvido pelo autor dessa Dissertação.

ArachNid é composto por um núcleo mestre e oito núcleos escravos, conectados por meio de barramento. Tal processador foi implementado em linguagem VHDL(*Verilog Hardware Description Language*) e SystemC. Esse modelo será melhor explicado na próxima seção.

4.1 O Processador *Multicore ArachNid*

ArachNid é um processador *multicore*, composto por nove núcleos, um núcleo mestre e oito núcleos escravos, e possui duas memórias compartilhadas, uma de instruções e uma de dados, ambas com capacidade de 512 KB. Na Figura 14 há a representação gráfica de *ArachNid*.

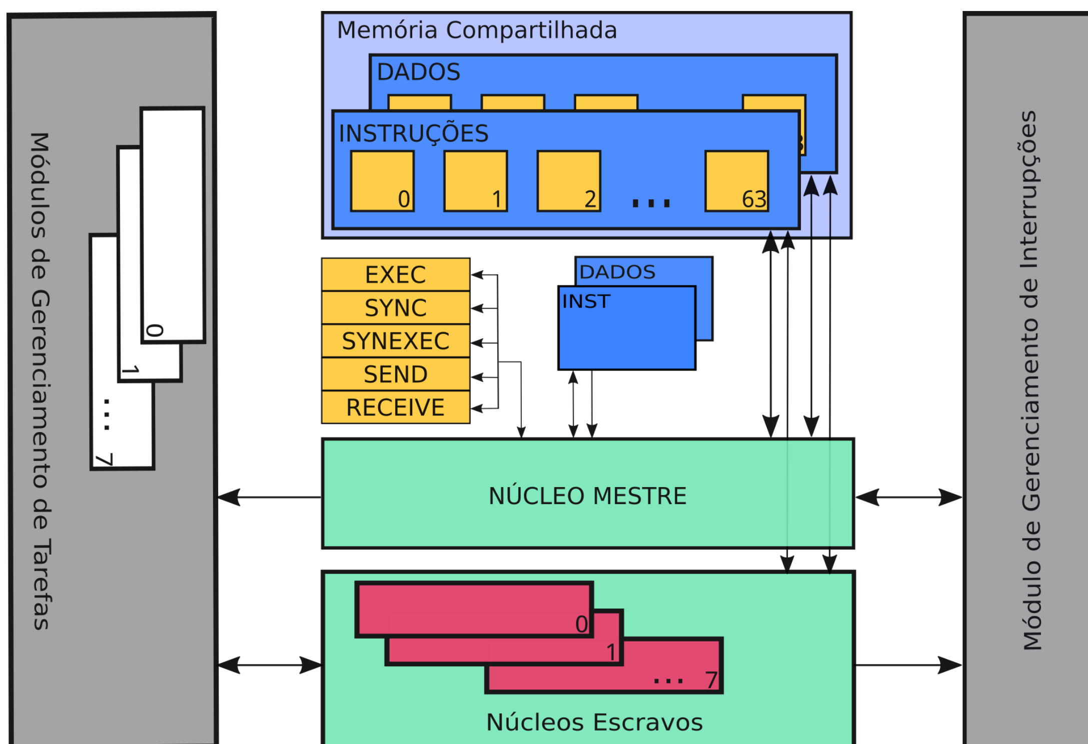

Figura 14 – Representação gráfica de *ArachNid*.

O núcleo mestre tem a função de tratar interrupções, provenientes dos núcleos escravos, além de enviar e executar tarefas. As interrupções tratadas pelo núcleo mestre são

interrupções de software (*traps*), necessárias para a importação do modelo de programação *IPNoSys*. Na Figura 14, os módulos Exec, Sync, Synexec, Send e Receive, são blocos de memória dedicados ao armazenamento do código das rotinas de tratamento dessas *traps*. Os núcleos escravos, por sua vez, apenas executam as tarefas enviadas pelo mestre.

Os núcleos de processamento são baseados na arquitetura MIPS32, descrita em Hennessy e Patterson (2011), com algumas alterações. As alterações foram realizadas para prover suporte, nativo em *hardware*, à implantação do modelo de programação *IPNoSys*, principalmente no que diz respeito à necessidade de instruções de comunicação e sincronização entre processos.

A memória é constituída por 64 blocos com duas portas de acesso e 8KB, cada bloco. Dessa forma, cada bloco pode ser acessado por dois processadores ao mesmo tempo, sendo uma porta reservada para o núcleo mestre, enquanto a outra porta é disputada pelos núcleos escravos. Portanto, apenas um escravo consegue acessar um bloco de memória por vez.

Uma porta de acesso exclusivo do núcleo mestre, foi implementada devido a necessidade de execução de tarefas críticas, como o tratamento de interrupções. Esta, e outras tarefas do núcleo mestre, não podem atrasar por causa de eventuais preempções, sofridas na tentativa de acesso à memória. Dessa forma, o núcleo mestre sempre conseguirá acessar a memória.

ArachNid não possui uma rede de interconexão multiponto, portanto, os núcleos de processamento são conectados por meio de barramento, ponto-a-ponto. A comunicação entre os processos em execução ocorre por meio do compartilhamento de variáveis. Porém, a comunicação entre núcleos escravos e núcleo mestre para sinalização de *traps*, por exemplo, ocorre por meio de barramento dedicado.

A comunicação e sincronização entre processos, baseia-se na troca de informações por meio de instruções do modelo de programação IPNoSys. A execução dessas instruções gera uma *trap* que é tratada pelo núcleo mestre. As instruções de comunicação e sincronização são: EXEC, SYNC, SYNEXEC, SEND e RECEIVE. Onde:

- EXEC: ordem de execução assíncrona de uma *thread* tão logo seja possível;
- SYNC: sinal de sincronização enviado de uma *thread* para outra;
- SYNEXEC: ordem de execução síncrona de uma *thread*;
- SEND: envio de dados de uma *thread* para outra;
- RECEIVE: instrução bloqueante, por meio da qual o núcleo coleta os dados aguardados pela *thread*;

Quando essas instruções são executadas, são gerados sinais de interrupção que são enviados para o módulo de gerenciamento de interrupções (MGI). Esses sinais são armazenados pelo MGI, em *buffer*, na mesma ordem em que chegam. De posse desses sinais, o MGI retira o primeiro sinal da fila, decodifica-o para identificar o tipo (*EXEC, SYNC, SYNEXEC, SEND* ou *RECEIVE*) e sinaliza o núcleo mestre que há *trap* a ser tratada. O núcleo mestre, por sua vez, salta para a rotina de tratamento.

Ao fim do tratamento das *traps EXEC* e *SYNEXEC*, que determinam a execução de uma *thread*, o mestre possui o endereço inicial da *thread* a ser executada, e o envia para um dos módulos de gerenciamento de tarefas (MGT), selecionado por meio de um *round-robin*. Esse processo de seleção é explicado com mais detalhes na subseção 4.2.1.2. O MGT, por sua vez, enfileira os endereços recebidos, e os envia para execução em um núcleo escravo.

Os módulos de gerenciamento de tarefas e de interrupções são de fundamental importância para *ArachNid*. Sem esses módulos, os núcleos não poderiam receber tarefas enquanto estivessem executando alguma *thread*. Eles deveriam ser interrompidos sempre que chegasse uma nova tarefa. Portanto, cada núcleo possui um MGT para armazenar e serializar a execução das tarefas recebidas.

Para a construção de *ArachNoc*, o *ArachNid* foi adaptado para comunicação em rede. Para isso, foi adicionado uma interface de comunicação com os roteadores da rede.

Em *ArachNoc* optou-se por definir um nó da rede como o gerenciador, mestre, do sistema como um todo. Um nó responsável por tratar interrupções, gerenciar as informações de sincronização, e consequentemente, designar as tarefas a serem executadas pelos nós escravos.

4.2 O *manycore* ArachNoc

Em *ArachNoc* há 8 nós escravos e 1 nó mestre, conectados por meio da rede em *chip* SoCiN, (ZEFERINO; SUSIN, 2003).

ArachNoc implementa uma hierarquia de memória baseada em memória local, nos nós de processamento, fisicamente distribuída e logicamente compartilhada. Isto é, cada nó de processamento possui um espaço de endereçamento local composto por múltiplos blocos de memória, porém, todos os blocos de memória são compartilhados por núcleos internos ao nó.

ArachNoc implanta, em uma arquitetura convencional, o modelo de programação IP-NoSys, um modelo originalmente proposto para uma arquitetura paralela não-convencional. Entende-se por arquitetura convencional, uma arquitetura cujos nós de processamento se baseiam em processadores convencionais, isto é, que seguem o modelo de Von Newman.

Para conectar os nós de processamento é utilizada uma rede em *chip* em formato toróide, que é regida pelo protocolo *wormhole*, para o encaminhamento de mensagens. A chamada rede SoCiN (ZEFERINO; SUSIN, 2003). A rede SoCiN não foi desenvolvida no escopo deste trabalho.

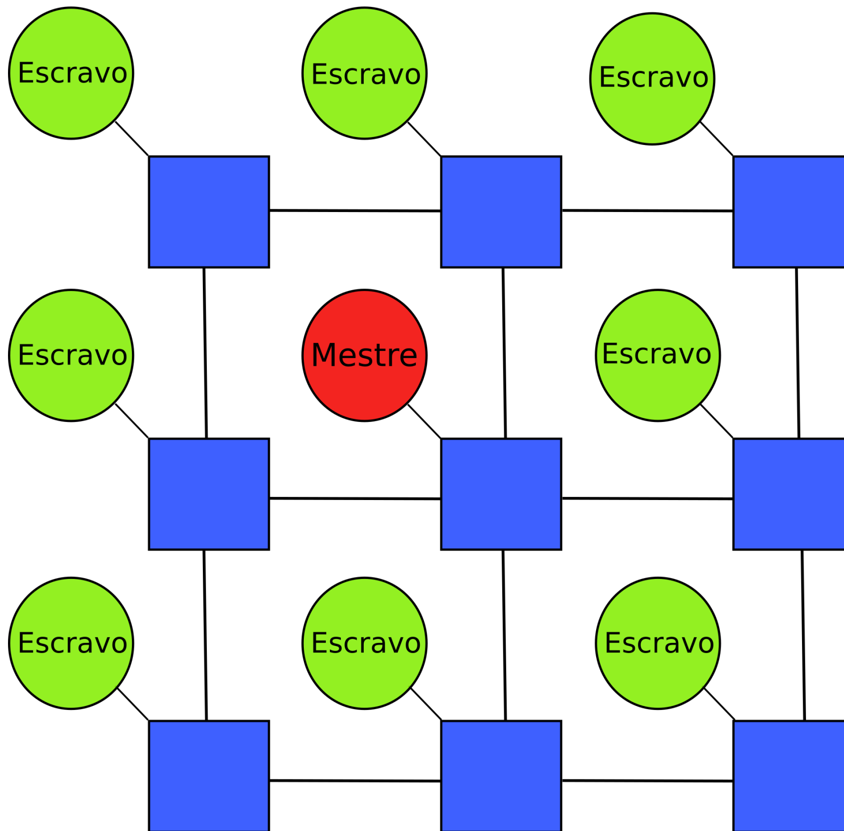

Figura 15 – Representação gráfica geral de ArachNoc.

Uma representação gráfica de *ArahcNoc* pode ser vista na Figura 15. Onde os elementos circulares representam os nó de processamento, os periféricos são os nós escravos e o central, o nó mestre. Os quadrados representam os roteadores da rede em *chip*.

Os nós de *ArachNoc* são construídos com base na arquitetura de *ArachNid*. Os componentes são separados de modo a compor os nós mestre e escravos de *ArachNoc*, com a conservação da estrutura de memória.

4.2.1 Nós de Processamento

Os nós de processamento de *ArachNoc* são provenientes de *ArachNid*. Esse foi adaptado para ser integrado a uma rede em *chip*, isto é, para se comunicar com roteadores da rede SoCiN.

O nó mestre de *ArachNoc* é constituído por um único núcleo, semelhante ao núcleo mestre de *ArachNid*. Este núcleo está conectado à duas memórias de propósito geral, uma de instruções e uma de dados, e outras cinco memórias dedicadas ao tratamento

de interrupções. Esse núcleo de processamento também se conecta a um módulo de gerenciamento de tarefas e interrupções, que por sua vez se comunica com a interface de rede.

O nó escravo é um *multicore* constituído de duas memórias, uma de instruções e uma de dados; um módulo de gerenciamento de tarefas e interrupções, que se conecta ao roteadores da rede e; oito núcleos de processamento, semelhantes aos núcleos escravos de *ArachNid*.

Os nós de processamento de *ArachNoc* são melhor explicados na secções a seguir.

4.2.1.1 Nó Escravo

Os nós escravos possuem processadores *MIPS* de 32 bits, com algumas modificações que possibilitam a implementação das instruções de sincronização e comunicação, oriundas do modelo de programação IPNoSys, e possibilitam a criação de uma hierarquia entre os nós do sistema. As adaptações para implementação das instruções de sincronização e comunicação implicam em suporte ao tratamento de exceções e interrupções. As adaptações para a criação da hierarquia de nós de processamento implicam em mecanismos de comunicação entre os nós, isto é, instruções que permitam a troca de dados entre os nós de processamento e instruções para enviar dados de interrupções dos nós escravos para o nó mestre.

No tocante às arquiteturas *manycore*, um ponto deve ser analisado com cuidado, qual seja: a distribuição de carga de trabalho. O desempenho dos nós depende do desempenho dos seus núcleos. Para obter-se alto desempenho é necessário fazer todos os núcleos funcionarem, isto é, deve-se ter uma eficiente distribuição de tarefas para os núcleos escravos. Para isso os nós contam com o módulo de gerenciamento de tarefas e interrupções (MGTI), como mostrado na Figura 16.

Na hierarquia dos nós de processamento, os núcleos de processamento do nó escravo têm a função de executar tarefas. Se, durante a execução da *thread*, um núcleo encontrar uma instrução de sincronização ou comunicação, o mesmo envia uma *trap* para o MGTI. Posteriormente, a interrupção é enviada, juntamente com os dados necessários ao tratamento da mesma, para o nó mestre. O nó mestre é responsável pelo tratamento de interrupções e pelo envio de tarefas para os nós escravos.

A comunicação entre os nós escravos e o nó mestre pode ocorrer por meio da troca direta de mensagens. A comunicação entre processos em execução, dentro de um nó escravo, ocorre por meio do compartilhamento de variáveis.

