UNIVERSIDADE FEDERAL DO PAMPA

Geaninne Marchezan do Nascimento Lopes

Investigando Paradigmas de Programação Paralela sobre o MPSoC HeMPS

Alegrete
2018

Geaninne Marchezan do Nascimento Lopes

Investigando Paradigmas de Programação Paralela sobre o MPSoC HeMPS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Profª. Drª. Aline Vieira de Mello

Alegrete
2018

Geaninne Marchezan do Nascimento Lopes

Investigando Paradigmas de Programação Paralela sobre o MPSoC HeMPS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 05 de dezembro de 2018
Banca examinadora:

Aline Vieira de Mello

Profª. Drª. Aline Vieira de Mello
Orientador
UNIPAMPA

Prof. Dr. Arthur Francisco Lorenzon
UNIPAMPA

Claudio Schepke

Prof. Dr. Claudio Schepke
UNIPAMPA

AGRADECIMENTOS

Não poderia deixar de expressar a minha gratidão a todos que foram essenciais para a realização deste Trabalho de Conclusão de Curso, e na minha trajetória acadêmica.

Primeiramente, agradeço a Deus por sempre iluminar meu caminho.

À Universidade Federal do Pampa, pela educação de qualidade, e pelo ambiente agradável que proporciona. Aos funcionários da instituição que contribuem para a sua qualidade. Aos professores, pelos ensinamentos, sabedoria, e conselhos transmitidos. Aos colegas de curso pela boa convivência, e auxílio e colaboração na realização deste e outros trabalhos.

Aos professores Dr. Cláudio Schepke e Dr. Arthur Francisco Lorenzon que aceitaram contribuir com este trabalho, fazendo parte da banca de avaliação.

Ao professor Ewerson Carvalho, que foi orientador do projeto deste TCC, e sempre me auxiliou da melhor forma possível.
À minha professora orientadora, Dra. Aline Vieira de Mello, que desde o meu primeiro dia de aula na Ciência da Computação (Disciplina de Arquitetura de Computadores II), se tornou uma inspiração para mim. Agradeço por ter me introduzido na iniciação científica e por todo o comprometimento, dedicação, e paciência que teve na realização deste e de outros trabalhos. Agradeço por ser essa orientadora incrível, que mesmo nos momentos mais difíceis me motivou a seguir em frente e me ajudou da melhor forma possível.

Aos amigos que, durante a minha trajetória acadêmica, entenderam minhas ausências, suportaram minha angústia, e me ofereceram apoio e amizade verdadeira.

Ao meu namorado, Roberto Tubino, por todo o suporte, amor, carinho e paciência.

Agradeço à minha família, que acreditou em mim e sempre esteve do meu lado. Aos meus irmãos, Graciane, Leandro e Lucas, pelo carinho, amizade e conselhos. Aos meus pais Lereci Marchezan e Luiz Adão do Nascimento Lopes, que sempre torceram por mim, e estiveram presentes me oferecendo suporte, incentivo e amor incondicional.

RESUMO

Um Sistema Multiprocessado no Chip MPSoC é um sistema composto por vários elementos de processamento (PEs). Trata-se de uma tecnologia que permite a execução de múltiplas tarefas em paralelo, tal como computadores *multicore*. Entretanto, diferentemente desses computadores com poucos núcleos, um MPSoC tende a possuir centenas de PEs. Geralmente a comunicação entre os PEs de um MPSoC é implementada via mensagens, as quais são transmitidas através dos canais de uma Rede Intra-Chip (NoC). Devido a semelhança estrutural, acredita-se que os mesmos modelos empregados na programação de máquinas paralelas (*clusters* e Unidades de Processamento Gráfico (GPUs)) podem ser empregados em MPSoCs. Dentre eles, pode-se citar: Mestre-Escravo; *Pipeline*; Fases Paralelas; e Divisão e Conquista. Neste contexto, o presente trabalho tem como proposta investigar o emprego de paradigmas de programação paralela no desenvolvimento de aplicações para MPSoCs. Para este propósito, foram implementadas aplicações Multiplicação de Matrizes, Manipulação de Imagens e Padrão de Criptografia Avançada (AES) nos paradigmas Mestre-Escravo, *Pipeline* e Divisão e Conquista. Adicionalmente, foi proposta uma ferramenta, denominada HeMPS *Parallel Programming*, com o objetivo de auxiliar o programador na paralelização das aplicações. Dois critérios foram considerados para a comparação entre os paradigmas: tempo de execução e consumo de energia dos processadores. Os resultados obtidos permitiram concluir que *Pipeline* foi o paradigma que apresentou melhor desempenho e menor consumo de energia para a aplicação Manipulação de Imagens. No entanto, seus resultados foram piores do que os obtidos pela versão sequencial. Assim, não justificando a adoção de paralelismo para o tamanho de imagens simuladas. Já o paradigma Mestre-Escravo apresentou melhor desempenho e menor consumo de energia para as aplicações Multiplicação de matrizes e AES para os tamanhos de entrada simulados. No entanto, o paradigma Divisão e Conquista tende a ter melhores resultados de tempo de execução para as aplicações Multiplicação de matrizes e AES quando o tamanho da entrada aumentar.

Palavras-chave: MPSoC, Paradigmas de Programação Paralela, Programação Paralela.

ABSTRACT

A Multiprocessor System on the Chip MPSoC is a system composed of several processing elements PEs. It is a technology that allows the execution of multiple tasks in parallel, such as multicore computers. However, unlike those computers with few cores, a MPSoC tends to have hundreds of PEs. Generally the communication between the PEs of an MPSoC is implemented via messages, which are transmitted through the channels of an Intra-Chip Network (NoC). Due to the structural similarity, it is believed that the same models used in the programming of parallel machines (*clusters* and Graphics Processing Units (GPUs)) can be used in MPSoCs. Among them, we can mention: Master-Slave; Pipeline; Parallel Phases; and Divide and Conquer. In this context, the present work aims to investigate the use of parallel programming paradigms in the development of applications for MPSoCs. For this purpose, Matrix Multiplication, AES, and Image Manipulation applications were implemented in the Master-Slave, Pipeline and Divide and Conquer paradigms. In addition, a tool, called HeMPS Parallel Programming, was proposed with the objective of assisting the programmer in the parallelization of applications. Two criteria were considered for the comparison between the paradigms: execution time and consumption of energized processors. The results obtained allowed to conclude that Pipeline was the paradigm that presented better performance and lower power consumption for the Image Manipulation application. However, their results were worse than those obtained by the sequential version. Thus, not justifying the adoption of parallelism for the size of simulated images. Already the Master-Slave paradigm presented better performance and lower energy consumption for the Matrix Multiplication and AES applications for the simulated input domains. However, the Divide and Conquer paradigm tends to have better run-time results for Matrix Multiplication and AES applications when the input size increases.

Key-words: MPSoC, Parallel Programming Paradigms, and Parallel Programming.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AES Advanced Encryption Standard

API *Application Programming Interface*

CI Circuito Integrado

CM *Cluster Manager*

DMA Direct Memory Access

DMNI Direct Memory Network Interface

GPU Graphics Processing Units

HeMPS Hermes Multiprocessor System

IP Intellectual Property Core

MIPS Microprocessor without interlocked pipeline stages

MPI Message Passing Interface

MPSoC Multi-processor System-on-Chip

NIST National Institute of Standards and Technology

NoC Network-on-Chip

PE *Processing Element*

RGB Red - Green - Blue

RISC *Reduced Instruction Set Computer*

SO Sistema Operacional

SoC *System-on-Chip*

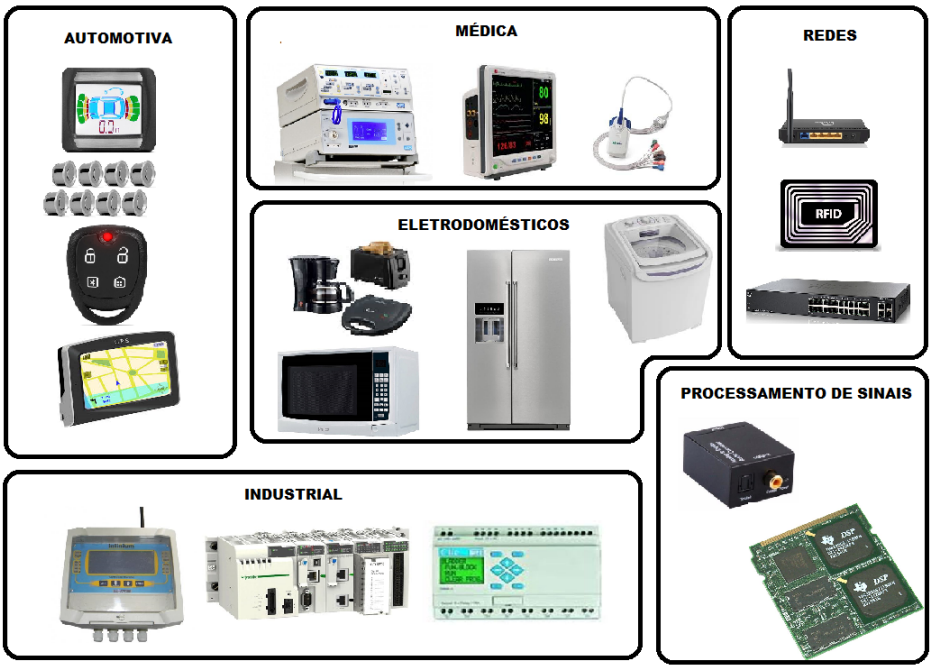
SP Slave Processor

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados são sistemas computacionais projetados para exercer funções específicas dentro de um sistema maior (MÜCK et al., 2013). Segundo Oliveira e Andrade (2006), também podem ser definidos como sistemas de processamento digital que possuem seu *software* de controle armazenado em uma memória presente no circuito eletrônico. Estes sistemas podem ser encontrados em equipamentos das mais diversas áreas (médica, industrial, redes, etc.) como exemplificado na Figura 1. O que diferencia este conjunto de dispositivos de um computador convencional é o seu projeto baseado em um conjunto dedicado e especialista constituído por *hardware*, *software* e periféricos (REIS, 2004).

Figura 1 – Exemplos de Sistemas Embarcados.



Fonte: Próprio Autor.

Há algum tempo atrás estes sistemas deveriam ser implementados em Circuitos Integrados (CIs) separados e soldados em uma placa de circuito impresso. No entanto, a complexidade crescente das aplicações resultou na necessidade de recursos que fossem mais baratos e acelerassem o processamento das aplicações. Em decorrência disso e da evolução tecnológica, surgiram os sistemas integrados em um único chip (SoC - do inglês *System-on-Chip*). SoC é conceituado por Jerraya e Wolf (2005) como um CI que implementa a maioria ou todas as funções de um sistema eletrônico completo. De acordo com Martin e Chang (2012), estes sistemas baseiam-se no reuso de blocos de *hardware*, os quais são chamados de blocos de Propriedade Intelectual (IP - do inglês *Intellectual Property*).

O avanço da tecnologia permitiu inserir mais unidades de processamento ao mesmo CI, formando um Sistema Multiprocessador no Chip (MPSoC - do inglês *Multiprocessor System-on-Chip*). Jerraya e Wolf (2005) definem MPSoCs como sistemas multiprocessados implementados na forma de um SoC. Um MPSoC consiste de uma arquitetura composta por recursos heterogêneos, incluindo múltiplos processadores embarcados, módulos de *hardware* dedicados, memórias e um meio de interconexão (WOLF, 2004). A utilização de MPSoCs possibilita que os sistemas embarcados sejam mais eficientes e baratos, uma vez que propicia maior velocidade de operação, e redução no consumo de potência na execução de aplicações. Desta forma, a utilização de MPSoCs se tornou uma tendência no projeto de sistemas embarcados (TANURHAN, 2006).

Para que as aplicações possam ser paralelizadas em um MPSoC, é necessário dividi-las em tarefas. Então, estas tarefas são executadas paralelamente em processadores distintos. Alguns fatores interferem no desempenho das aplicações e devem ser considerados a fim de que a paralelização ocorra de forma eficaz. Dentre eles, cita-se o paradigma de programação selecionado, o qual implica diretamente no comportamento dos processadores e na quantidade de informações trocadas entre eles. Os paradigmas mais populares na literatura são: Mestre - Escravo, *Pipeline*, Fases Paralelas e Divisão e Conquista.

Atualmente, existem vários modelos de MPSoCs, tanto acadêmicos como modelos industriais. Cita-se como exemplos de MPSoCs acadêmicos: HeMPS (CARARA et al., 2009), STORM (REGO, 2006) e SoCLib (CONSORTIUM et al., 2003). Dentre os MPSoCs industriais, pode-se citar o Tile-Gx100 (TILERA, 2012), o Kalray MPSoC (DINECHIN et al., 2014), e o KiloCore (BOHNENSTIEHL et al., 2016) que incluem 100, 256 e 1000 núcleos, respectivamente. Apesar da existência de vários MPSoCs industriais, estes, em sua maioria, ainda não chegaram ao consumidor, uma possível causa para tal efeito é a falta de gerenciamento eficiente para tais sistemas, ou a falta de modelos de programação adequados, dentre outros (CARVALHO, 2009).

Neste contexto, pôde-se observar uma alta complexidade no gerenciamento e codificação de aplicações para MPSoCs, o que pode resultar em: dificuldade de aprendizado, maior probabilidade de cometer erros, grande parcela de tempo e esforço gasto em trivialidades, falta de otimização dos recursos, e inconsistências em aplicações desenvolvidas. Portanto, é necessário o desenvolvimento de uma ferramenta que auxilie o gerenciamento e a codificação de aplicações para MPSoCs. Além disso, não foram encontrados trabalhos que investiguem diferentes paradigmas de programação em MPSoCs. Considerando que já existem MPSoCs industriais, torna-se eminente o preenchimento desta lacuna, para que no momento de programá-los possa ser escolhido o paradigma mais adequado.