Os núcleos de processamento dos nós escravos não têm capacidade de armazenar as *threads* enviadas pelo nó mestre. Em razão disso, cada um deles tem um MGTI, que é um *hardware* capaz de guardar as informações das *threads* a serem executadas, e selecionar

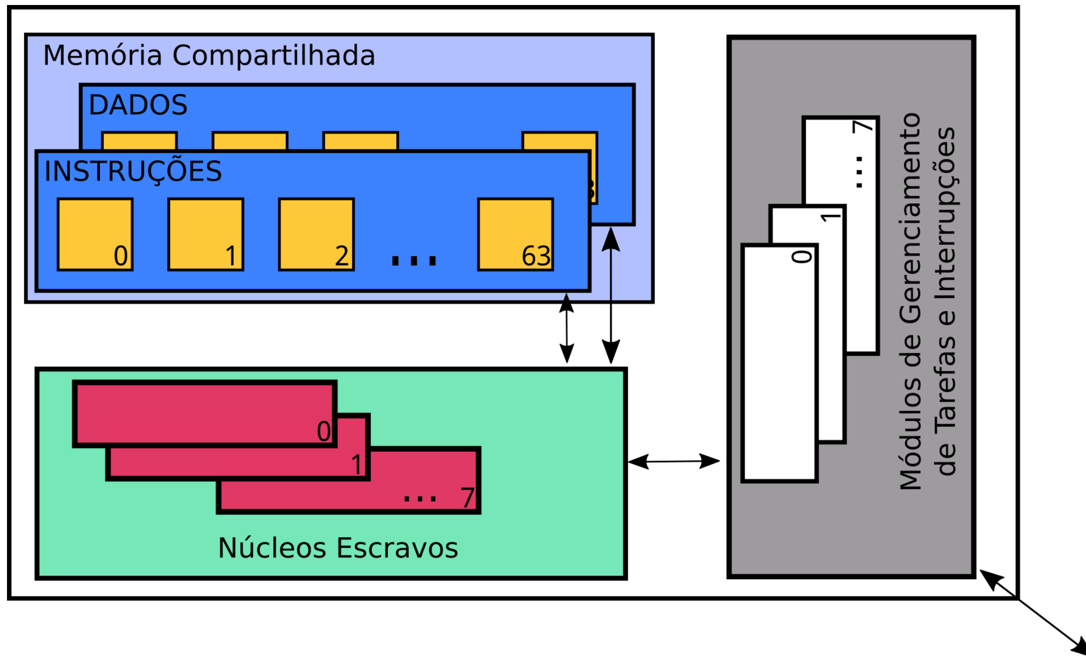

Figura 16 – Representação gráfica de um nó escravo de *ArachNoc*.

uma por vez para ser processada.

Na distribuição de tarefas, realizada pelo MGTI, a seleção de um núcleo para receber tarefa é baseada no algoritmo *round-robin*, pois é um algoritmo de simples implementação que garante um revezamento entre os núcleos escravos, e portanto, evita o sobrecarregamento de um deles. Dentro do MGTI, existem pares de *buffers* associados a cada núcleo de processamento. Um *buffer* de sinais de interrupção e um *buffer* de tarefas. Cada *buffer* de tarefa informa para o MGTI a quantidade de tarefas que possui e, no caso do *buffer* estar vazio e o núcleo de processamento ocioso, o próprio *buffer* sinaliza que está vazio e que o núcleo de processamento correspondente está ocioso, passando o mesmo a ter maior prioridade durante a distribuição de tarefas.

No processo de distribuição de tarefas, os *buffers* com menor quantidade de tarefas possuem maior prioridade para receber novas tarefas. Desse modo, o sistema mantém um equilíbrio de carga de trabalho e garante que a maioria dos núcleos de processamento permaneçam em funcionamento a maior parte do tempo, até que a aplicação seja finalizada. Esse método de seleção de núcleos para envio de tarefas objetiva manter o sistema como um todo em equilíbrio.

Com o acoplamento de MGTIs aos núcleos de processamento, não é necessário parar processamento algum para receber ou enviar *threads*. Isso evita um "gargalo" na comunicação mestre-escravo, pois permite que cada núcleo de processamento, do nó escravo, possua uma fila de tarefas para execução que, consequentemente, descongestiona o *buffer* de saída do nó mestre e reduz o tráfego de *threads* na rede em *chip*.

Portanto, o MGTI pode acumular interrupções oriundas do núcleo de processamento

a ele conectado, e enviá-las para o nó mestre. Pode também, acumular tarefas enviadas pelo nó mestre, e enviar uma nova tarefa para ser executada, sempre que o núcleo de processamento conectado a este, sinalizar que está “livre” (sem tarefa para executar).

4.2.1.2 Nó Mestre

Na hierarquia mestre-escravo, enquanto os nós escravos recebem ordens de execução, o nó mestre, Figura 17, é responsável por coordenar interrupções e tarefas (*threads*). Cabe ao nó mestre tratar interrupções de comunicação e sincronização (*EXEC, SYNC, SYNEXEC, SEND, RECEIVE*) e gerenciar as tarefas a serem executadas pelos nós escravos.

Para a implementação do nó mestre, algumas modificações foram necessárias: registradores para tratamento de exceções e interrupções, e componentes de *hardware* para suporte à comunicação entre os nós e núcleos da arquitetura *manycore*.

Os núcleos de processamento dos nós escravos podem, durante a execução de uma aplicação, gerar uma quantidade significativa de interrupções. Assim, é possível que os núcleos de processamento gerem interrupções a uma taxa maior do que a velocidade de tratamento destas. No nó mestre, o possível acúmulo de sinalizações de interrupções, é gerenciado, pelo MGTI, por meio de um *buffer* que armazena todas as interrupções enviadas, e as sinaliza para o processador do nó mestre. Esse, por sua vez, executa a rotina específica de tratamento de interrupções, tão logo seja possível.

Quando o nó mestre trata interrupções, a rotina pode retornar endereços de tarefas a serem executadas - como no caso da execução das instruções *EXEC*, *SYNEXEC* e *SYNC*-, gerando tarefas que posteriormente serão enviadas para o MGTI. O MGTI do nó mestre armazena as tarefas enviadas pelo processador do nó mestre, em um *buffer*. Se existir tarefa no *buffer*, o MGTI envia a primeira tarefa da fila para o nó escravo. O MGTI do nó escravo, a receberá, armazenará e, tão logo seja possível, escolherá um núcleo de processamento para executá-la.

No padrão MIPS, para o tratamento de interrupções, há um registrador para guardar o endereço de memória inicial da rotina de tratamento. Logo que chega uma interrupção, o processador salta para o endereço previamente guardado. Este procedimento também é utilizado para o tratamento de exceções. Dessa forma, para tratar interrupções de **Exec, Sync, SynExec, Send e Receive**, ao processador do nó mestre foram conectadas cinco memórias, cada uma dedicada ao armazenamento de uma rotina de tratamento.

Portanto, quando o nó mestre recebe uma *trap* **Exec**, o mesmo salva o contexto da *thread* em execução e desvia o fluxo de execução para a memória com rotina de tratamento de *trap Exec*. Procedimento similar ocorre com os demais tipos de interrupções, com uma memória para cada tipo.

Por fim, o nó mestre possui uma memória local, onde devem ser guardados os

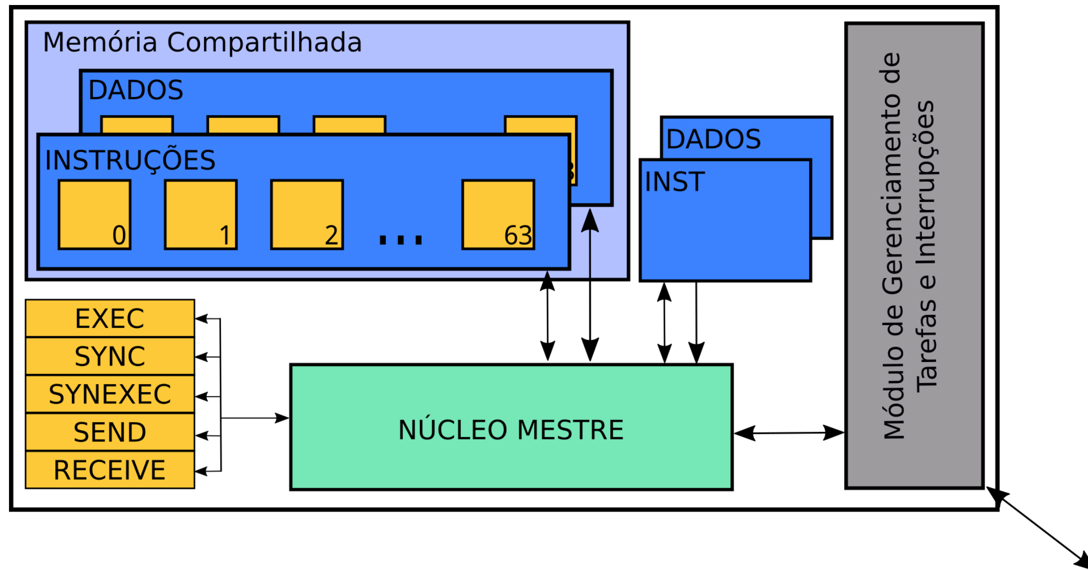

Figura 17 – Representação gráfica de um nó mestre de *ArachNoc*.

códigos iniciais das aplicações, a partir dos quais tais aplicações bifurcam em *threads* paralelas, seguindo o padrão *Fork-Join*.

4.3 Hierarquia de Memória

Uma memória compartilhada entre os nós de processamento do sistema, exige um maior tempo de acesso e exige que os dados sejam copiados desta para as memória privadas de cada nó do sistema.

Portanto, em *ArachNoc* cada nó de processamento possui uma memória privada, que é compartilhada pelos núcleos do nó. Porém, não há uma memória compartilhada entre os nós da rede, apesar de os mesmos poderem se comunicar por meio da troca de mensagens.

4.3.1 Memória Compartilhada

As memórias de instruções e de dados de um nó, são espaços de endereçamento de 512 KB constituídos de 64 blocos físicos de 8 KB cada. O bloco é uma memória com uma única porta de acesso, pois em *ArachNoc* o nó mestre possui um único processador e, esse, não disputa o acesso à memória com nenhum outro.

Quando um núcleo de processamento consegue acessar um bloco de memória, nenhum outro núcleo consegue acessá-lo por essa mesma porta. Dessa forma, em arquiteturas *multicore* com memória compartilhada, em que se tem vários núcleos competindo pelo acesso à memória, em um espaço de endereçamento contíguo de 512 KB, há alta probabilidade de ocorrerem muitas preempções. Uma memória compartilhada, com apenas uma porta e grande capacidade de armazenamento, uma vez acessada por um núcleo de

processamento, impossibilita que os demais endereços sejam acessados pelos outros núcleos do nó. Dessa forma, ocorre o bloqueio de mais endereços de memória que os que estão sendo acessados, isto é, subutilização de memória.

Portanto, a memória local foi implementada com 64 espaços menores de endereçamento, isto é, a probabilidade de dois ou mais núcleos disputarem a única porta de acesso é menor, reduzido ao tamanho de 2048 palavras de 4 B. Seguindo o princípio da localidade espacial, a quantidade de acessos ao mesmo bloco é muito grande por parte de um núcleo, porém a quantidade de endereços é menor, com isso, permite-se que todos os núcleos acessem a memória compartilhada simultaneamente, desde que cada um tente acessar bancos de memória diferentes. Além disso, o consumo de energia é menor para menores bancos de memória.

4.4 Gerenciamento de Tarefas

A distribuição de tarefas é um processo fundamental para o funcionamento de arquiteturas *manycore*. É o processo por meio do qual faz-se todo o sistema funcionar e que, por alguma falha, pode provocar desequilíbrio de carga de trabalho que posteriormente pode acarretar no desgaste de uns recursos em detrimento de outros, isto é, deteriorar alguns recursos mais rápido que outros.

Em *ArachNoc*, nos nós escravos, a distribuição de tarefas é realizada pelo módulo de gerenciamento de tarefas e interrupções (MGTI). O nó mestre ao receber um sinal de interrupção, salta para a rotina de tratamento da interrupção correspondente, armazenadas nas memórias dedicadas, representadas no lado esquerdo do núcleo mestre na Figura 17. Se for uma *trap EXEC*,*SYNEXEC* ou *SYNC*, ao final da rotina de tratamento o mestre envia, um processo pronto para ser executado, para o MGTI.

O MGTI do nó mestre envia, através da rede em *chip*, pacotes com os dados do processo a ser executado para o nó escravo que enviou a *trap*. Ao chegar no nó escravo o pacote é desempacotado e o processo é armazenado no *buffer* de tarefas de um dos núcleos escravos, que posteriormente o enviará para um núcleo processá-lo.

O módulo gerenciador de tarefas contém algumas informações úteis a respeito dos *buffers* dos núcleos de processamento, como: quantidade de tarefas que cada um possui em fila, se o *buffer* está vazio ou cheio, e se o processador está ocioso ou não. De posse dessas informações, o módulo é capaz de distribuir tarefas entre os escravos, mantendo o equilíbrio de carga de trabalho, e garantir que todos os processadores estejam operando (balanceamento de recursos).

Para a distribuição de carga de trabalho, o módulo prioriza os escravos ociosos. Caso não haja ociosos, a prioridade passa a ser dos escravos que possuírem menor quantidade

de tarefas no *buffer*.

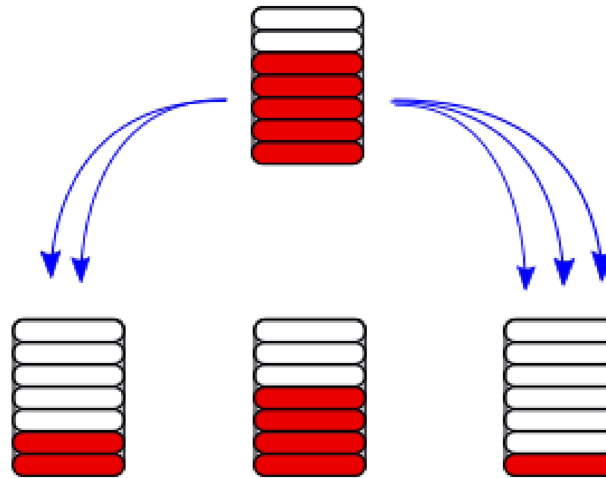

Figura 18 – Simulação do processo de distribuição de tarefas entre o *buffer* de entrada e os *buffers* de cada escravo do MGTI de um nó escravo.

Na Figura 18 é apresentada uma simulação de distribuição de tarefas, dentro do MGTI do nó escravo, entre o *buffer* de entrada do MGTI e os *buffers* dos núcleos de processamento. Na Figura 18 o *buffer* mais à direita, com menor quantidade de tarefas, possui maior prioridade para receber tarefas (característica representada pela quantidade de setas direcionadas para tal). Logo em seguida o *buffer* mais a esquerda passará a ter prioridade até que as tarefas a serem enviadas se esgotem. O processo se repetirá sempre que houver tarefa no *buffer* de entrada do MGTI.

4.5 Sistema de Comunicação e Sincronização

Em arquiteturas *manycore*, devido ao paralelismo, há muitas e frequentes trocas de contexto. Isso exige um sistema operacional que administre tais trocas de contexto de modo a manter cada tarefa em seu fluxo de execução correto, mantendo a coerência dos dados. Em um nível mais próximo do hardware, isso acontece pela troca de ponteiros, que endereçam processos gravados na memória, entre os processadores.

Por exemplo, para o processador I executar uma tarefa J, basta que o fluxo de execução do mesmo seja alterado para o endereço de memória no qual encontra-se a primeira instrução da tarefa J, isto é, escreve-se, tal endereço, no registrador PC (*program-counter*) deste processador. Dessa forma, para alocar ou desalocar recursos basta operar sobre o registrador *program-counter* dos núcleos de processamento do sistema.

O sistema de comunicação e sincronização do *ArachNoC*, adaptado do modelo IPNoSys, é baseado em tarefas, isto é, cada processador recebe diferentes processos de aplicações. Isso, exige um sistema operacional, senão, no mínimo um gerenciamento de

tarefas, para que sejam realizadas as trocas de contexto e alocação e desalocação de recursos.

Esse sistema permite que as tarefas em execução troquem dados e informações entre si. Composto pelas instruções *EXEC, SYNC, SYNEXEC, SEND* e *RECEIVE*, ele permite que uma tarefa dispare execuções paralelas de outras tarefas, e permite também que estas tarefas paralelas sejam sincronizados, a fim de concluírem suas execuções de maneira ordenada.

ArachNoc, segue o modelo de programação de IPNoSys, onde o paralelismo é explicito, ou seja, o programador é responsável pelo desenvolvimento de um código de aplicação, onde a invocação (chamada) de execução síncrona ou assíncrona de novas tarefas é explicita. Isto é feito com as instruções EXEC e SYNEXEC, enquanto a instrução SYNC emite sinais de sincroninazação. A comunicação entre tarefas é feita com a instrução SEND, que envia dados a serem armazenados nos pacotes, que são as unidades de código da aplicações em IPNoSys. Na arquitetura IPNoSys, estas instruções geram pacotes de controle que são encaminhadas a uma unidade específica de hardware *(MAU - Memory Access Unit)* que verifica a necessidade e oportunidade de iniciar a execução de novas tarefas.

Em *ArachNoC*, estas instruções *(EXEC, SYNEXEC, SYNC e SEND)* foram adaptadas para o padrão da ISA MIPS, bem como, foram adaptadas para execução em uma arquitetura *manycore*. Adicionalmente, foi criada a instrução *RECEIVE*, para permitir programação com troca de mensagens.

Como já comentado anteriormente, os núcleos de processamento, presentes nos nós escravos, executam as tarefas, eventualmente executando tais instruções de comunicação e sincronização. Quando executadas, estas instruções geram *traps*, que são encaminhadas ao nó mestre para verificar se uma nova tarefa deverá ser enviada para execução. Dada a atual inexistência de um sistema operacional, o programador também é responsável pela criação da estrutura que representa as aplicações e pelo desenvolvimento do código executado no nó mestre, para dar início da execução de novas tarefas. Em razão disso, foi criada um biblioteca, chamada **app_descriptor.h** que possui rotinas para a criação automática dessa estrutura e rotinas de tratamento de interrupções e eventuais condições de corrida.

Esta estrutura, que armazena todas as informações necessárias à execução das instruções de comunicação e sincronização, é chamada de descritor de aplicações. O descritor é armazenado em uma porção da memória, na qual fica armazenada uma estrutura com os dados das aplicações e suas respectivas tarefas. A representação desta estrutura pode ser vista na Figura 19.

4.5.1 Descritor de Aplicações

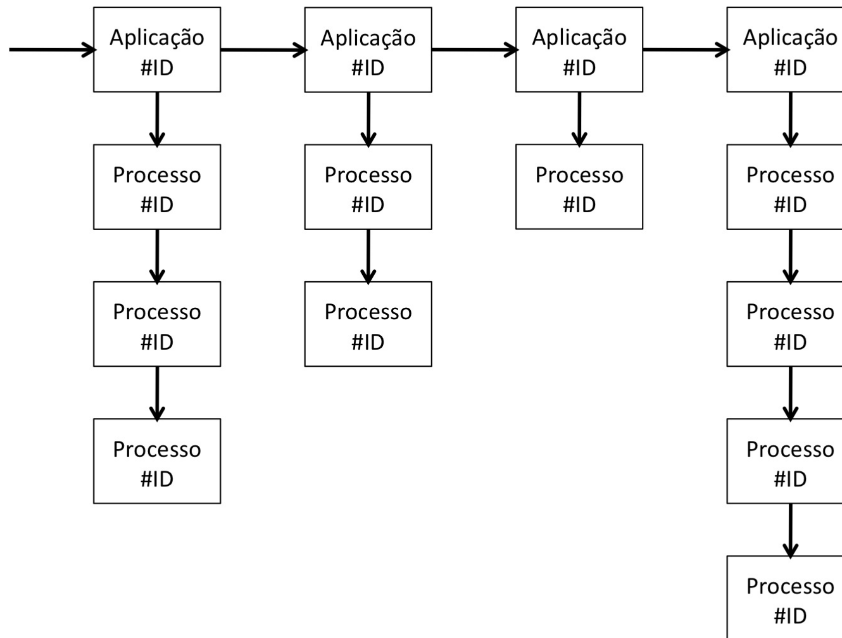

Figura 19 – Representação esquemática do descritor de aplicações.

O **descritor de aplicações** é uma estrutura de dados no formato de matriz, constituída por um vetor de aplicações no qual cada aplicação é constituída por um vetor de tarefas. Desta forma, todo acesso à matriz é indexado pelo *ID* da aplicação, e somente depois, busca-se o *ID* da tarefa.

Durante a execução das aplicações, este espaço de memória é acessado sempre que o nó mestre precisa tratar uma interrupção.

A construção do descritor de aplicações é realizada em tempo de execução. Como o descritor muda de acordo com a aplicação, é possível determinar como o descritor será gerado, por meio de funções pertencentes à biblioteca desenvolvida e deixar que o nó mestre construa-o em tempo de execução.

4.5.2 Adaptação das instruções de comunicação e sincronização

A fim de adaptar o subconjunto de instruções de sincronização e comunicação para um *manycore* convencional, as instruções *Exec, Sync, SynExec, Send* e *Receive* foram implementadas conforme explanado a seguir.

4.5.2.1 Exec

A instrução Exec foi adaptada para o formato I do padrão MIPS, como mostrado na Figura 20.

Figura 20 – Representação do formato da instrução Exec.

Esta instrução corresponde a uma ordem de execução assíncrona, o que significa que a tarefa não requer sincronização e pode ser iniciada tão logo seja possível. Na codificação adotada, REG_D e REG_O são dois registradores, do banco de registradores MIPS, que contêm dois pares de identificadores (ID_aplicação, ID_tarefa). O registrador REG_D contém o par de identificadores da Aplicação/Tarefa de destino (requisitada), enquanto que o registrador REG_O contem o par de identificadores da tarefa de origem (requerente).

Ao executar uma instrução *Exec*, o núcleo de processamento envia durante a sinalização da *trap*, os dois pares de identificadores para o módulo de gerenciamento de tarefas do nó mestre. No caso de *traps* de *Exec*, apenas o par de IDs da tarefa chamada será útil para o nó mestre pois, por ser uma ordem de execução assíncrona, é suficiente identificar qual tarefa deve ser executada.

Durante a execução da rotina de tratamento da *trap* de *Exec*, o nó mestre busca, no descritor de aplicações, o endereço inicial do processo identificado pelo *ID* armazenado no registrador REG_D. De posse desse, o processador do nó mestre retorna-o para o MGTI do nó mestre, que por sua vez, providenciará o envio da tarefa para um nó escravo.

4.5.2.2 Sync

A instrução Sync também foi adaptada para o formato I, conforme apresentado na Figura 21.

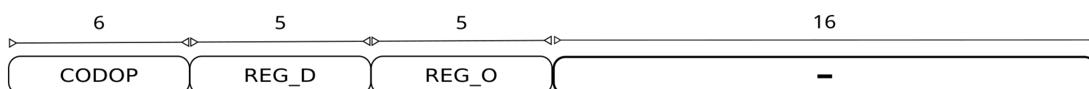

Figura 21 – Representação do formato da instrução Sync.

Sync é usada para enviar um sinal de sincronização de uma tarefa para outra. Na Figura 21, o campo REG_O é o endereço de um registrador cujo conteúdo é um número binário que identifica a aplicação e a tarefa que enviam o sinal de sincronização; e o campo REG_D é o endereço de um registrador cujo conteúdo é um binário que identifica a aplicação e o processo que aguardam o sinal de sincronização.

Ao receber uma *trap SYNC*, o núcleo mestre executará uma rotina de tratamento que consiste em acessar o decritor de aplicações, identificar a aplicação e a tarefa que aguardam o sinal de sincronização e registrar a chegada desse sinal, identificando também quem o enviou, para que não se confundam sinais de sincronização de aplicações diferentes.

4.5.2.3 SynExec

A instrução SynExec também é montada no formato I como mostrado na Figura 22.

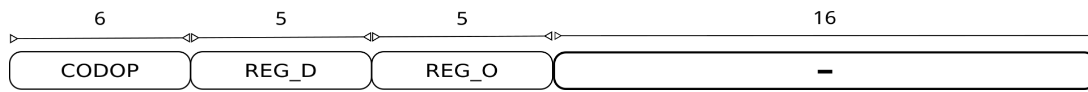

Figura 22 – Representação do formato da instrução SynExec.

Esta instrução corresponde a uma ordem de execução síncrona, isto é, determina que a tarefa chamada, identificada pelo conteúdo do registrador REG_D, somente seja executado quando determinados sinais de sincronização, (*syncs*), tiverem chegado. O registrador REG_O, contém os *IDs* da aplicação e da tarefa que enviam a ordem de execução síncrona.

Na instrução de *SynExec* não há espaço em bits para identificar quais tarefas irão sincronizar para que a tarefa chamada possa executar.

O descritor de aplicações deve, portanto, manter a informação de quantos sinais de sincronização são necessários e de quais tarefas eles serão enviados. Sempre que o nó mestre tratar uma rotina de *Sync* ou *SynExec*, registra no descritor os *Syncs* recebidos e se o *Syncexec* foi emitido ou não. Se todos os sinais de sincronização necessários tiverem sido emitidos e registrados, e se a ordem de execução já ocorreu, o processo está pronto para execução. Caso contrário, o nó mestre conclui o tratamento da *trap* e aguarda uma nova interrupção do módulo de gerenciamento.

4.5.2.4 Send e Receive

As instruções de *Send* e *Receive* possuem o mesmo formato das demais instruções, como mostrado na Figura 23.

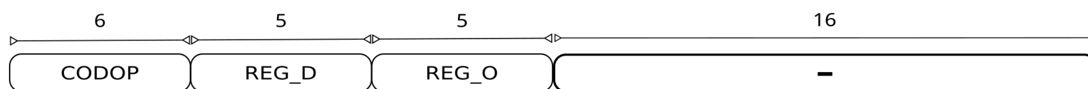

Figura 23 – Representação do formato das instruções *Send* e *Receive*.

A instrução *SEND* determina o envio de um conjunto de dados de uma tarefa para outra. Supondo-se a execução de uma instrução *SEND*, para enviar dados da tarefa 3 da aplicação 2 para a tarefa 1 da aplicação 5, o comportamento se dá da seguinte forma.

O núcleo de processamento envia para o MGTI, o par de *IDs* (ID da aplicação e ID da tarefa de destino dos dados, 5 e 1, respectivamente), o qual está armazenado no registrador REG_D, e o endereço de memória onde o pacote está salvo, que é o conteúdo do registrador REG_O. A instrução *SEND* é enviada para o MGTI, que possui uma tabela chamada **tabela de mapeamento de tarefas (TMT)**, como mostrado na Tabela 1.

Tabela 1 – Tabela de mapeamento de tarefas do MGTI de um nó escravo

ID_NUCLEO	ID_APP	ID_PROC	LA
0	1	3	15879
5	2	3	16453
2	1	8	45789

A tabela de mapeamento de tarefas armazena a informação de que tarefa está sendo executada, e por qual núcleo de processamento ela está sendo executada. Além disso, a tabela guarda também um apontador, que irá orientar o MGTI na hora de carregar conjuntos de dados para enviar para outros nós da rede, quando for realizado um *SEND*. E quando for realizado um *RECEIVE*, servirá para apontar, para o MGTI, onde armazenar dados vindo de outros nós da rede.

De posse do endereço do conjunto de dados enviado pelo núcleo de processamento (LA), o MGTI acessa a memória de dados e carrega todo o pacote. O MGTI irá compor um pacote de dados com: o par de *IDs* da tarefa que irá receber os dados; o par de *IDs* da tarefa que está enviando os dados, coletado da tabela; e o conjunto de dados, propriamente dito. Esse pacote é enviado para o nó mestre.

No nó mestre, o MGTI recebe o pacote de dados, extrai todas as suas informações e guarda-as numa tabela chamada **tabela de controle de pacotes (TCP)**, representada pela Tabela 2. Nessa tabela são guardadas informações necessárias para a sincronização entre cada *SEND* e seu respectivo *RECEIVE*. Cada linha da tabela registra que nó escravo (ID_NO) irá receber dados, de que tarefa (ID_AO | ID_PO), esses dados foram enviados, e para qual tarefa (ID_AD | ID_PD), eles deverão ser entregues.

Tabela 2 – Tabela de controle de pacotes do MGTI do nó mestre após receber SEND.

ID_NO	ID_AD	ID_PD	ID_AO	ID_PO	LA	S	R	STATUS
1	5	1	2	3	65536	1	0	1
0	3	4	2	1	14789	0	0	0
3	2	5	7	3	4097	0	1	1

Além disso, o MGTI necessita armazenar todo o conjunto de dados enviado. Para isso, ele utiliza um apontador para uma região da memória de dados reservada para armazenamento de pacotes de dados, representado na tabela pela coluna (*local address - LA*). Para cada pacote que chega, o MGTI lê o apontador e salva o novo pacote na memória. Em seguida, atualiza o apontador para a próxima região livre. Por fim, o MGTI registra que chegou um *SEND* (**S** = TRUE) e atualiza o *STATUS* dessa linha da tabela, indicando que a mesma está em uso.

Quando um núcleo de processamento executar a instrução *RECEIVE* da tarefa 1 da aplicação 5, ele ficará **bloqueado**, aguardando o retorno dos dados que requisitou.

O núcleo de processamento envia para o MGTI o par de *IDs* (ID da aplicação e ID da tarefa de origem dos dados, 2 e 3, respectivamente), o qual está armazenado no registrador REG_D, e o endereço de memória onde, o pacote que chegará, será salvo, que é o conteúdo do registrador REG_O.

O MGTI do nó escravo, vai consultar a TMT para identificar que tarefa está requisitando dados (ID_APP | ID_PROC), e em que endereço (LA) esses dados deverão ser salvos. Para isso, a mesma utiliza o ID_NUCLEO.

De posse dos *IDs* da tarefa que realiza o *RECEIVE*, o MGTI irá compor um pacote de dados com: o par de *IDs* da tarefa que está requisitando os dados, coletado da tabela; o par de *IDs* da tarefa que enviou os dados; e o endereço de onde os dados serão armazenados. Esse pacote é enviado para o nó mestre.

No nó mestre, o MGTI recebe o pacote de dados e extrai todas as suas informações. Ao identificar que é uma interrupção do tipo *RECEIVE*, o MGTI busca pela linha de **STATUS** ativo, cujos campos ID_AO e ID_PO, correspondem ao par de *IDs* da tarefa que enviou os dados, no caso, 2 e 3, respectivamente. O MGTI, confere os campos ID_AD e ID_PD, verificando se correspondem ao par de *IDs* da tarefa que está requisitando os dados, no caso, 5 e 1, respectivamente. Em seguida, o MGTI verifica se o respectivo *SEND* já chegou, verificando se o campo **S** tem valor TRUE. Se o pacote de dados não houver chegado ainda, o MGTI adiciona uma linha na tabela para registrar a chegada do *RECEIVE* e, o núcleo de processamento que o enviou, permanecerá bloqueado.

Se o pacote de dados esperado, houver chegado, o MGTI registra que o *RECEIVE* chegou, (**R** = TRUE), e identifica para qual nó da rede o pacote deverá ser enviado (ID_NO), isto é, de qual nó escravo veio o *RECEIVE*. A TCP será atualizada como mostrado na Tabela 3.

Tabela 3 – Tabela de controle de pacotes do MGTI do nó mestre após receber o RECEIVE.

ID_NO	ID_AD	ID_PD	ID_AO	ID_PO	LA	S	R	STATUS
1	5	1	2	3	65536	1	1	1
0	3	4	2	1	14789	0	0	0
3	2	5	7	3	4097	0	1	1

Em seguida, o MGTI, com o endereço de onde o conjunto de dados está armazenado na memória do nó mestre, LA, carrega todo o conjunto de dados e forma um pacote endereçado para o nó 1, (ID_NO), o qual requisitou os dados, identificando as tarefas de origem (ID_AO | ID_PO), (2 | 3), e de destino (ID_AD | ID_PD), (5 | 1).