1.1 Objetivos do Trabalho

Dado o contexto apresentado, o objetivo principal deste trabalho é realizar uma investigação sobre os diferentes paradigmas de programação no MPSoC HeMPS. Para

isso, foram definidos os seguintes objetivos estratégicos:

- Investigar os tópicos abordados na literatura sobre MPSoCs;
- Investigar paradigmas de programação paralela;
- Investigar trabalhos que avaliam os paradigmas de programação paralela;
- Investigar critérios utilizados para avaliar aplicações executando em MPSoCs

Dentre os objetivos específicos deste trabalho, constam:

- Implementar um conjunto de aplicações considerando os diferentes paradigmas;
- Definir critérios para avaliar as aplicações implementadas;
- Elaborar uma ferramenta que auxilie desenvolvedores na programação de aplicações para a plataforma HeMPS; e
- Avaliar o desempenho das aplicações implementadas usando os critérios definidos.

1.2 Organização do Documento

Este documento está organizado da seguinte forma:

- Capítulo 1: Apresenta uma introdução ao tema abordado, abrangendo conceitos e definições de elementos essenciais ao estudo, motivação e objetivos do presente trabalho;
- Capítulo 2: Apresenta a fundamentação teórica, que foi utilizada como base para o desenvolvimento deste estudo. Sobretudo são descritos MPSoC, aplicações e paradigmas utilizados;
- Capítulo 3: Apresenta os trabalhos relacionados a este estudo, os quais foram classificados em 2 grupos: trabalhos que desenvolvem aplicações em MPSoCs com base em algum paradigma, e trabalhos que investigam diferentes paradigmas em arquiteturas como *clusters* e processadores *multicore*;
- Capítulo 4: Primeiramente, apresenta o desenvolvimento das aplicações Multiplicação de Matrizes, Manipulação de Imagens e AES para cada paradigma. Após, a é apresentada a ferramenta HeMPS *Paralel Programming* desenvolvidas para facilitar a paralelização de aplicações para o MPSoC HeMPS;
- Capítulo 6: Apresenta a configuração do MPSoC utilizado para realizar as simulações, os critérios utilizados para avaliar os diferentes paradigmas de programação, os resultados obtidos, assim como a análise realizada sobre estes resultados; e

- Capítulo 7: Apresenta a conclusão do trabalho, descrevendo em linhas gerais o desenvolvimento e a analise realizada. Além disso, apresenta as ameaças a validação deste estudo e as sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo são apresentados conceitos e definições de elementos que nortearam o desenvolvimento deste trabalho. Primeiramente, na seção 2.1, são apresentadas características referentes ao MPSoC HeMPS. Na seção 2.2, são apresentados os paradigmas de programação abordados neste trabalho. Após, na seção 2.3 são discutidas as aplicações implementadas. O conteúdo descrito neste Capítulo é fruto de uma pesquisa que consistiu na leitura de artigos, textos, livros e materiais pertinentes a cada tópico estudado.

2.1 MPSoC HeMPs

As aplicações implementadas neste trabalho foram descritas no Sistema Multiprocessado Hermes (HeMPS - do inglês *Hermes Multiprocessor System*). A HeMPS é um MPSoC homogêneo baseado no processador Plasma e na rede intra-chip (NoC - do inglês *Network-on-Chip*) Hermes (CARARA et al., 2009). A seguir são descritas a arquitetura, o Sistema Operacional (SO) e a parte de execução de aplicação de usuário da HeMPS.

2.1.1 Arquitetura

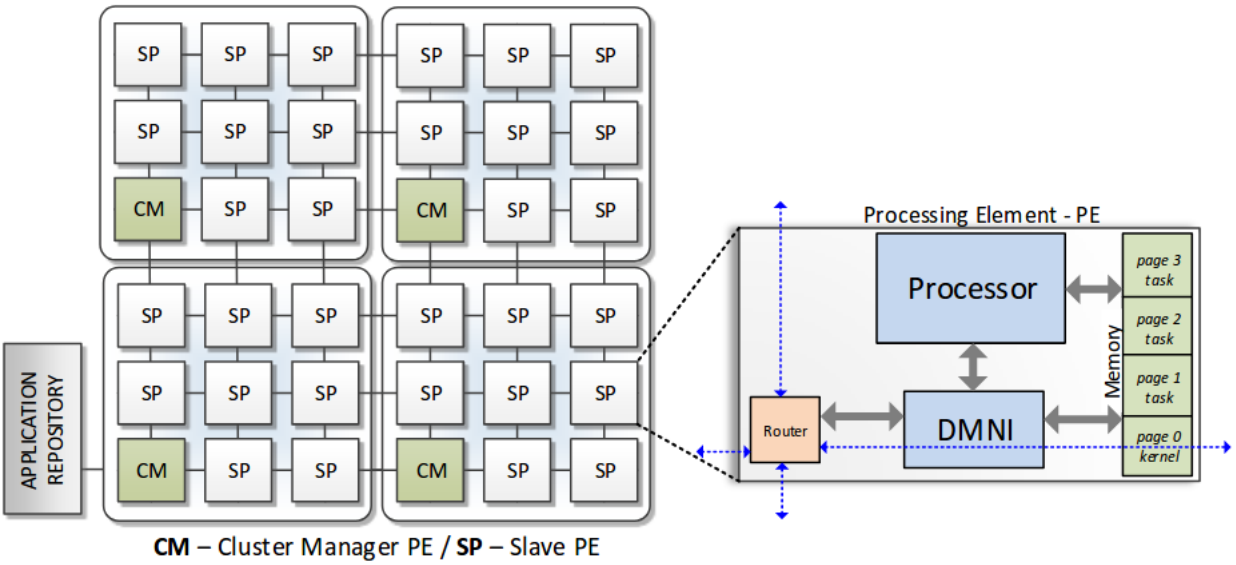
A Figura 2, apresenta uma visão geral do MPSoC HeMPS, onde os Plasmas-IP são os elementos de processamento (PEs) do sistema e estão classificados em dois tipos: mestre ou gerenciador de *clusters* (CM) e processadores escravos (SPs). Os recursos são gerenciados pelo CM, enquanto os SPs executam as aplicações. Além do processador propriamente dito, estão contidos em cada PE: uma memória (*Memory*) particionada em páginas, um roteador (*Router*) e uma interface de memória direta (DMNI - do inglês *Direct Memory Network Interface*). A seguir, encontra-se uma descrição mais detalhada de cada componente contido nos PEs do MPSoC HeMPS (MORAES et al., 2016).

- **Processador Plasma** (OPENCORES, 2001): Processador com um conjunto reduzido de instruções (RISC - do inglês *Reduced Instruction Set Computer*) de 32 bits com subconjunto de instruções de um microprocessador sem estágios intertravados de *pipeline* (MIPS - do inglês *Microprocessor without interlocked pipeline stages*). Apresenta uma organização de memória *Von Neumann* e oferece suporte a linguagem C e tratamento de interrupções.

- **Memória Privada**: Armazena o SO e as tarefas de aplicação do usuário. Nos PEs escravos, a memória é dividida em páginas de tamanho fixo onde é realizada a alocação de tarefas e cada tarefa pode utilizar apenas uma página de memória. Possui uma interface de porta dupla que permite acesso simultâneo ao processador e ao DMNI. O tamanho total da memória é parametrizável.

- **Roteador**: É responsável pelo envio e recebimento de pacotes. Cada roteador contém: *buffers* de entrada, uma lógica de controle compartilhada por todas as

Figura 2 – Estrutura do MPSoC HeMPs.