No nó escravo 1, o MGTI irá consultar a TMT, para identificar o núcleo de processamento que está aguardando os dados, e irá salvar o conjunto de dados na região de memória indicada pelo apontador (LA). Em seguida, o MGTI desbloqueia o núcleo

de processamento que aguardava pelos dados. No exemplo, o núcleo 2, como mostra a Tabela 4.

Tabela 4 – Tabela de mapeamento de tarefas do MGTI do nó escravo 1.

ID_NUCLEO	ID_APP	ID_PROC	LA
0	1	3	15879
5	2	3	16453
2	5	1	45789

4.6 Ambiente de Desenvolvimento

A programação para sistemas *manycore* tem sido alvo de pesquisas por diversos motivos, sendo o principal deles a forma de paralelizar as aplicações de modo a extrair o desempenho máximo da arquitetura. Assim, o desenvolvimento de um ambiente de programação, vai desde o modelo de programação à linguagem de programação passando pelas bibliotecas de funções e geração do código *assembly*. Em todo caso, ele deve permitir o melhor paralelismo e conforto possível ao programador.

ArachNoc é um *manycore* com memória compartilhada e a comunicação entre os núcleos ocorre por meio do **compartilhamento de variáveis**. Dessa forma, o desenvolvimento de aplicações para o mesmo exige um modelo de programação baseado em variáveis compartilhadas.

Algumas bibliotecas e *frameworks* têm sido desenvolvidos para permitir a programação paralela nos mais variados tipos de arquiteturas, como OpenMP, POSIX e MPI. Essas bibliotecas são compatíveis com algumas linguagens de programação consagradas, como C,C++ e Fortran. Essas linguagens, por serem bastante utilizadas, garantem a utilização e o progresso das APIs (*Application Programming Interface*).

Porém, em Sanchez et al. (2012), Diaz, Munoz-Caro e Nino (2012), Shekhar et al. (2011) pode-se visualizar a aplicabilidade de cada API, o que ajuda a entender que, de acordo com a hierarquia de memória, determinada biblioteca, ou API, pode proporcionar desempenho melhor que outras. Isso limita sua usabilidade e portabilidade. Para desenvolver aplicações para *ArachNoC* há duas possibilidades, por meio de compilação cruzada ou não-cruzada. A primeira possibilidade utiliza a linguagem C com compilação cruzada e uma metodologia de desenvolvimento de código paralelo, desenvolvida no escopo desta Dissertação de mestrado. A segunda utiliza a linguagem OpenCl e o compilador Ocean *(OpenCL Compiler to ArachNoc)* (FERREIRA, 2016), desenvolvido pelo mesmo grupo de pesquisa que o autor dessa dissertação participa.

4.6.1 Compilação Cruzada

Um compilador cruzado é capaz de gerar código executável para uma arquitetura diferente daquela na qual ele está instalado. Portanto, é uma ferramenta muito útil quando se trata de escrever aplicações para arquiteturas em desenvolvimento, já que na maioria das vezes o programador compila em uma arquitetura distinta da que está desenvolvendo.

Dentre as arquiteturas-alvo, algumas delas são bastante conhecidas e utilizadas na construção de diversos dispositivos, por isso são alvos já consagrados de compiladores cruzados, são elas : *PowerPC, SPARC, ARM* e *MIPS*.

Durante o desenvolvimento de aplicações para *ArachNoc*, o compilador cruzado é fundamental, exatamente porque através dele é possível não só gerar códigos para o processador *MIPS*, como também ligar os códigos-fonte utilizando um *script* de descrição de memória (*linker script*). Neste o programador determina que espaço de memória cada parte do código irá ocupar, ou seja, onde alocar determinada função ou variável.

A opção de utilizar a linguagem C para o desenvolvimento de aplicações para *ArachNoc* se deve à universalidade da linguagem já consolidada e, também, à propriedade de permitir acessos a espaços de memória e registradores específicos. O programador tem controle sobre alocação e desalocação de espaços de memória, através de ponteiros e pode manusear registradores através do modificador *register*.

É possível programar para *MIPS* utilizando a linguagem C por meio da compilação cruzada, disponibilizada pela GNU GCC. No entanto, esta programação exige um nível de conhecimento prévio da estrutura e organização da memória e dos registradores da arquitetura-alvo. Por isso, foram desenvolvidas duas bibliotecas a serem utilizadas pelo programador.

Uma delas, chamada **app_descriptor.h**, facilita a paralelização do código, criando e organizando todas as possíveis *threads*, de modo a conectá-las de acordo com o sistema de comunicação e sincronização. A outra, chamada **mutex.h**, trata as possíveis "condições de corrida" que poderiam acontecer, por meio da implementação de *mutexes*.

Dessa forma, o programador só precisa chamar as funções que lançam interrupções e estas mesmas já possuem um controle baseado em *mutex*. Uma explicação mais detalhada, com um exemplo de aplicação, é apresentada no Apêndice A.

Todo esse aparato em *software* só funciona corretamente se obedecidas algumas regras de programação, as quais estão descritas pelo padrão de programação paralela adotado para *ArachNoc*, *Fork-Join*. O padrão de programação *fork-join*, ilustrado na Figura 24, determina que existe um processo "iniciador" que se divide, (*fork*), em vários processos paralelos, que também podem se sub-dividir, mas ao final da aplicação, convergem para um único processo "finalizador" (*join*).

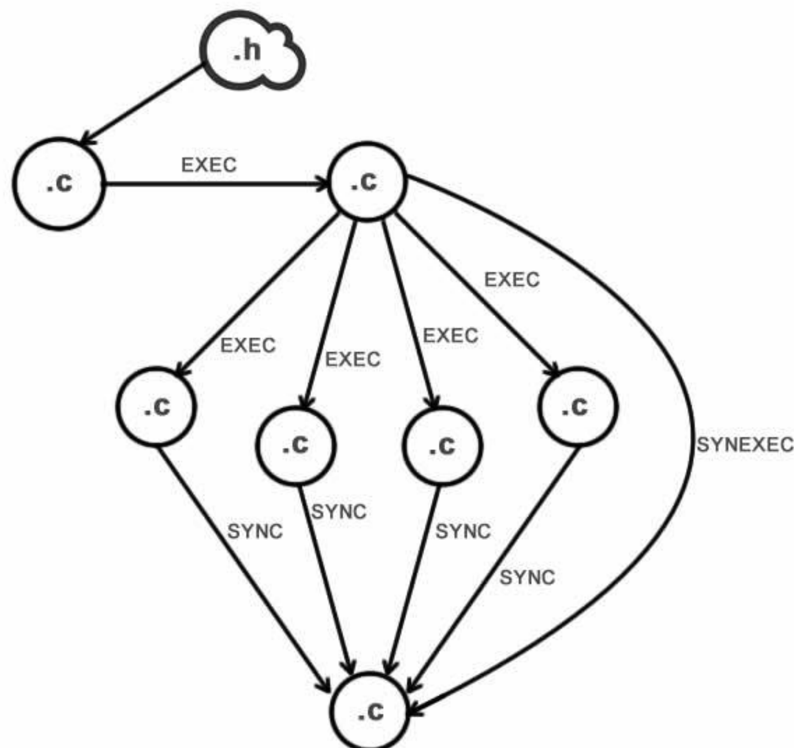

Figura 24 – Representação do padrão de programação para ArachNoC.

Portanto, com um modelo de programação definido, o programador possui uma especificação de quais arquivos ele deve escrever e como escrevê-los, mantendo a conectividade e garantindo o funcionamento do sistema de comunicação e sincronização. Esse modelo define qual a função de cada *thread* e quais sinais de sincronização cada uma delas envia ou recebe.

4.6.2 O compilador *Ocean*

OpenCL é um *framework* para linguagem de programação C, que permite a programação paralela para arquiteturas *manycore* heterogêneas, tais como as *GPGPUs*.

OpenCL segue o padrão de programação adaptado para arquiteturas *SIMD*. Duas funções principais compõem esse *framework*, a função *host*, que inicializa a aplicação e define um conjunto de tarefas a serem executadas, a esse conjunto é dado o nome de *work-group*. A segunda função, chamada *kernel*, contém o código que será executado por todas as unidades de processamento, com diferentes dados de entrada, cada execução paralela, com seus devidos dados de entrada, tem o nome de *work-item*.

O compilador *Ocean* é composto pela ferramenta LLVM, que constitui seu *frontend*, e por um *backend MIPS*, desenvolvido com uma API de funções para adaptação à *ArachNoC*: __exec, __synexec, __sync, __send, __receive.

4.7 Considerações Finais

A arquitetura apresentada neste trabalho possui uma configuração de memória compartilhada e portanto possui um modelo de programação paralela adaptado para tal.

ArachNoc é um *manycore* composto por nós *multicore* de processamento, com exceção do nó mestre, que possui apenas um processador. Este é destinado a enviar *threads* para os nós escravos e tratar interrupções oriundas dos mesmos. Este nó não executa tarefas.

Cada nó escravo em ArachNoc possui 8 processadores e uma memória compartilhada dividida em 128 blocos de 8 KB cada, sendo 64 blocos de memória de instruções e outros 64 de memória de dados. Faz-se necessário uma memória compartilhada para os núcleos escravos poderem executar processos iguais, para isso eles se alternam no acesso aos blocos de memória. Com isso, evita-se desperdício de memória, repetição de código e permite-se que todos os núcleos compartilhem dados entre si.

Para *ArachNoc* foi desenvolvido um ambiente de programação paralela, com modelo de programação adaptado da plataforma IPNoSys (ARAUJO, 2012), bibliotecas para o gerenciamento de *threads* e interrupções, e dois compiladores, um compilador cruzado, *GCC to ArachNoc*, e o compilador Ocean.

5 Resultados

Em arquiteturas paralelas, a existência de mais de um núcleo de processamento exige que a programação de aplicações seja capaz de gerar código para cada um se seus núcleos. Para tal, existem diversos *frameworks* e bibliotecas para permitir a programação paralela.

Para *ArachNoc*, foram utilizados dois compiladores: um compilador cruzado *GCC* e o compilador *Ocean*.

A fim de explorar a capacidade de comunicação entre processos, por meio do compartilhamento de variáveis, e o poder de processamento do sistema, foram desenvolvidas aplicações paralelas com diferentes metodologias de paralelização.

Dentre as aplicações desenvolvidas, encontra-se: Algoritmo de *Dijkstra*,Filtro Laplaciano, Algoritmos Genéticos e Multiplicação de Matrizes.

5.1 Experimentos com Compilador Cruzado

O compilador cruzado *GNU GCC* compila códigos descritos na linguagem C ou C++. Esse compilador pode ser configurado para gerar código *assembly* de diferentes arquiteturas, inclusive *MIPS*.

As aplicações foram desenvolvidas seguindo o padrão de programação paralela *ForkJoin*. Devido ao fato de o compilador cruzado gerar apenas código sequencial, cada processo da aplicação foi desenvolvido em um arquivo fonte (**.c**). Unindo todos os arquivos fonte da aplicação, com a biblioteca **app_descriptor.h** e o *script* de configuração de memória, **MIPS_IPS.ld**, torna-se possível gerar aplicações paralelas para *ArachNoc*.

Dessa forma, a seguir, cada aplicação será descrita de maneira detalhada, sendo especificados as dimensões da aplicação, o modo como foi paralelizado e o objetivo da simulação dessa aplicação. Todas elas serão avaliadas quanto a alguns parâmetros:

- **Tempo de Execução:** essa métrica expressa a quantidade de ciclos de operação que a aplicação leva desde o início da execução do processo inicial, até o fim do processo finalizador.
- **Tempo de Ociosidade:** significa a soma dos tempos, em quantidade de ciclos de relógio, que os processadores passam sem realizar operação alguma (estado *IDLE*).
- **Tempo de Inicialização:** é o tempo que vai desde quando o processador do nó mestre monta o descritor de aplicações, até quando ele envia a tarefa inicial da

aplicação para um nó escravo.

- **Tempo de Gerenciamento:** é o tempo gasto, pelo processador do nó mestre, para tratar interrupções.

Para todos os gráficos, o tempo total corresponde à soma do tempo de execução com o tempo de inicialização. Enquanto que, o **tempo de ociosidade** e o **tempo de gerenciamento** fazem parte do tempo de execução e foram destacados nos gráficos apenas para expressá-los em relação ao tempo total.

5.1.1 Multiplicação de Matrizes

A aplicação de multiplicação de matrizes é bastante conhecida no âmbito de programação paralela, pois é uma aplicação naturalmente paralela.

Nessa aplicação, deseja-se multiplicar uma matriz A por uma matriz B. Para esse experimento, um processo paralelo consiste na multiplicação de uma linha da matriz A, por todas as colunas da matriz B. Dessa forma, cada processo paralelo gera como resultado uma linha da matriz resultante.

A aplicação foi desenvolvida para ser executada em apenas um nó de *ArachNoc*. Com multiplicação de matrizes foram realizados dois experimentos diferentes. Em um deles o código do processo de multiplicação, de uma linha de uma matriz, pelas colunas da outra matriz, é enviado para um único bloco da memória, dessa forma, em um certo ponto da execução, todos os núcleos de processamento competem pelo acesso a esse bloco. No outro, o código de cada multiplicação paralela foi armazenado em um bloco de memória diferente. Isso, configura um gargalo no sistema, e foi programado no intuito de se avaliar o custo desse atraso, de competir por um bloco de memória, ao tempo total de execução.

Sendo executada em apenas um nó, força-se, a distribuição de mais de uma tarefa para cada núcleo escravo. Dessa forma, com uma matriz de 16 linhas e 16 colunas, foram criados 256 processos. Como o nó escravo possui 8 núcleos de processamento, cada núcleo executou 32 processos.

Dessa forma, foi verificada a eficiência do algoritmo de distribuição de carga de trabalho. Paralelamente, também é avaliada a eficiência do sistema de comunicação de sincronização entre processos. Pois, no algoritmo, um processo inicial dispara execuções paralelas de outros processos, por meio de *execs*, e ordena a execução síncrona do processo finalizador (*synexec*). O processo finalizador é responsável por unir os resultados das multiplicações paralelas em uma só região de memória. Esse só será realizado quando todos os *syncs*, de todos os processos paralelos tiverem chegado.

O gráfico da Figura 25 apresenta os dados coletados. Esse gráfico, mostra que o tempo de execução é aproximadamente 99% to tempo total de simulação. O tempo de

ociosidade dos núcleos de processamento, isto é, o tempo sem realizar operação alguma, até receber nova tarefa, é aproximadamente 8%. Além disso, o processador do nó mestre, gasta um tempo para configurar o descritor de aplicações e disparar as execuções paralelas que é aproximadamente, 1,5% do tempo total de simulação. Com tratamento de interrupções o mesmo gasta quase 7,5% do tempo total de simulação.

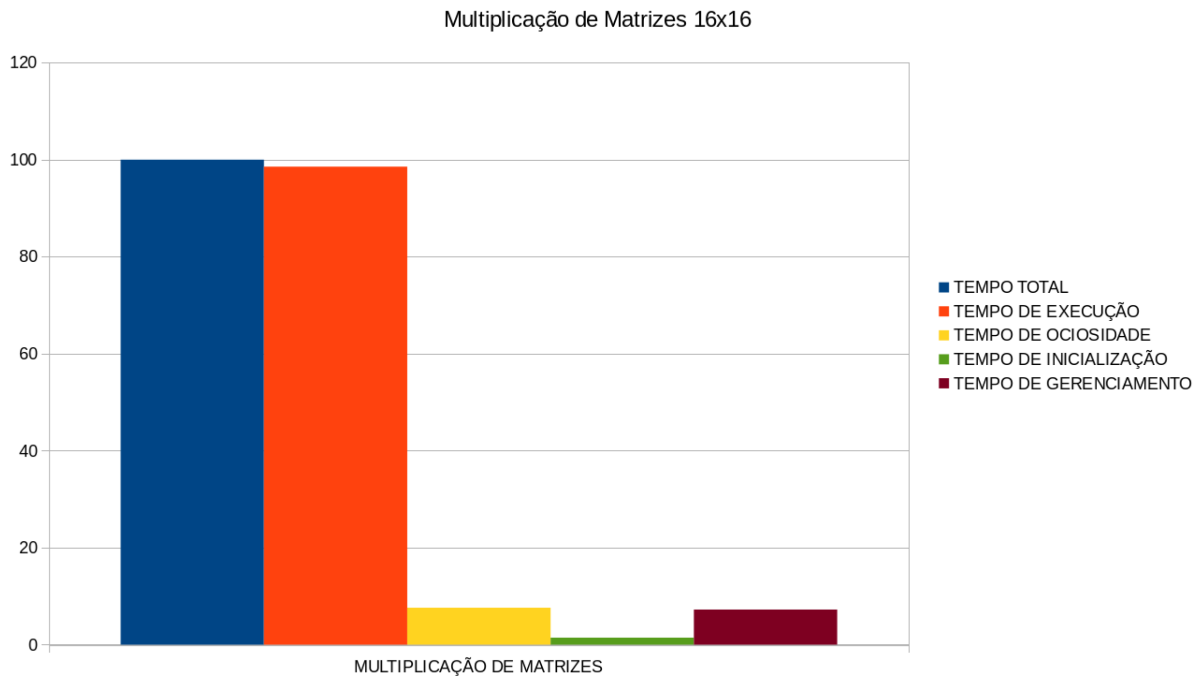

Figura 25 – Gráfico da multiplicação de matrizes de dimensões 16x16.

5.1.2 Algoritmo de *Dijkstra* e Filtro Laplaciano

Para esse experimento foram utilizados dois nós escravos da rede, cada um executando uma aplicação. Esse experimento tem o objetivo de mostrar que *ArachNoc* é capaz de executar diferentes aplicações ao mesmo tempo.

Como são executadas duas aplicações e o nó mestre é quem ordena o início de cada aplicação, de maneira serial, os processos da última aplicação inicializada terão tempo de espera maior para iniciarem a execução. Nesse experimento, *Dijkstra* foi enviado para o nó 3 de *ArachNoc*, e Laplaciano para o nó 6, nessa respectiva sequência.

O algoritmo de *Dijkstra* é um algoritmo guloso, usado para solucionar o problema do caminho mais curto num grafo, dirigido ou não dirigido, com arestas de peso não negativo. O algoritmo soluciona o problema em tempo computacional O([m+n]log n), onde **m** é o número de arestas e **n** é o número de vértices.

O filtro Laplaciano é utilizado para a obtenção dos contornos de uma imagem. O Laplaciano, *L(x,y)*, de uma imagem, é a derivada segunda do valor da função que descreve a intensidade dos seus pixels, *I(x,y)*, sendo representado por:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Considerando-se que a imagem é armazenada como uma coleção de *pixels* discretos, é necessário produzir uma **forma discreta da derivada segunda** que é representada por um núcleo de convolução (máscara ou *kernel*), por exemplo, como as da Figura 26.

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 26 – Aproximações do filtro Laplaciano.

Dessa forma, ao aplicar-se o filtro laplaciano em uma imagem, ele produz como resultado uma nova imagem, na qual todas as bordas encontradas são realçadas.

O algoritmo de Dijkstra foi paralelizado de forma que a tarefa inicial dispara as buscas nos nove nó do grafo. Enquanto isso, o filtro Laplaciano foi paralelizado de modo que cada núcleo de processamento trata 4 linhas de pixels da matriz de entrada e salva o resultado em um bloco de memória. Cada execução paralela envia um *Sync*, quando terminar, para o processo finalizador. Este por sua vez, quando tiverem chegado todos os *Syncs* e o *SynExec*, irá carregar todos os reslultados parciais e salvar em um único bloco de memória.

Devido ao fato de ser enviada depois de *Dijkstra*, a aplicação de detecção de borda utilizando o filtro Laplaciano possui maior tempo de inicialização, que é aproximadamente 7% do tempo total de simulação, enquanto que o tempo de inicialização de *Dijkstra* é de aproximadamente 6%. Essa diferença de algo em torno de 1% corresponde ao tempo necessário para tratar a interrupção de *Exec* para a primeira tarefa do Laplaciano.

Além disso, o grafo de entrada de Dijkstra possui apenas 9 nós, enquanto que a imagem a ser tratada pelo filtro Laplaciano é de 32 x 32 pixels. Consequência disso, é que o tmepo de execução do Laplaciano é em torno de 92% do tempo total de simulação, e para o algoritmo de *Dijkstra* é de 35,7%. Os tempos de gerenciamento variam, de um para outro, em 5%. Para o filtro Laplaciano o tempo de gerenciamento corresponde a 20,1% e para o *Dijkstra* é 15,8% do tempo total de simulação. Essa diferença corresponde aos tratamentos de *Sync* e *SynExec*, necessários no Laplaciano.

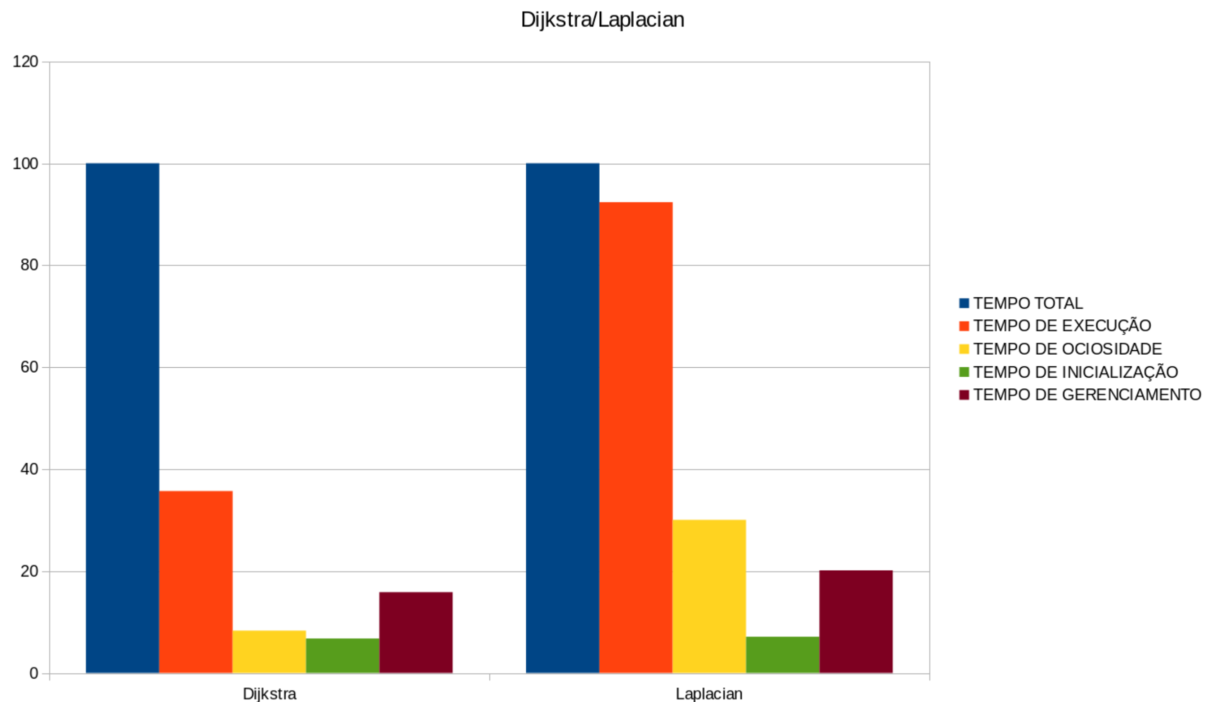

Figura 27 – Gráfico da aplicação com Algoritmo de Dijkstra e Filtro Laplaciano.

5.1.3 Algoritmo Genético

Um algoritmo genético (AG) é a técnica de busca utilizada na ciência da computação para encontrar soluções aproximadas em problemas de otimização e busca. Algoritmos genéticos são uma classe particular de algoritmos evolutivos que usam técnicas inspiradas pela biologia evolutiva como hereditariedade, mutação, seleção natural e recombinação(*crossing over*).

Para a simulação em *ArachNoc*, um AG foi aplicado para solucionar o problema das **N** rainhas. Esse problema consiste em encontrar a melhor forma de dispor **N** no tabuleiro de xadrez, de modo que nenhuma delas possa ser capturada por nenhuma outra.

O *AG* implementado gera tabuleiros de xadrez com diferentes disposições de rainhas, isto é, diferentes **indivíduos**. A execução de um *AG* possui parâmetros de configuração como: **probabilidade de recombinação**, que determina quais características dos indivíduos devem ser recombinadas; e **pressão de seleção**, que determina quantos indivíduos passam de uma geração para outra.

Como uma só execução de um *AG* pode nunca achar a solução ótima, uma técnica muito utilizada é chamada **mapa de controle**. Um mapa de controle dispara a execução de vários *AGs* com diferentes configurações.

Na simulação realizada foi implementado um mapa de controle, em um nó escravo de *ArachNoc*, para solucionar o problema das **N** rainhas para tabuleiros de tamanho 4x4. Dentro do nó, cada núcleo de processamento ficou responsável pela execução de um *AG*,

com configuração específica.

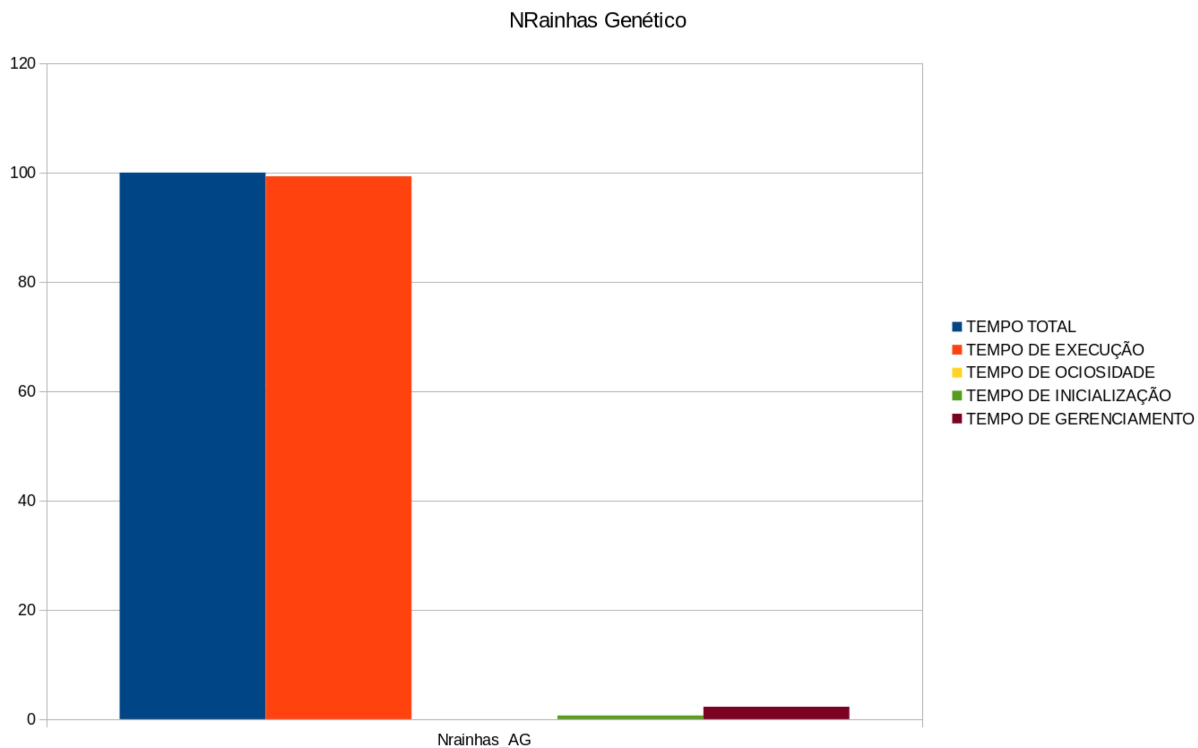

Figura 28 – Gráfico do algoritmo genético aplicado ao problema das N rainhas.

Como no mapa de controle cada execução de AG inicializa seus tabuleiros com valores aleatórios, o tempo de inicialização nessa aplicação é muito pequeno. O gráfico da Figura 28 mostra que o tempo de inicialização é aproximadamente 0,7% do tempo total de execução. Além disso, como a paralelização exige apenas que se dispare as execuções paralelas, não há muito sincronismo na aplicação, consequentemente, o tempo de gerenciamento também é muito pequeno, em torno de 2% do tempo total de execução.

O tempo de ociosidade, que significa o tempo que os processadores esperam por uma nova tarefa é imperceptível no gráfico, pois de uma geração para outra não há necessidade de sincronização.

Portanto, o tempo de execução, isto é, o tempo contado desde o início da primeira tarefa disparada pelo nó mestre até o final da última tarefa executada, é 99% do tempo de simulação.

5.1.4 Experimentos com Compilador *Ocean*

O compilador Ocean foi utilizado no desenvolvimento de aplicações para validar sua implementação e, avaliar o desempenho de *ArachNoc* ao executar códigos *OpenCL*. Para tal objetivo, utilizou-se um nó escravo de *ArachNoc* para executar o algoritmo de Dijkstra. Isso porque programação em *OpecnCL* possui um sistema de comunicação entre processos, baseado no compartilhamento de variáveis.

Para comparar o desempenho do compilador *Ocean* com o desempenho do compilador cruzado *GCC*, é necessário reduzir-se ao máximo a quantidade de variáveis a serem analisadas, isto é, para as duas simulações, deve-se usar os mesmos códigos-fonte, os mesmos dados de entrada, e o mesmo processador. Portanto, o grafo de entrada utilizado nessa simulação é o mesmo utilizado na simulação com o compilador cruzado, representado na **??**.

Em *OpenCL*, toda aplicação possui um processo inicial, que irá disparar as tarefas a serem executadas em paralelo, chamado *host*; e um processo cujo código será enviado para as unidades de processamento, chamado *kernel*. O *host*, ao enviar um código para ser executado nos dispositivos, cria um *work-group*. O *work-group* é composto pelas tarefas paralelas, que por sua vez são chamadas de *work-item*.

O experimento realizado calcula o menor caminho de cada nó do grafo para todos os outros. Dessa forma, nesse experimento, cada núcleo de processamento recebeu, como "nó de partida", um nó do grafo. Ou seja, o nó escravo recebeu 9 *work-items*.