Fonte: Moraes et al. (2016).

portas, um *crossbar* interno e até cinco portas bidirecionais (norte, sul, leste, oeste e local). A porta local estabelece a comunicação entre o roteador e o IP, enquanto as demais portas são utilizadas para comunicação com os roteadores vizinhos.

- **DMNI** (RUARO et al., 2016): Integra, em um único componente, funcionalidades de acesso direto à memória (DMA - do inglês *Direct Memory Access*), e interface de rede, efetuando a conexão entre o roteador da NoC e a memória local, e possibilitando que o processador possa continuar sua execução de tarefas, sem controlar diretamente a troca de mensagens com a rede. O DMNI suporta recepção e transmissão de pacotes simultâneos.

Como apresentado na Figura 2, os PEs são agrupados em *clusters*, o que proporciona características como escalabilidade de gerenciamento, isolamento de trânsito, permissão de reagrupamento, e acesso ao repositório externo (*Aplication Repository*) por apenas um processador mestre. O repositório de tarefas é uma memória externa ao MPSoC, contendo o código-objeto de todas as tarefas que executarão no sistema.

2.1.2 Sofware

Em nível de *software*, cada processador escravo executa um SO denominado *microkernel*. O *microkernel* é responsável pela comunicação entre tarefas, serviços de gerenciamento do processador e execução multitarefa. Outro componente de software, são as

tarefas de usuário, que são encarregadas pelas características lógicas referentes a execução de aplicação de usuário.

2.1.2.1 Gerenciamento do Sistema

A HeMPS possui um gerenciamento baseado em *clusters*, o que significa que o sistema está dividido em grupos de processadores gerenciados por um CM. O CM executa o *kernel* gerenciador, que realiza as seguintes funções (MORAES et al., 2016):

- **Mapeamento de Tarefas**: A aplicação é armazenada no repositório de tarefas, e solicita a execução no MPSoC. O CM recebe o pedido e seleciona um *cluster* apropriado para receber a descrição da aplicação. Então, o código é transferido para o CM do *cluster* selecionado, e este mapeia as tarefas entre os SPs.

- **Reclusterização**: Ocorre quando é necessário realizar a alocação de uma tarefa e todos os SPs do *cluster* estão ocupados. Neste caso, o CM daquele *cluster* requisita recursos para outros CMs, então escolhe um dos recursos e libera os demais. Por fim, a tarefa é alocada no processador escravo emprestado.

- **Migração de Tarefas**: A HeMPS inclui um protocolo de migração de tarefas que auxilia no processo de reclusterização, migrando tarefas que foram mapeadas em *clusters* vizinhos para o *cluster* local. Quando determinada tarefa, termina sua execução, o CM do *cluster* é avisado, e verifica se existem tarefas de sua responsabilidade alocadas em *clusters* vizinhos. Caso haja, é realizada uma requisição de migração e a tarefa é realocada.

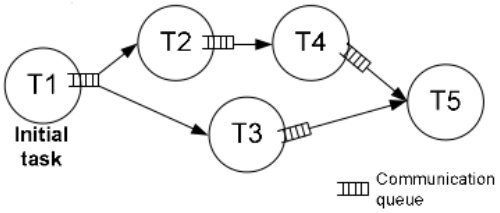
2.1.2.2 Execução de Aplicação de Usuário

. Uma aplicação pode ser definida como um conjunto de tarefas, onde cada tarefa executa uma parte da funcionalidade do algoritmo. Na plataforma HeMPS, a comunicação entre as tarefas pode ser modelada na forma de grafos de tarefas, como apresentado na Figura 3, onde os vértices $T1, T2, T3, T4$ e $T5$ representam o conjunto de tarefas da aplicação, e as arestas representam a comunicação entre as tarefas.

As aplicações de usuário são executadas pelos processadores escravos, já descritos anteriormente. Estes processadores contém o sistema operacional *kernel slave* (escravo), o qual possui uma API para a implementação de tarefas de usuário. Nesta API é definida a estrutura para a comunicação entre as tarefas, chamada de $Message$, e composta por um vetor msg de 128 elementos e por um inteiro $length$ que representa o tamanho da mensagem. Na API são definidas as seguintes funções que realizam chamadas de sistema:

- ***Send***: Função utilizada para enviar uma mensagem para outra tarefa. Devem ser passados por parâmetro a mensagem a ser enviada e a tarefa alvo.

Figura 3 – Grafo de Tarefas.



Fonte: Castilhos et al. (2014).

- ***Receive***: Função utilizada para receber uma mensagem provinda de outra tarefa. Devem ser passados por parâmetro a mensagem recebida e o nome da tarefa que enviou a mensagem.
- ***GetTick***: Captura o tempo do processador em *Ticks*.
- ***Echo***: Semelhante ao *printf* da linguagem *C*, imprime a informação na tela.
- ***Exit***: Interrompe a execução do programa.

A seguir, é demonstrada uma aplicação Produtor (Listing 2.1) - Consumidor (Listing 2.2), onde o uso dos recursos descritos pode ser observado. No exemplo, a tarefa *produtor* preenche 250 mensagens, de tamanho 30, e as envia para a tarefa *consumidor*. A tarefa consumidor aguarda a recepção de 250 mensagens enviadas pela tarefa *produtor*.

```
1  #include <api.h>
2  #include <stdlib.h>
3  Message msg;
4  int main(){
5      int i, j,t;
6      Echo("task produtor started.");
7      Echo(itoa(GetTick()));
8      for(i=0;i<250;i++){
9          msg.length = 30;
10         for(j=0;j<30;j++) msg.msg[j]=i;
11         Send(&msg,consumidor);
12         Echo("Message sent");
13     }
14     Echo(itoa(GetTick()));
15     Echo("task produtor finished.");
16     exit();
17 }
```

Listing 2.1 – Tarefa - produtor Adaptado de (GAPH, 2017)

```
#include <api.h>
#include <stdlib.h>
Message msg;
int main(){
    int i;
    Echo("task consumidor started.");
    Echo(itoa(GetTick()));
    for(i=0;i<250;i++){
        Receive(&msg,produtor);
        Echo("Message received");
    }
    Echo(itoa(GetTick()));
    Echo("task consumidor finished.");
    exit();
}
```

Listing 2.2 – Tarefa - consumidor Adaptado de (GAPH, 2017)

Além da criação de arquivos para cada tarefa, são necessários dois arquivos de configuração: (i) arquivo do projeto, que especifica a(s) tarefa(s) inicial(is) da aplicação e as dependências existentes; e (ii) arquivo de *testcase* que define questões de *hardware*, *software*, e aplicação. Cada parâmetro definido no *testcase* é descrito a seguir:

- ***page_size_KB***: Tamanho de cada página de tarefas;
- ***tasks_per_PE***: Número máximo de tarefas executadas em cada PE;
- ***repository_size_MB***: Tamanho do repositório de tarefas;
- ***model_description***: Descrição da infraestrutura de hardware;
- ***noc_buffer_size***: Profundidade dos *buffers* dos roteadores da NoC.
- **dimensions**: Define a dimensão do MPSoC.
- ***cluster_size***: Define a dimensão de cada cluster do MPSoC.
- ***master_location***: Localização do processador mestre;
- ***mapping_algorithm***: Define o algoritmo de mapeamento de tarefas;
- ***task_scheduler***: Define o algoritmo de agendamento de tarefas;
- ***name***: Nome da aplicação; e
- **star_time_ms**: Tempo de início de execução da aplicação.

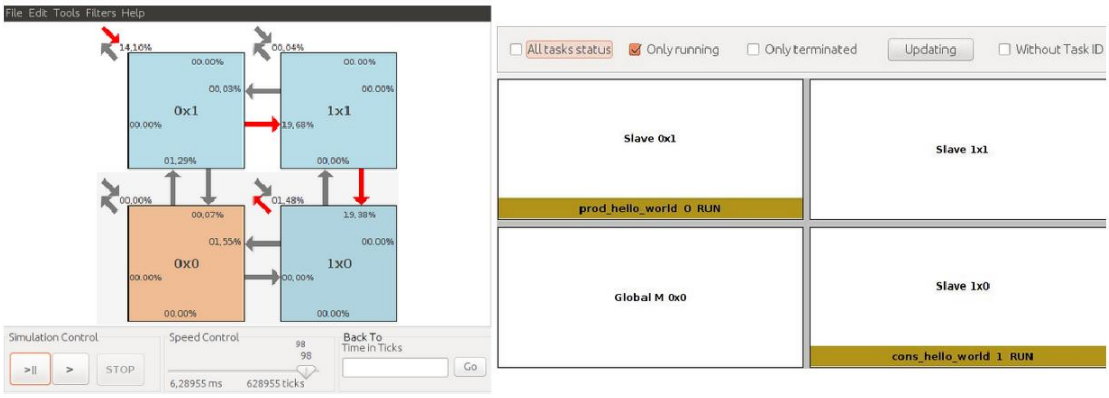
Em suma, o desenvolvimento de aplicações no MPSoC HeMPS demanda um grande esforço, e quanto maior o número de tarefas, maior será o número de arquivos

a serem criados. Dado que MPSoCs visam utilizar centenas de processadores, o processo de desenvolvimento pode se tornar muito árduo e custoso, a medida que mais tarefas sejam adicionadas. Além disso, a probabilidade de cometer erros também será maior com mais tarefas executando a aplicação, e o processo de depuração pode ser extremamente difícil e demorado (VIDAL; MELLO, 2018).

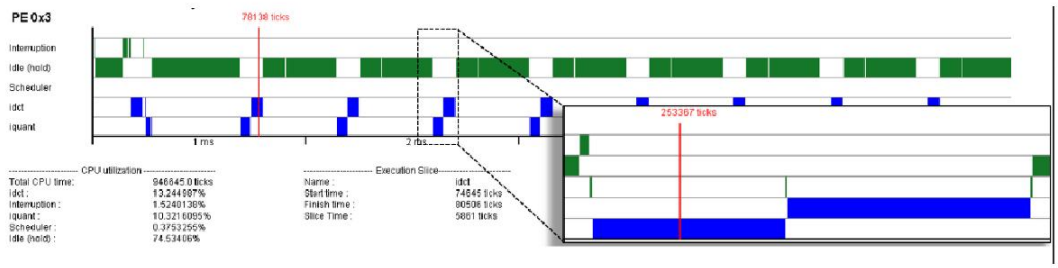
2.1.3 Depuração na Plataforma HeMPS

A HeMPS dispõe de interfaces gráficas, que permitem a depuração das aplicações. A tela principal (Figura 4 (a)) mostra o fluxo de comunicação entre os PEs e a utilização das portas, em tempo de execução. A tela apresentada na Figura 4 (b) mostra a ocupação dos PEs pelas tarefas, e o *status* de cada tarefa. Na Figura 4 (c) podem ser visualizados: utilização, interrupções, ociosidade e agendamento do processador ao longo do tempo.

Figura 4 – Interface HeMPS.



(a) Tela Inicial **(b) Mapeamento das tarefas**



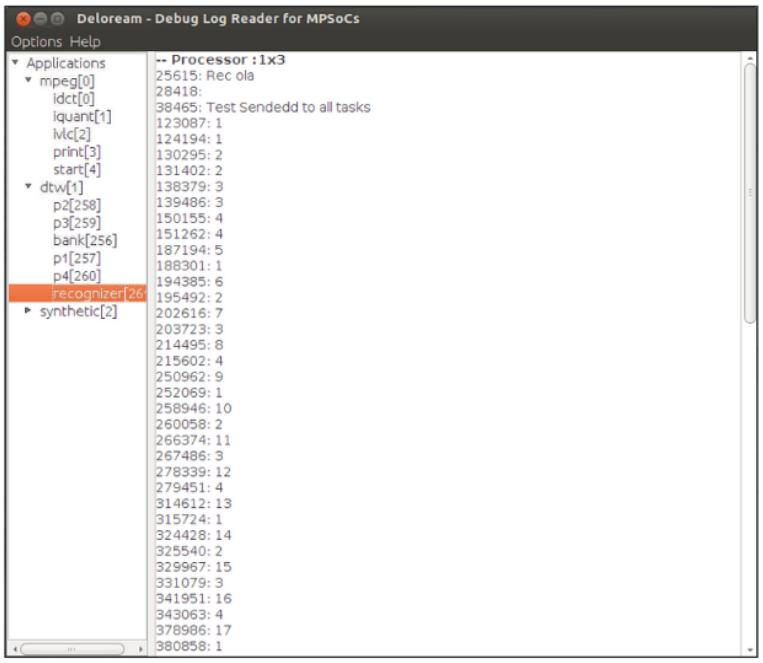
(c) Utilização da CPU

Fonte: GAPH (2017).

Outras ferramentas contidas na HeMPS são a *Deloream* (Figura 5), utilizada para depurar os *logs* de tarefas do aplicativo, e a *Communication Overview* (Figura 6), que

permite a visualização da ocupação dos canais de comunicação dos PEs.

Figura 5 – Tela de Depuração da plataforma HeMPs.



Fonte: Moraes et al. (2016)

Figura 6 – Task Mapping Overview.

Fonte: Moraes et al. (2016).

Através dessas interfaces pode ser realizada a simulação, depuração e extração de dados das aplicações. A HeMPS suporta várias etapas do projeto MPSoC, incluindo personalização da arquitetura, mapeamento de aplicações e integração de *hardware-software*.

Todas as telas apresentadas nas figuras foram utilizada na depuração das aplicações implementadas neste trabalho.

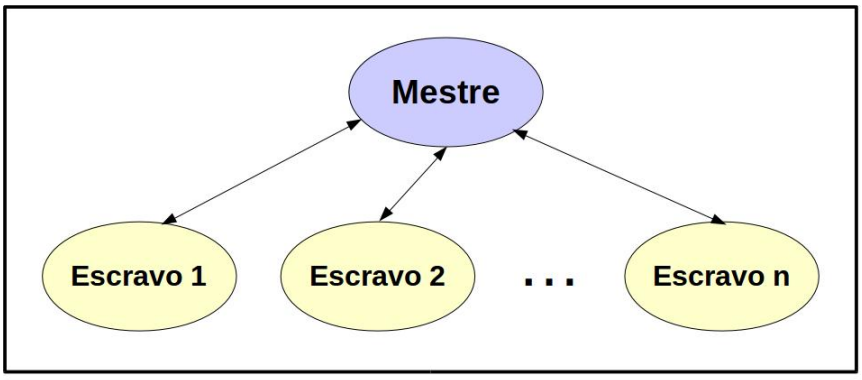
Diversos trabalhos vem sendo publicados utilizando a HeMPS como plataforma base. Carara et al. (2009), Fochi (2015) e Carvalho (2009) são trabalhos referentes a serviços de comunicação no nível de rede, técnicas de tolerância a falhas, e mapeamento dinâmico de tarefas.