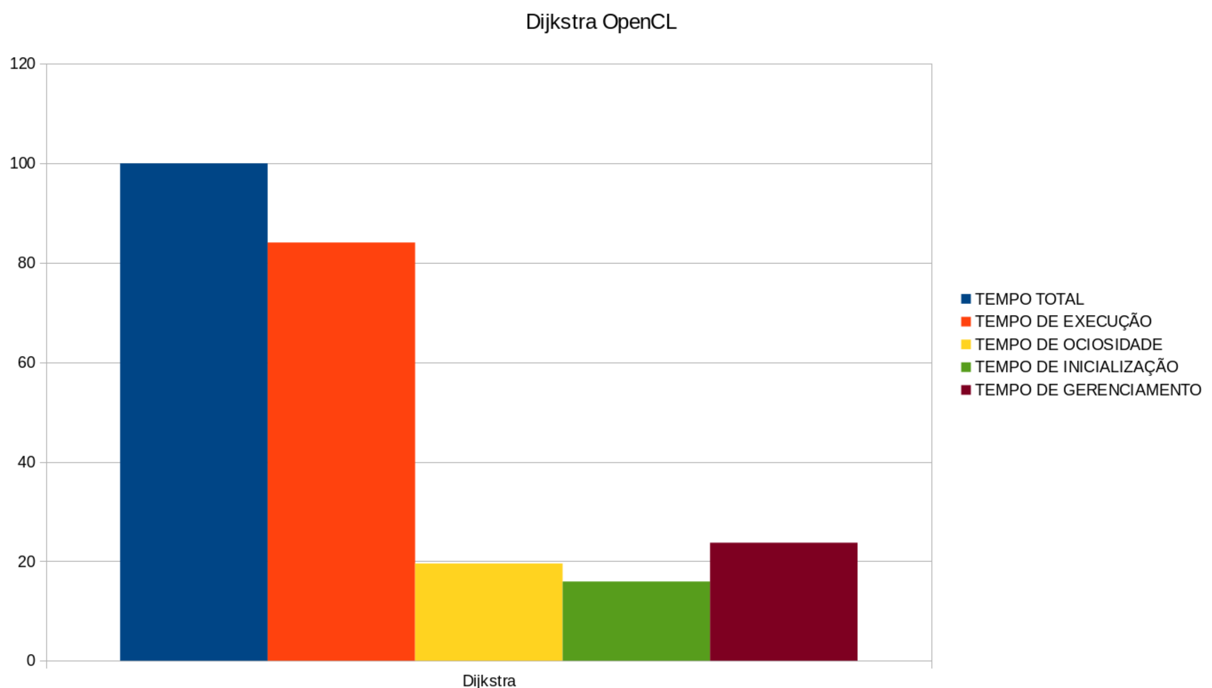

Figura 29 – Gráfico da simulação do algoritmo de *Dijkstra* com o compilador *Ocean*.

A metodologia de paralelização utilizada na programação *OpenCL* possui pouco comunicação e sincronização. Porém, o tempo de processamento dos *work-items* é pequeno, ou seja, é gasto um bom tempo para disparar as tarefas paralelas, que são rapidamente executadas. Por conta disso, nesse experimento, o tempo de gerenciamento é aproximadamente 23% do tempo total de execução, como mostrado na Figura 29. O que abrange os tempos de tratamento *EXECS* para os 9 *work-items*.

Além disso, pode-se perceber que, os núcleos de processamento passam um bom

tempo esperando seus *work-items*, pois o tempo de ociosidade é quase 20% do tempo total de execução. O tempo de inicialização, desde quando o nó mestre inicializa o descritor até envia a primeiro *work-item*, é aproximadamente 15% do tempo total de execução. Porém o tempo de execução, é 84% do tempo de simulação, ou seja, apesar de relativa ociosidade, a aplicação demonstrou ser capaz de extrair um bom desempenho de *ArachNoc*.

5.2 Implementação VHDL

Os nós de processamento de *ArachNoc*, possuem uma estrutura base, que provêm da estrutura do processador *multicore ArachNid*. Esta, possui uma versão em VHDL, que foi desenvolvida utilizando as ferramentas da Altera, Quartus II e ModelSim; onde são realizadas, síntese e prototipação, e simulação em software, respectivamente. Outra ferramenta utilizada foi o editor de texto Sigasi, no qual foi feita toda a implementação.

A implementação iniciou-se pela descrição dos menores componentes funcionais, conforme visto na Figura 30. Inicialmente foram descritos os núcleos de processamento, núcleo mestre (*Master Core*) e núcleos escravos (*Slave Core*). Após a validação isolada de cada um desses processadores, foram instanciadas duas memórias compartilhadas, uma de dados e outra de instruções. Ambas são de propriedade intelectual da Altera, que permite ao projetista utilizá-las.

Esses módulos foram instanciados e conectados entre si, a fim de construir um processador *multicore*. Para comunicá-los, foram implementados os barramentos de comunicação entre processadores e memória, e **módulos de gerenciamento de tarefas e interrupções** (*Trap Manager* na Figura 30). Por fim, foram instanciadas memórias dedicadas, com as rotinas de tratamento de interrupções, módulo *Memories* na Figura 30.

ArachNid, em sua versão VHDL, ainda não possui uma interface de conexão com elementos externos. Essa propriedade foi adicionada com a implementação em SystemC. Simulações em VHDL exigem muito tempo computacional, e as simulações em *ArachNid* já estavam muito custosas, contando com apenas 8 núcleos de processamento. A implementação de um *MPSoC* com nós *ArachNid*, em VHDL, exigiria um tempo de simulação impraticável. Devido a isso, decidiu-se pela mudança de linguagem de desenvolvimento, pois C++ (SystemC), permite simulações em um nível mais alto de abstração que VHDL, e por isso, as simulações são mais rápidas.

5.2.1 Experimentos com Compilação Cruzada

A fim de verificar o funcionamento do sistema de comunicação e sincronização desenvolvido, foi implementado um protótipo de processador *multicore*, com cinco núcleos, um mestre e quatro escravos. O protótipo possui menos núcleos que o modelo *ArachNid*, pois o experimento realizado visava apenas validar o funcionamento do sistema de comunicação

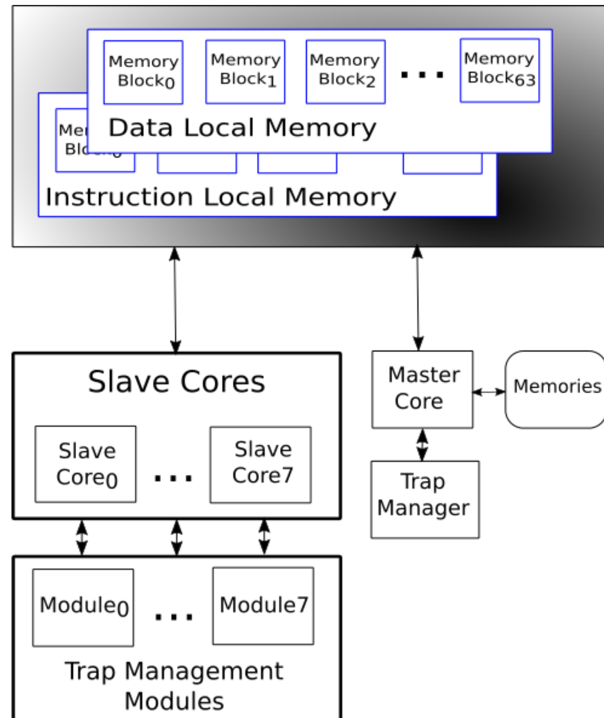

Figura 30 – ArachNid em módulos VHDL.

e sincronização, e a instanciação de oito núcleos escravos exigiria maior tempo de simulação. Esse protótipo possui uma memória compartilhada de 32 KB, tanto de dados quanto de instruções.

Tal modelo foi implementado utilizando-se a linguagem VHDL e apresentou uma frequência de operação em torno de 52 MHz.

Para a implementação das aplicações, foram utilizadas algumas ferramentas:

- Compilador cruzado: compilador GNU GCC configurado para gerar código assembly para MIPS32.
- ELF2MIF: foi desenvolvida em razão de, compilador gerar código *assembly* no formato ELF(*Executable and Linkable Format*) e as ferramentas de simulação utilizadas, não serem capazes de ler arquivos ELF. Tal ferramenta, converte ELF em MIF(*Memory Initialization File*), formato aceito pelas ferramentas de simulação.
- *Linker Script*: arquivo de configuração da memória, para automatizar a organização de dados e processos da aplicação, em tempo de compilação.

Para avaliar o desempenho do sistema foram utilizadas aplicações que possibilitassem o máximo de paralelismo e exigissem o funcionamento máximo de todos os recursos. Dessa forma, foram selecionadas duas aplicações com características peculiares, Multiplicação de Matrizes e Algoritmo *Smith-Waterman*, descrito em (ZHANG; QIAO; LIU, 2002).

Uma multiplicação de matrizes é completamente paralelizável e o seu paralelismo é regular, isto é, não se altera em tempo de execução. Uma vez determinado como as linhas da primeira matriz serão multiplicadas pelas colunas da segunda, os núcleos executarão a mesma sequência de *threads* durante toda a execução da aplicação. Enquanto que em aplicações com paralelismo irregular, a ordem em que as *threads* paralelas são executadas muda durante a aplicação.

Essas aplicações são distintas no que diz respeito ao paralelismo. Na multiplicação de matrizes o cálculo de cada elemento da matriz resultante é completamente independente do cálculo dos demais elementos. O algoritmo de Smith-Waterman, assim como a multiplicação de matrizes, também calcula uma matriz de similaridades. Entretanto, o cálculo do elemento H(i,j) depende do cálculo dos elementos H(i-1,j), H(i,j-1) e H(i-1, j-1), isto é, diferentemente da multiplicação de matrizes, em *Smith-Waterman* há dependência uma dependência de dados entre os cálculos de alguns elementos da matriz.

Esta característica faz com que os elementos de uma diagonal da matriz de similaridades possam ser calculados em paralelo. Na implementação aqui apresentada cada núcleo escravo NEi ($0 \leq i \leq 3$) ficou responsável pelo cálculo dos elementos de quatro linhas da matriz (NEi calcula as linhas $i + 4j$, com j variando de 0 a 3).

Nas simulações realizadas foram implementadas matrizes de tamanho 16x16. Isto é, na multiplicação de matrizes, foram multiplicadas duas matrizes quadradas, com dimensões iguais a 16x16, e as mesmas foram inicializadas com uma sequência de números de 1 a 256. Para os experimentos com o algoritmo *Smith-Waterman*, como o objetivo do mesmo é o alinhamento de cadeias genéticas, foram utilizadas duas cadeias com 16 nucleotídeos cada, dessa forma, a matriz de similaridades possui 256 células a serem processadas.

Com matrizes formadas por 16 linhas e 16 colunas, é possível distribuir a carga de trabalho igualmente entre os núcleos escravos e provocar a realocação, a troca de contexto, isto é, a necessidade de alocar nova tarefa para cada um dos núcleos, para avaliar quanto tempo é gasto neste processo, e se isso configura um gargalo.

Na Figura 31 e na Figura 32, os gráficos demonstram a execução das aplicações: os núcleos escravos começam inoperantes até que recebam a primeira tarefa; o núcleo mestre ordena uma execução assíncrona e em seguida alterna entre tratamentos de interrupções e períodos de ociosidade; um dos núcleos escravos executa três *threads* (a inicial da aplicação, uma paralelamente aos demais escravos, e a *thread* de finalização da aplicação).

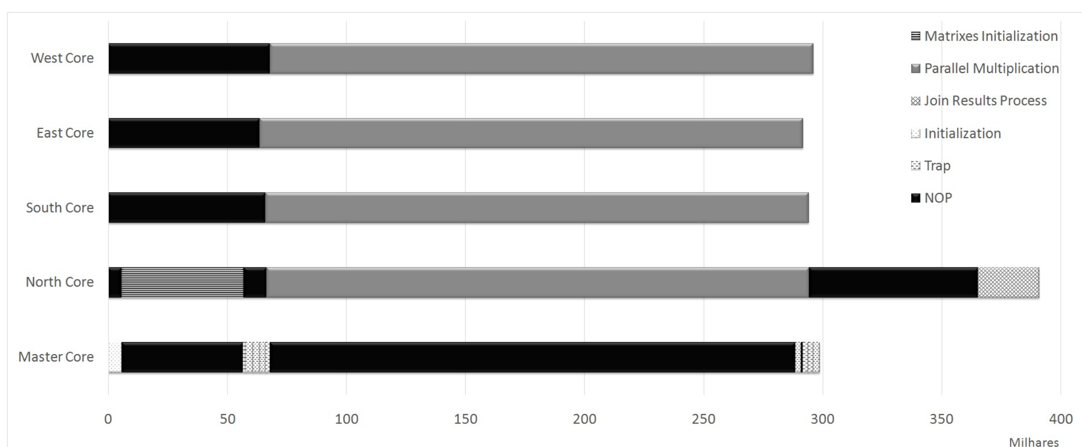

Figura 31 – Gráfico da simulação de multiplicação de matrizes.

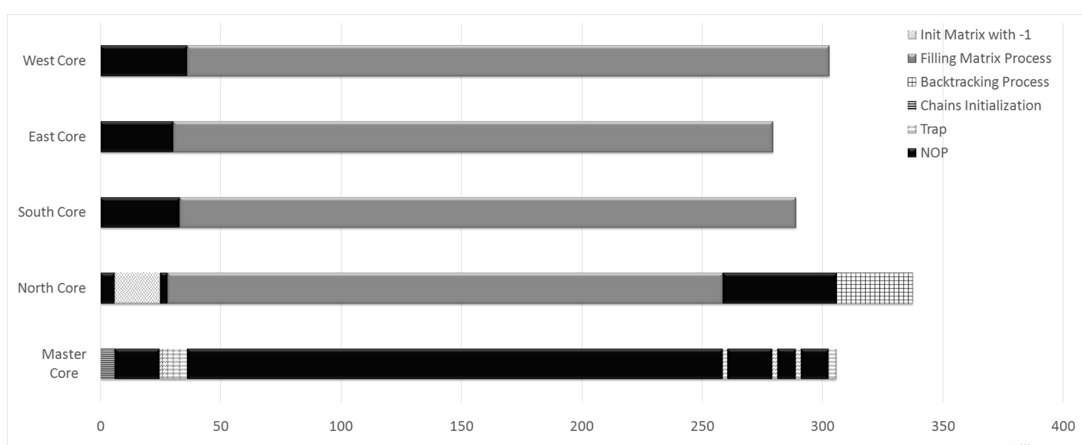

Figura 32 – Gráfico da simulação de Smith-Walterman.

Conclusões e Trabalhos Futuros

Conclusões

Arquiteturas paralelas têm sido o foco das pesquisas científicas, com as mais diversas configurações de memória, microarquiteturas e estruturas de comunicação. Porém, os objetivos permanecem sendo o melhor desempenho, com menor consumo de energia e a maior escalabilidade possível.

A obtenção de um desses objetivos implica em distanciar-se do alcance dos outros. Por exemplo, o melhor desempenho, tem sido obtido por meio da exploração de paralelismo, em níveis cada vez maiores, por meio da inserção de cada vez mais unidades de processamento no sistema. Isso implica em maior espaço ocupado em *chip*, que consequentemente, implica em maior consumo de energia. Devido a isso, pode-se perceber que processadores *manycore*, apresentam bom desempenho quando utilizados para certas aplicações, como algoritmos de criptografia ou algoritmos de processamento de imagens, mas para outras, tais arquiteturas não apresentam desempenho satisfatório.

ArachNoc, é um processador *manycore* com nós de processamento *multicore*, que confere suporte à programação paralela baseada no compartilhamento de variáveis e troca de mensagens (*Send* e *Receive*). Em *ArachNoc*, os nós de processamento são conectados por meio de uma rede de interconexão chamada SoCIN (ZEFERINO; SUSIN, 2003).