2.2 Paradigmas de Programação

De acordo com Baldo et al. (2006), um dos maiores problemas na área de computação de alto desempenho é a dificuldade de definir qual a melhor estratégia de paralelização de uma aplicação. Dentre os principais paradigmas de programação, se encontram: Mestre-Escravo, Fases Paralelas, *Pipeline*, e Divisão e Conquista. Estes paradigmas divergem na forma em que a aplicação é implementada. A seguir estão contextualizados os paradigmas já referidos:

- **Mestre - Escravo**: Este paradigma consiste em duas entidades: mestre e múltiplos escravos. O mestre é responsável pela decomposição do problema em pequenas tarefas, em distribuir estas tarefas entre os escravos, e reunir os resultados parciais obtidos (GALANTE et al., 2004). Os escravos por sua vez, recebem uma mensagem com a tarefa, processam a tarefa e retornam o resultado para o mestre (MEYER, 2017), de modo que os escravos não se comunicam entre si. Conforme Meyer (2017), apesar deste paradigma alcançar elevadas acelerações lineares e graus de expansibilidade, se o número de processadores for muito grande, o controle centralizado pode provocar um gargalo no sistema. A Figura 7 apresenta o grafo deste paradigma.

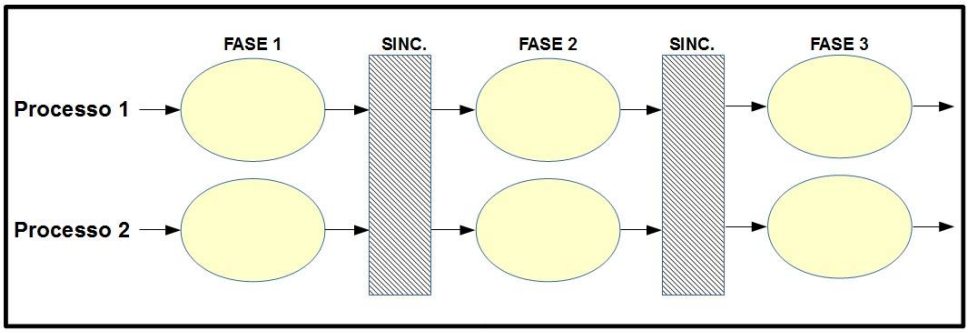
Figura 7 – Paradigma de Programação Mestre-Escravo.



Fonte: Próprio Autor.

- **Fases Paralelas**: Paradigma mais comumente utilizado, conforme (GALANTE et al., 2004) e (MEYER, 2017). Tem como característica a divisão da execução em duas fases básicas: a fase de processamento e a fase de transmissão. A sincronização de todos os processos ocorre na fase de transmissão, e esta é uma desvantagem deste paradigma, já que facilita a ocorrência de sobrecarga na comunicação. Este paradigma está representado na Figura 8, contendo 2 processos que ao final de cada processamento realizam a sincronização entre si, e então iniciam uma nova fase;

Figura 8 – Paradigma de Programação Fases Paralelas.

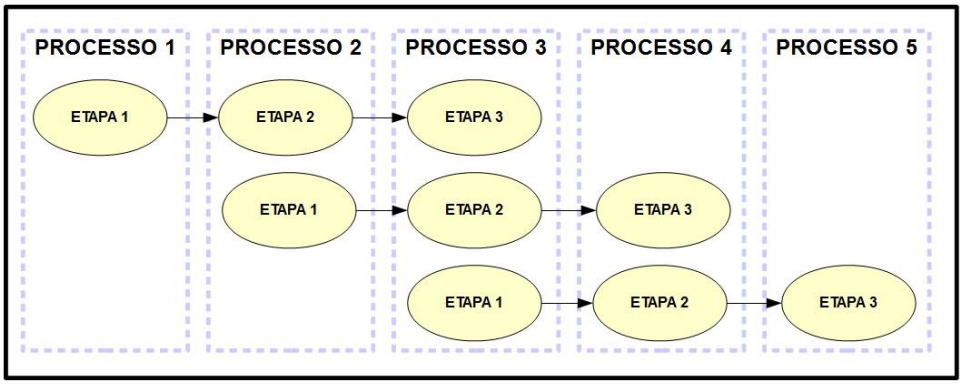


Fonte: Próprio Autor

- **Pipeline**: Neste paradigma, a aplicação é dividida em etapas, que são organizadas em um *pipeline*, como apresentado na Figura 9. Pode ser um paradigma desvantajoso para aplicações que não possuem vários estágios em sequência, uma vez que o fluxo de execução é quase que sequencial. Dependendo do número de processos disponíveis, a aplicação pode ser executada em mais de um fluxo de execução, como exemplificado na Figura 9 onde são demonstrados 3 *pipelines*.

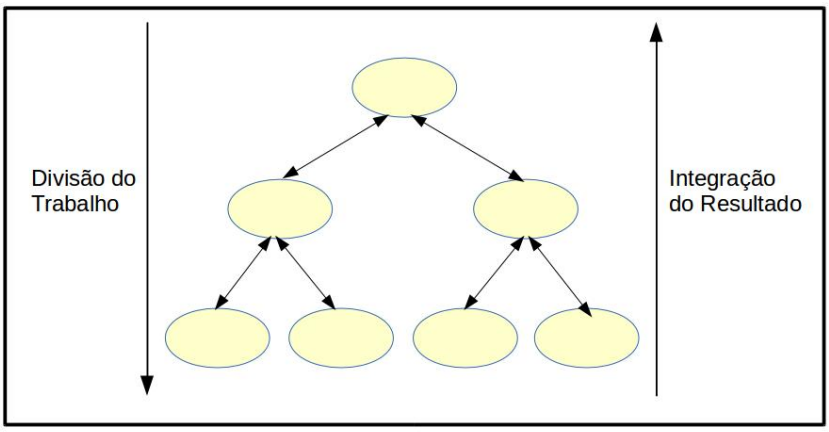
- **Divisão e Conquista**: Paradigma onde uma instância de um problema, é decomposta em sub-instâncias menores, que são resolvidas separadamente. O problema é decomposto até que a solução da sub-instância seja simples e tenha solução imediata, e então as soluções parciais são combinadas até se obter a solução inteira do problema. É uma modificação do paradigma mestre-escravo, pois ao invés de haver um processo mestre responsável pela divisão da carga de trabalho, dois ou mais nós realizam a distribuição das tarefas em uma hierarquia (MEYER, 2017), assim como apresentado na Figura 10.

Figura 9 – Paradigma de Programação Pipeline.



Fonte: Próprio Autor.

Figura 10 – Paradigma de Programação Divisão e Conquista.



Fonte: Próprio Autor.