Em um nó *ArachNoc*, a comunicação entre os processos ocorre por meio do compartilhamento de variáveis, mas a comunicação entre os nós da rede dar-se-á pela troca de mensagens, através dos roteadores.

Mediante os resultados obtidos, *ArachNoc* apresenta um bom desempenho para execução de aplicações de propósito geral. Faz-se necessário apenas que, a aplicação seja paralelizada segundo a metodologia de programação paralela de *ArachNoc*. Isto é, seguindo o padrão de programação paralela *fork-join* e utilizando o modelo de programação que conta com as instruções (*Exec, Sync, SynExec, Send, Receive*).

Os experimentos demonstram que o sistema de comunicação e sincronização apresentou-se bastante eficiente, provocando pouco atraso no tempo de execução das aplicações, e não apresentou nenhum erro de sincronização entre os processos. O sistema de distribuição de tarefas proporcionou atrasos pequenos nos tempos de simulação e foi eficiente para manter o equilíbrio de carga de trabalho no sistema como um todo.

Portanto, *ArachNoc* é um *manycore* de propósito geral, com modelo e metodologia de programação paralela, de alto desempenho. *ArachNoc* não apenas é um modelo

funcional, possui uma plataforma escalável e flexível que permite a construção de novos multiprocessadores.

Trabalhos Futuros

ArachNoc possui uma memória compartilhada, interna aos nós escravos, que possui arbitragem baseada em *round-robin*. Essa memória, é constituída de 64 blocos de memória de 8 KB cada.

Os blocos de memória podem ser aprimorados a ponto de permitir dois ou mais acessos simultâneos à memória, de modo a evitar gargalos no sistema. Além da quantidade de blocos de memória poder ser aumentada, a capacidade de armazenamento deles também pode ser aumentada.

Existem diversas possibilidades de pesquisa fundamentadas em *ArachNoc*. As memórias dos nós, tanto do mestre quanto dos escravos, devem vir a se tornar caches. Para a implementação de memórias cache, será necessário atentar para o tamanho dos pacotes transferidos através da rede, por meio das instruções *Send* e *Receive*. Isso exige também, a implementação de técnicas para manter a coerência entre as caches dos diversos nós de processamento e do nó mestre.

Ainda a nível de *hardware*, *ArachNoc* é escalável. Atualmente, o sistema é homogêneo, possui todos os núcleos de processamento com a mesma ISA. Porém, novos nós de processamento podem ser adicionados, a fim de satisfazer diferentes tipos de aplicações. Além disso, a própria rede de interconexão pode ser substituída, ou mesmo ter seus barramentos aprimorados para poder transmitir pacotes maiores, em menor tempo, por exemplo.

No que diz respeito ao gerenciamento do sistema, *ArachNoc* necessita de um sistema operacional para coordenar as trocas de contexto, a alocação dos recursos, o gerenciamento do descritor de aplicações para controlar as interrupções e exceções, e controlar dispositivos de entrada e saída.

ArachNoc confere suporte à programação paralela híbrida, e dá suporte também a sistema operacional. Com isso, *ArachNoc* possibilita a exploração de uma grande quantidade de linhas de pesquisas, tanto a nível de *hardware* quanto a nível de *software*.

Referências

APPLE, I. *Concurrency programming guide*. http:// developer.apple.com/mac/library/documentation/ General/Conceptual/ConcurrencyProgrammingGuide/ ConcurrencyandApplicationDesign/ ConcurrencyandApplicationDesign.html., 2009. Citado na página 23.

ARAUJO, S. R. F. d. *Projeto de Sistemas Integrados de Propósito Geral Baseados em Redes em Chip – Expandindo as Funcionalidades dos Roteadores para Execução de Operações: A plataforma IPNoSys*. Tese (Doutorado) — Universidade Federal do Rio Grande do Norte, Março 2012. Citado 5 vezes nas páginas 2, 34, 35, 36 e 58.

ASPRAY, W. The intel 4004 microprocessor: What constituted invention? *IEEE Ann. Hist. Comput.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 19, n. 3, p. 4–15, jul. 1997. ISSN 1058-6180. Disponível em: <http://dx.doi.org/10.1109/85.601727>. Citado na página 1.

BENINI, L.; MICHELI, G. D. Networks on chips: a new soc paradigm. *Computer*, v. 35, n. 1, p. 70–78, Jan 2002. ISSN 0018-9162. Citado na página 1.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. *IEEE Signal Processing Magazine*, IEEE, v. 26, n. 6, p. 26–37, nov. 2009. ISSN 1053-5888. Disponível em: <http://dx.doi.org/10.1109/msp.2009.934110>. Citado na página 9.

BOUKNIGHT, W. J.; DENENBERG, S. A.; MCINTYRE, D. E.; RANDALL, J. M.; SAMEH, A. H.; SLOTNICK, D. L. The illiac iv system. *Proceedings of the IEEE*, v. 60, n. 4, p. 369–388, April 1972. ISSN 0018-9219. Citado na página 1.

BRANOVER, A.; FOLEY, D.; STEINMAN, M. Amd fusion apu: Llano. *Micro, IEEE*, v. 32, n. 2, p. 28–37, 2012. ISSN 0272-1732. Citado na página 15.

BUTENHOF, D. R. *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2. Citado na página 14.

BUTENHOF, D. R. *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2. Citado na página 23.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. [S.l.]: The MIT Press, 2007. ISBN 0262533022, 9780262533027. Citado na página 23.

CHEN, Q.-k.; ZHANG, J.-k. A stream processor cluster architecture model with the hybrid technology of mpi and cuda. In: *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICISE '09), p. 86–89. ISBN 978-0-7695-3887-7. Disponível em: <http://dx.doi.org/10.1109/ICISE.2009.171>. Citado na página 15.

DAGUM, L.; MENON, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 46–55, jan. 1998. ISSN 1070-9924. Disponível em: <http://dx.doi.org/10.1109/99.660313>. Citado na página 14.

DIAZ, J.; MUNOZ-CARO, C.; NINO, A. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, v. 23, n. 8, p. 1369–1386, Aug 2012. ISSN 1045-9219. Citado 7 vezes nas páginas 13, 15, 21, 22, 27, 36 e 55.

DOWDECK, J. *Inside Intel Core Microarchitecture and Smart Memory Access*. [S.l.], 2006. Disponível em: <https://software.intel.com/sites/default/files/m/d/4/1/d/8/sma.pdf>. Citado na página 28.

DRUCK, G. A nostalgia do fordismo: modernização e crise na teoria da sociedade salarial. *Revista Brasileira de Ciências Sociais*, scielo, v. 20, p. 180 – 185, 02 2005. ISSN 0102-6909. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0102-69092005000100011&nrm=iso>. Citado na página 9.

ÉTIENNE, E. Y. *Hyper-threading*. [S.l.]: TurbsPublishing, 2012. ISBN 6135655345, 9786135655346. Citado na página 29.

FERREIRA, J. C. dos S. *CLEM & OCEAN: Dois Compiladores OpenCL para as Arquiteturas Multicore METAL e ArachNoC*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DO PIAUÍ, 2016. Citado na página 55.

FLYNN, M. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21, n. 9, p. 948–960, Sept 1972. ISSN 0018-9340. Citado na página 16.

GIRAO, G. B. d. S. *Estudo sobre o Impacto da Hierarquia de Memória em MPSoCs baseados em NoC*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul - UFRGS, 2009. Citado 2 vezes nas páginas 11 e 12.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728. Citado 4 vezes nas páginas 7, 9, 10 e 40.

IBM. *IBM System/360 Principles of Operation*. [S.l.]: IBM Press, 1964. Citado na página 10.

JOHNSON, P. Pthread performance in an mpi model for prime number generation. In: *CSCI 4576 - High-Performance Scientific Computing*. Univesidade do Colorado: [s.n.], 2007. Citado na página 15.

KAWAKAMI, Y.; ISHIZUKA, H.; WATARI, M.; SAKOE, H.; HOSHI, T.; IWATA, T. A microprocessor for speech recognition. *IEEE Journal on Selected Areas in Communications*, v. 3, n. 2, p. 369–376, March 1985. ISSN 0733-8716. Citado na página 32.

KENNEDY, K.; KOELBEL, C.; ZIMA, H. The rise and fall of high performance fortran: An historical object lesson. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, USA: ACM, 2007. (HOPL III), p. 7–1–7–22. ISBN 978-1-59593-766-7. Disponível em: <http://doi.acm.org/10.1145/1238844.1238851>. Citado na página 13.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. The influence of parallel programmming interfaces on multicore embedded systems. In: *IEEE COMPSAC 2015: The 39th Annual International Computers, Software and Applications Conference*. [S.l.: s.n.], 2015. p. 617–625. Citado 2 vezes nas páginas 25 e 26.

MANCHANDA, N.; ANAND, K. *New York*, April 2012. Citado na página 3.

MCKEE, S. A. Reflections on the memory wall. In: *Proceedings of the 1st Conference on Computing Frontiers*. New York, NY, USA: ACM, 2004. (CF '04), p. 162–. ISBN 1-58113-741-9. Disponível em: <http://doi.acm.org/10.1145/977091.977115>. Citado na página 3.

MOLKA, D.; HACKENBERG, D.; SCHöNE, R. Main memory and cache performance of intel sandy bridge and amd bulldozer. In: *Proceedings of the Workshop on Memory Systems Performance and Correctness*. New York, NY, USA: ACM, 2014. (MSPC '14), p. 4:1–4:10. ISBN 978-1-4503-2917-0. Disponível em: <http://doi.acm.org/10.1145/2618128.2618129>. Citado na página 15.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, April 1965. Citado na página 8.

NEUMANN, J. von. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 15, n. 4, p. 27–75, out. 1993. ISSN 1058-6180. Disponível em: <http://dx.doi.org/10.1109/85.238389>. Citado na página 7.

PACHECO, P. S. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN 1-55860-339-5. Citado na página 22.

PEREIRA, A. L. V. *Projeto e Implementação de um MPSoC Utilizando a IPNoSys como Unidade de Processamento*. Dissertação (Mestrado) — Universidade Federal Rural do Semi-Árido - UFERSA, 2014. Citado na página 36.

REGO, R. S. de L. S. *Projeto e Implementação de uma Plataforma MPSoC usando SystemC*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE, 2006. Citado na página 33.

SANCHEZ, L.; FERNANDEZ, F.; SOTOMAYOR, R.; GARCIA, J. A comparative evaluation of parallel programming models for shared-memory architectures. In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. [S.l.: s.n.], 2012. p. 363–370. Citado 7 vezes nas páginas 13, 14, 17, 24, 29, 30 e 55.

SCHAUER, B. *Multicore Processors - A necessity*. [S.l.], 2008. Citado 4 vezes nas páginas 2, 9, 28 e 29.

SCHELLE, G.; COLLINS, J.; SCHUCHMAN, E.; WANG, P.; ZOU, X.; CHINYA, G.; PLATE, R.; MATTNER, T.; OLBRICH, F.; HAMMARLUND, P.; SINGHAL, R.; BRAYTON, J.; STEIBL, S.; WANG, H. Intel nehalem processor core made fpga synthesizable. In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2010. (FPGA Í10), p. 3–12. ISBN 978-1-60558-911-4. Disponível em: <http://doi.acm.org/10.1145/1723112.1723116>. Citado na página 29.

SHEKHAR, T. D.; VARAGANTI, K.; SURESH, R.; GARG, R.; RAMAMOORTHY, R. Comparison of parallel programming models for multicore architectures. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. [S.l.: s.n.], 2011. p. 1675–1682. ISSN 1530-2075. Citado 4 vezes nas páginas 17, 21, 23 e 55.

SMITH, J. E.; SOHI, G. S. *The Microarchitecture of Superscalar Processors*. 1995. Citado na página 9.

SMITH, L.; BULL, M. Development of mixed mode mpi / openmp applications. *Sci. Program.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 9, n. 2,3, p. 83–98, ago. 2001. ISSN 1058-9244. Disponível em: <http://dl.acm.org/citation.cfm?id=1239928.1239936>. Citado na página 15.

SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. *MPI-The Complete Reference, Volume 1: The MPI Core*. 2nd. (revised). ed. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262692155. Citado na página 15.

SOARES, T. R. B. S.; SILVA, I. S.; FERNANDES, S. R. Ipnosys ii: A new architecture for ipnosys programming model. In: *Proceedings of the 28th Symposium on Integrated Circuits and Systems Design*. New York, NY, USA: ACM, 2015. (SBCCI '15), p. 40:1–40:7. ISBN 978-1-4503-3763-2. Disponível em: <http://doi.acm.org/10.1145/2800986.2801012>. Citado na página 36.

STERLING, T.; MESSINA, P.; SMITH, P. H. *Enabling Technologies for Petaflops Computing*. Cambridge, MA, USA: MIT Press, 1995. ISBN 0-262-69176-0. Citado na página 15.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 3, p. 66–73, maio 2010. ISSN 0740-7475. Disponível em: <http://dx.doi.org/10.1109/MCSE.2010.69>. Citado na página 13.

WANG, Y.; FENG, Z.; GUO, H.; HE, C.; YANG, Y. Scene recognition acceleration using cuda and openmp. In: *2009 1st International Conference on Information Science and Engineering (ICISE)*. [S.l.: s.n.], 2009. p. 1422–1425. Citado na página 15.

WIKIPEDIA - Cross Compiler. [S.l.]. Disponível em: <https://en.wikipedia.org/wiki/Cross_compiler>. Citado na página 89.

WOLF, W.; JERRAYA, A.; MARTIN, G. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 27, n. 10, p. 1701–1713, Oct 2008. ISSN 0278-0070. Citado na página 1.

ZEFERINO, C.; SUSIN, A. Socin: a parametric and scalable network-on-chip. In: *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.* [S.l.: s.n.], 2003. p. 169–174. Citado 4 vezes nas páginas 2, 41, 42 e 71.

ZEFERINO, C. A. *Redes-em-Chip: arquiteturas e modelos para avaliação de área e desempenho*. Tese (Doutorado) — UNIVERSIDADE FEDERAL DE RIO GRANDE DO SUL, 2003. Citado na página 1.

ZHANG, F.; QIAO, X.-Z.; LIU, Z.-Y. A parallel smith-waterman algorithm based on divide and conquer. In: *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on.* [S.l.: s.n.], 2002. p. 162–169. Citado na página 67.

Apêndices

APÊNDICE A – Multiplicação de matrizes em C

Para a multiplicação de matrizes paralela utilizando a linguagem C, é necessário escrever códigos-fonte paralelos, isto é, dividir a aplicação em processos que serão executados em paralelo, e para fazer isso sem o uso de *frameworks* ou bibliotecas, utiliza-se o padrão de paralelismo *fork/join* 24.

Por conta da metodologia de programação desenvolvida, o programador descreve cada uma das *threads* da aplicação em um arquivo “.c”, conforme o código utilizado para a multiplicação de matrizes de dimensões 16x16, abaixo:

Processo inicializador:

```
#include "matrix.h"

int main() __attribute__((section(".block30")));
void enviaSynExec() __attribute__((section(".block30")));
void enviaExec() __attribute__((section(".block30")));
void identity(int a[16][16], int offset) __attribute__((section(".block30


int main() {
        identity(a, 0);
        identity(b, 0);

        enviaSynExec();
        enviaExec();
        enviaExec();
        enviaExec();
        enviaExec();
}

void enviaSynExec() {}
void enviaExec() {}

void identity(int a[16][16], int offset) {
    int i;
```

```
    int j;

    for (i = 0; i < 16; ++i) {
        for (j = 0; j < 16; ++j) {
            a[i][j] = i + offset == j;
        }
    }
}
```

No código, as marcações “___attribute___((section(".block30")))” servem para estabelecer a comunicação entre compilador e *linker script*, informando ao compilador onde tal variável ou função deve ser armazenada na memória, de acordo com o descrito no *linker script*. No caso, o *linker script* identifica uma região de memória como “.block30” e o compilador irá armazenar todas as variáveis e funções com essa marcação nessa região de memória determinada.