2.3 Aplicações

Nesta seção são discutidas as aplicações implementadas neste trabalho. Na subseção 2.3.1, são apresentados conceitos relacionados a aplicação Multiplicação de Matrizes. A aplicação de Manipulação de Imagens é descrita na subseção 2.3.2. E por fim é apresentada, na subseção 2.3.3, a aplicação Padrão de Criptografia Avançada *Rinjdael* (AES do inglês *Advanced Encryption Standard*).

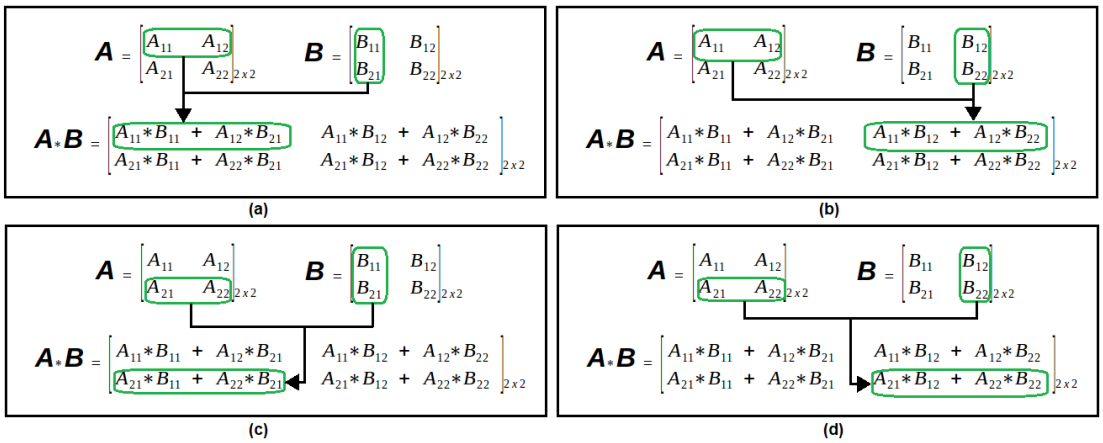
2.3.1 Multiplicação de Matrizes

Dentre os métodos que realizam a multiplicação de matrizes, o mais comum consiste na multiplicação dos elementos das linhas da matriz A pelos elementos das colunas da matriz B, seguida do somatório dos produtos resultantes (PRESS, 2007), como apresentado na Equação 2.1.

$$(AB)_{ij} = \sum_{m=1}^{N} a_{im} b_{mj} = a_{i1} b_{1j} + a_{i2} b_{2j} + ... + a_{iN} b_{1N} \tag{2.1}$$

Iezzi e Hazzan (2004) mencionam que a existência de um produto entre A e B é condicionada à equivalência entre o número de colunas de A e o número de linhas de B. A multiplicação de matrizes é demonstrada em detalhes na Figura 11, que contém duas matrizes A e B, ambas quadradas (mesmo número de linhas e colunas) e de ordem 2 (ambas possuem duas linhas e duas colunas), e uma terceira matriz que representa a multiplicação de A por B. Primeiramente, é realizado o produto entre os elementos da primeira linha de A com os elementos da primeira coluna de B, então os valores resultantes são somados, como indicado na Figura 11(a). Este procedimento é realizado para as demais linhas de A e colunas de B, como apresentado na Figura 11(b) (c) e (d).

Figura 11 – Exemplo de Multiplicação de Matrizes.



Fonte: Próprio Autor.

Partindo da premissa que três matrizes (A, B e C) são quadradas, de ordem n e não-nulas, um exemplo de algoritmo sequencial da multiplicação entre matrizes é apresentado no Listing 2.3, onde é realizada a multiplicação das matrizes A e B, e o resultado é armazenado na matriz C. O algoritmo contém três laços de repetição (comandos for) aninhados que percorrem de 0 até $n-1$, obtendo assim uma complexidade $\mathcal{O}(n^3)$.

```
1 for(i = 0; i < n; i++){
2     for(j = 0; j < n; j++){
3         for(k = 0; k < n; k++){
```

```
4                C[i][j] += A[i][k] * B [k][j];
5            }
6        }
7    }
```

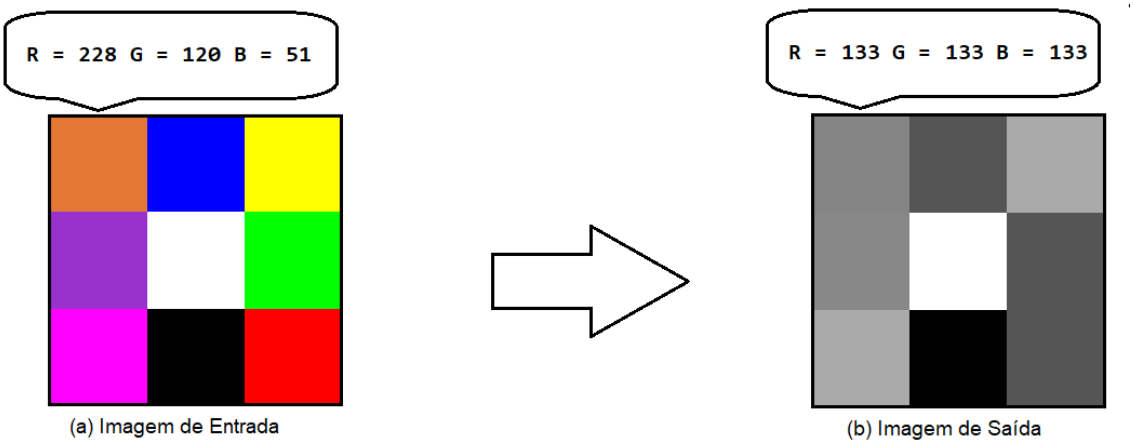
Listing 2.3 – Algoritmo Multiplicação de Matrizes.

2.3.2 Manipulação de Imagens

Esta aplicação se baseia no trabalho de Meyer (2016), que tem como objetivo criar um modelo que execute aplicações em *pipeline* com a utilização de elasticidade. Nesta aplicação são realizadas três transformações em uma imagem representada no padrão Vermelho - Verde - Azul (RGB - do inglês *Red - Green - Blue*). RGB é um modelo de cor concebido com base nos dispositivos de saída gráfica com três cores primárias: vermelho, verde e azul (LOPES, 2013). Sendo que cada cor primária pode variar entre 0 e 255, resultando na cor do pixel da imagem. Cada transformação da imagem é explicada a seguir:

- **Tons de cinza** : Nesta etapa é realizada uma média dos valores primários contidos em cada pixel. Então, o valor de cada cor primária é alterado para a média do pixel em questão, resultando em uma imagem com tons de cinza. Um exemplo é apresentado na Figura 12, em que (a) exibe a imagem de entrada e (b) a imagem de saída (após a transformação). Como pode ser observado na Figura 12(a), a cor de tom laranja é formada pelas cores primárias $R = 228$, $G = 120$ e $B = 51$. A média desses valores primários é 133, portanto a cor da imagem de saída é composta por $R = 133$, $G = 133$ e $B = 133$.

Figura 12 – Exemplo de Transformação em Tons de Cinza.



Fonte: Próprio Autor.