A função “identity” inicializa uma matriz com uma sequência de números. A função “main”, ordena a inicialização das matrizes que serão multiplicadas e envia um sinal de **enviaSynExec** e ordena a execução das multiplicações paralelas, **enviaExec**. Tais funções são vazias em suas implementações, pois não pertencem a ISA MIPS, isto é, não são reconhecidas pelo compilador e portanto, o compilador cruzado não consegue gerar código *assembly* para as mesmas. Devido a isso, faz-se apenas a implementação vazia e após a geração de código, adiciona-se o código *assembly* das mesmas.

Processos de multiplicações paralelas:

```
#include "matrix.h"

void mult1(int a[16][16], int b[16][16], int c[16][16])
_attribute__((section(".block31")));
void enviaSync() __attribute__((section(".block31")));

void mult1(int a[16][16], int b[16][16], int c[16][16]) {
    int i;
    int j;
    int k;
    int t;

    for (i = 0; i < 16; ++4) {
        for (j = 0; j < 16; ++j) {
            t = 0;
```

```
            for (k = 0; k < 16; ++k) {
                t += a[i][k] * b[k][j];
            }

            c[i][j] = t;
        }
    }

    enviaSync();
}


void enviaSync(){}
```

```
 #include "matrix.h"

void mult2(int a[16][16], int b[16][16], int c[16][16])
attribute__((section(".block32")));
void enviaSync() __attribute__((section(".block32")));

void mult2(int a[16][16], int b[16][16], int c[16][16]) {
    int i;
    int j;
    int k;
    int t;
    for (i = 1; i < 16; ++4) {
        for (j = 0; j < 16; ++j) {
            t = 0;

            for (k = 0; k < 16; ++k) {
                t += a[i][k] * b[k][j];
            }

            c[i][j] = t;
        }
    }

    enviaSync();
}
```

```
void enviaSync(){}

 #include "matrix.h"

void mult3(int a[16][16], int b[16][16], int c[16][16])
attribute__((section(".block33")));
void enviaSync() __attribute__((section(".block33")));

void mult3(int a[16][16], int b[16][16], int c[16][16]) {
    int i;
    int j;
    int k;
    int t;

    for (i = 2; i < 16; ++4) {
        for (j = 0; j < 16; ++j) {
            t = 0;

            for (k = 0; k < 16; ++k) {
                t += a[i][k] * b[k][j];
            }

            c[i][j] = t;
        }
    }

    enviaSync();
}


void enviaSync(){}

 #include "matrix.h"

void mult4(int a[16][16], int b[16][16], int c[16][16])
attribute__((section(".block34")));
void enviaSync() __attribute__((section(".block34")));

void mult4(int a[16][16], int b[16][16], int c[16][16]) {
```

```
    int i;
    int j;
    int k;
    int t;

    for (i = 3; i < 16; ++4) {
        for (j = 0; j < 16; ++j) {
            t = 0;

            for (k = 0; k < 16; ++k) {
                t += a[i][k] * b[k][j];
            }

            c[i][j] = t;
        }
    }

    enviaSync();
}


void enviaSync(){}
```

Os códigos das multiplicações paralelas são muito semelhantes, porém, para evitar um “gargalo”, durante a tentativa dos processadores em acessar o mesmo bloco de memória para executar tal *thread*, os mesmos são armazenados em diferentes blocos de memória.

É feito para **enviaSync** o mesmo que é feito para **enviaSynExec** e **enviaExec**, apenas uma implementação vazia, para identificá-la no código *assembly* e adicionar-se o *assembly* correspondente.

Processo finalizador:

```
#include "matrix.h"

void junta() __attribute__((section(".block35")));
void juntar(int result[16][16], int a[16][16], int offset) __attribute__(

void juntar(int result[16][16], int a[16][16], int offset) {
    int i;
    int j;
```

```
        for (i = offset; i < 16; ++4) {
                for (j = 0; j < 16; +j) {
                        result[i][j] = a[i][j];
                }
        }
}

void junta() {
        juntar(result, c0, 0);
        juntar(result, c1, 1);
        juntar(result, c2, 2);
        juntar(result, c3, 3);
}
```

O processo finalizador junta os resultados das multiplicações paralelas (c0,c1,c2,c3), em um único bloco de memória.

O mapeamento de todos os dados utilizados na multiplicação de matrizes é feito por meio de uma biblioteca, cujo código segue abaixo:

```
#ifndef MATRIX_H
#define MATRIX_H

#define MAX 16
#define TAM 16
static int a[16][MAX] __attribute__((section(".block7")));
static int b[16][MAX] __attribute__((section(".block14")));

static int c0[4][MAX] __attribute__((section(".block15")));
static int c1[4][MAX] __attribute__((section(".block16")));
static int c2[4][MAX] __attribute__((section(".block17")));
static int c3[4][MAX] __attribute__((section(".block18")));

static int result[MAX][MAX] __attribute__((section(".block19")));

#endif
```

Portanto, cada matriz ficou armazenada em um bloco de memória diferente, simplesmente para facilitar verificações de erro durante os experimentos.

Anexos

ANEXO A – Compilador Cruzado GCC

Um compilador cruzado é um compilador que irá gerar código executável para uma arquitetura diferente da que ele está instalado. Portanto, é uma ferramenta muito útil quando se trata de desenvolvimento de aplicações, pois na grande maioria das vezes o programador utiliza uma arquitetura, geralmente intel x86_64 ou amd, para gerar código executável para outra arquitetura, as mais comuns são: powerpc, arch, mips.

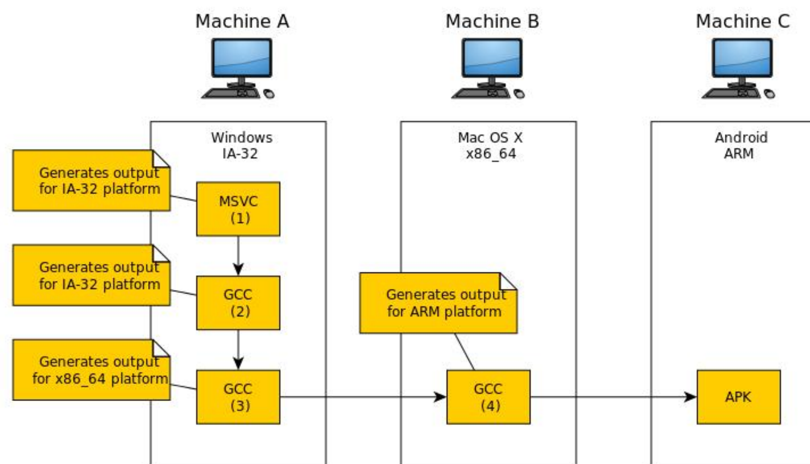

Figura 33 – Compilação Cruzada
Fonte: (WIKIPEDIA. . . ,)

Durante o desenvolvimento de aplicações para a arquitetura MIPS_IPS, o compilador cruzado é de fundamental importância, exatamente porque através dele é possível não só gerar códigos para arquiteturas MIPS, como também linkar, os códigos fonte com um script de descrição de memória (linker script), no qual o programador determina que espaço de memória cada parte do código irá ocupar. Isto é, onde alocar determinada função ou variável.

A opção de utilizar a linguagem C para o desenvolvimento de aplicações para a plataforma MIPS_IPS se deve à universalidade da linguagem, bastante consolidada e conhecida e à propriedade de permitir acessos á espaços de memória e registradores específicos. O programador tem controle sobre alocação e desalocação de espaços de memória, através de ponteiros, e pode manusear registradores através do modificador register. Além das propriedades da linguagem, o GCC (GNU Compiler Collection) permite o desenvolvimento de compiladores cruzados para algumas arquiteturas, dentre elas, mips de 32 e 64 bits, portanto, a configuração de um cross-compiler C to MIPS se torna uma tarefa fácil quando têm-se um compilador C pré-instalado, e o mesmo é de fácil instalação.

Para a instalação de um compilador cruzado são necessários: Alguma versão do

compilador gcc; O pacote binutils de versão compatível com a versão do gcc instalada e; Os pacotes GMP, MPFR e MPC da GNU, nas versões mais recentes (estes pacotes são isntalados juntamente com o gcc). Para verificar as versões compatíveis do gcc e do binutils use as orientações do link: Cross-Compiler Successful Builds.

Deve-se atentar para o ambiente de configuração do compilador cruzado, o mais indicado a se utilizar é ambiente Linux, devido ao fácil acesso aos arquivos binários de cada um dos pacotes e a necessidade de executar instruções shell script. Em caso de utilizar Windows, é necessário configurar um ambiente like-Linux. Ainda, se o sistema operacional utilizado for MAC OS, faz-se necessário a biblioteca libconv, que pode ser adquirida diretamente dos servidores de arquivos GNU. A versão do gcc que poderá ser utilizada depende da distribuição Linux que você usa. Verifique qual a versão gcc compatível com sua distribuição Linux e use o link acima para baixar a versão compatível do binutils.

Este tutorial foi desenvolvido utilizando-se as distribuições e versões: Linux Ubuntu 13.04; gcc 4.7.2 e binutils 2.23.1.

De posse dos pacotes citados anteriormente, a instalação inicia-se por configurar as variáveis de ambiente, através dos comandos:

```
export PREFIX="$HOME/opt/cross"
export TARGET=mips-elf
export PATH="$PREFIX/bin:$PATH"
```

A variável PREFIX deve conter o caminho da pasta na qual serão configurados todos os arquivos binários para a instalação do cross-compiler.

A variável TARGET deve conter o nome-padrão da arquitetura alvo (mips-elf), a arquitetura para a qual deseja desenvolver aplicações. Em seguida, no terceiro comando é feita a adição do caminho da pasta de configuração ao PATH do sistema para a sessão atual do shell. Isto é, se no meio da instalação algo acontecer e o termina se fechar, é necessário refazer os comandos acima citados. Após a preparação do ambiente, compila-se o pacote binutils, por meio dos comandos a seguir:

```
cd $HOME/src
mkdir build-binutils
cd build-binutils
../binutils-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --disable-nls
make
make install
```

O parâmetro **–disable-nls** comunica ao binutils para não incluir suporte á linguagem nativa, o que é opcional, mas reduz dependências e tempo de compilação.

O passo seguinte é instalar o compilador GCC, em um diretório específico, e instalando dentro deste os conteúdos dos pacotes GMP, MPFR e MPC, para realizar a instalação completa do GCC, sem que haja interrupções por falta de dependências. Portanto para configurar e instalar o GCC, siga os seguintes comandos:

```
cd $HOME/src

mv libconv-x.y.z gcc-x.y.z/libconv #MAC OS X users
mv gmp-x.y.z gcc-x.y.z/gmp
mv mpfr-x.y.z gcc-x.y.z/mpfr
mv mpc-x.y.z gcc-x.y.z/mpc

mkdir build-gcc
cd build-gcc

../gcc-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --disable-nls enable-l

make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

O parâmetro *–without-headers* comunica ao GCC para não importar nenhuma biblioteca padrão do C. *–enable-languagens* comunica ao GCC para compilar, dentre as linguagens que o mesmo suporta, apenas aquelas linguagens passadas como parâmetro.

Uma vez que a instalação concluiu com sucesso, para verificar a execução do cross-compiler basta requisitar a versão do compilador, através do seguinte comando:

```
\$TARGET-gcc --version
```

E deverá ser exibido uma mensagem com o nome da arquitetura alvo seguido de "-gcc" com o número de versão e algumas outras descrições padrões a respeito do cross-compiler.

ANEXO B – *Linker Script*

Para desenvolver aplicações utilizando o compilador cruzado é necessário descrever a organização da memória da arquitetura alvo. Isto é, o compilador precisa saber qual o tamanho do espaço de endereçamento, em que região guardar processos/*threads* e em que região guardar variáveis ou constantes, dados em geral. Para definir esses parâmetros é que serve o *linker script*. Um exemplo de código é mostrado abaixo:

```
 MEMORY
{
      TEXT (RX) : ORIGIN = 0, LENGTH = 4K
      GLOBAL_MEM (RX) : ORIGIN = 7168, LENGTH = 8K

      A_MATRIX(LW) : ORIGIN = 1024, LENGTH = 1K
      B_MATRIX(LW) : ORIGIN = 2048, LENGTH = 1K
      C_MATRIX (LW) : ORIGIN = 3072, LENGTH = 4K
      BLOCO (RX) : ORIGIN = 7168, LENGTH = 4K
}

SECTIONS {
/* Codigo executavel, deve ser copiado diretamente para a memoria */
.text :
{
    *(.entrada)
    *(.text)
    *(.init)
    *(.fini)
} > TEXT



.block :
{
    *(.block)
} > BLOCO

.cmatrix :
{
    *(.cmatrix)
```

```
} > C_MATRIX

.amatrix :
{
    *(.amatrix)
} > A_MATRIX

.bmatrix :
{
    *(.bmatrix)
} > B_MATRIX

/* Dados inicializados, devem ser copiados da forma que estao direto para a memoria */
.data :
{
    *(.data)
    *(.rodata)
    *(.global_)
} > GLOBAL_MEM

/* Dados nao inicializados, devem ser zerados durante a carga na memoria */
.bss :
{
    *(.bss)
    *(COMMON)
    *(.scommon)
} > GLOBAL_MEM

.tbr :
{
    *(.tbr)
} > GLOBAL_MEM

.lixo :
{
    *(.eh_frame)
    *(.ctors)
    *(.dtors)
    *(.jcr)
```

```
} > GLOBAL_MEM
}
```

Um *linker script* possui duas marações fundamentais: *Memory* e *Sections*. O parâmetro *MEMORY* informa ao compilador o tamanho da memória, que comumente é particionado como no exemplo acima, pois particionar permite definir regiões específicas para código e para dados. No caso, **TEXT** é uma região da memória onde é permitido apenas leitura e execução "(RX)", ou seja, a mesma é inicializada em tempo de compilação. Por outro lado, regiões como **A_MATRIX**, que iniciam no endereço 1024 (ORIGIN) e possuem o tamanho de 1KB (LENGTH), são regiões inicializadas em tempo de execução (L) e nas quais é permitido leitura e escrita (W).

O parâmetro *SECTIONS* define que informações serão guardadas em cada uma das regiões definidas em *MEMORY*. Por exemplo, no código acima, tudo no código-fonte que tiver a marcação ".cmatrix" será armazenado no bloco **C_MATRIX**, desde que **C_MATRIX** tenha espaço suficiente, caso contrário o dado continua sendo armazenado nos endereços seguintes, invadindo por exemplo a região **BLOCO**. Enquanto que, tudo que for marcado com ".data" ou ".rodata" ou ainda, ".global_" será armazenado na região da memória que foi denominada **GLOBAL_MEM**.

Com a implementação do *linker script* o programador é capaz de controlar toda a movimentação de dados na memória, para fins de experimentos e correções de erros. Isto é, o programador tem controle sobre o que é aramazenado na memória e seus endereços, o que facilita a busca por *bugs* e erros de implementação, seja em *sofware* ou *hardware*.