O Listing 2.4 apresenta o algoritmo sequencial desta etapa da aplicação, no qual os valores RGB são armazenados em um vetor *img*. Cada pixel de *img* é percorrido pelo laço de repetição da linha 1. Outros dois laços, das linhas 5 e 13, percorrem os valores RGB de cada pixel, somando-os e substituindo-os pela sua média, respectivamente. Assim, todos os valores de *img* são percorridos duas vezes, resultando em uma complexidade linear $\mathcal{O}(n)$. A imagem também pode ser representada por um vetor do tipo *struct*, onde a *struct* representa um pixel e contém três valores inteiros que equivalem aos valores RGB;

```
1  for(i = 0; i < n; i += 3){
2      aux = 0;
3      /*soma os tres valores do pixel i*/
4      for(j = 0; j < 3; j++){
5          aux += img[i+j];
6       }
7       /*calcula a media*/
8       media = aux/3;
9       /*armazena a media nos valores primarios do pixel i*/
10     for(j = 0; j < 3; j ++){
11         img[i+j] = media;
12     }
13 }
```

Listing 2.4 – Algoritmo da Transformação em Tons de Cinza.

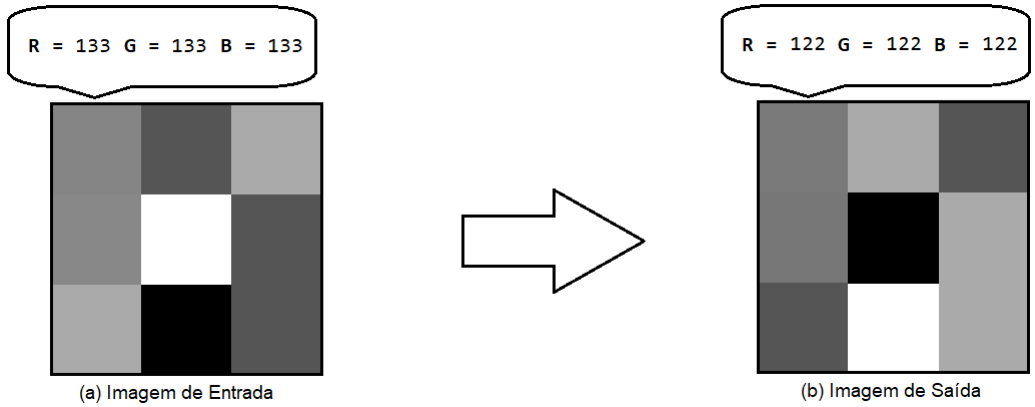
- **Inversão de Cores** : Nesta etapa subtrai-se o valor máximo (255) pelo valor do pixel em questão. A Figura 13 apresenta em (a) a imagem de entrada e em (b) a imagem de saída. Como pode ser observado, o quadrado superior esquerdo possui a cor $R = 133$, $G = 133$ e $B = 133$, logo após a operação $(255 - 133)$ a cor resultante é $R = 122$, $G = 122$ e $B = 122$. Assim, as áreas escuras se tornaram claras e vice-versa.

  O algoritmo sequencial desta etapa é apresentado no Listing 2.5. Os pixels são percorridos através do laço de repetição da linha 1, e em cada iteração deste laço calcula-se o valor da inversão. Então, o resultado é armazenado nos valores primários de cada pixel. A complexidade deste algoritmo é linear e representada por $\mathcal{O}(n)$.

```
1  for(i = 0; i < n; i += 3){
2      /*calcula a inversao*/
3      aux = 255 - img[i];
4      /*armazena o resultado nos valores RGB do pixel*/
5      for(j = 0; j < 3; j++){
6          img[i+j]= aux;
7       }
8  }
```

Listing 2.5 – Algoritmo da Inversão de Cores.

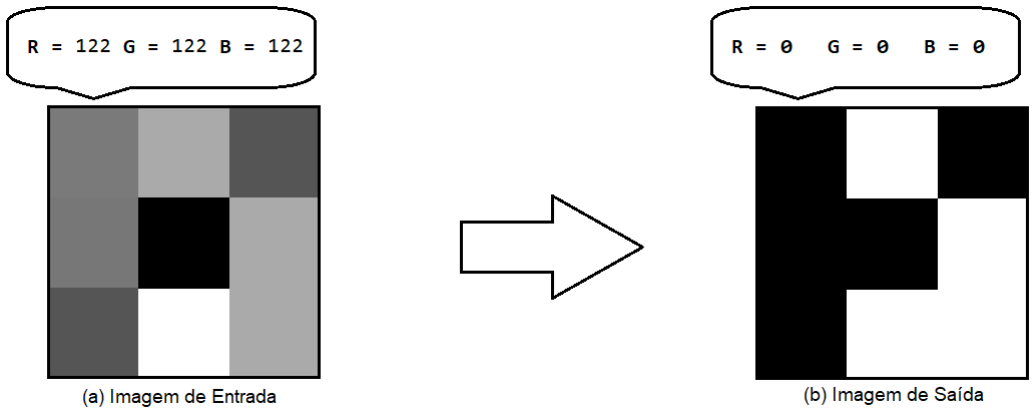
Figura 13 – Exemplo da Inversão de Cores.



Fonte: Próprio Autor.

- **Limiarização** : Nesta etapa é definido um valor limite, e então cada ponto é verificado, se estiver abaixo desse limiar é alterado para (0,0,0) correspondendo a cor preta e se estiver acima, ou igual, é alterado para (255,255,255) correspondendo a cor branca. Alterando a imagem para branco e preto a partir de um parâmetro de análise, como exemplificado na Figura 14. Por exemplo, o quadrado superior esquerdo que possui a cor $R = 122$, $G = 122$ e $B = 122$ na imagem de entrada, por estar abaixo do limiar (150), é alterado para cor $R = 0$, $G = 0$ e $B = 0$.

Figura 14 – Exemplo de Transformação de Limiriazação (limiar 150).



Fonte: Próprio Autor.

O Listing 2.6 apresenta o algoritmo sequencial desta etapa da aplicação. Assim como o evidenciado nas outras etapas, este algoritmo apresenta um comando for externo que percorre todos os pixels de um vetor img. Para cada pixel verifica-se, através do comando if, se o valor está acima ou abaixo do limiar. Então percorre-se

os valores RGB, substituindo-os pelos valores 255 ou 0. Esta etapa da aplicação também possui uma complexidade linear $\mathcal{O}(n)$.

```
for(i = 0; i < n; i += 3){

    aux = img[i];

    /*verifica se o valor do pixel    maior ou igual que o limiar*/
    if(aux >= 150){

        /*substitui os valores RGB por 255*/
        for(j=0;j<3;j++){

            img[i+j] = 255;
        }

    }else{

        /*substitui os valores RGB por 0*/
        for(j=0;j<3;j++){

            img[i+j] = 0;
        }
    }
}
```

Listing 2.6 – Algoritmo de Limiarização.

Os estágios descritos são executados de forma sequencial. Isso quer dizer que a imagem passa pelo processamento do primeiro estágio e seu resultado serve de entrada para o segundo estágio. Por fim, o resultado do segundo serve de entrada para o terceiro estágio, produzindo a imagem final da aplicação. Dado que todas as etapas possuem uma complexidade $\mathcal{O}(n)$, a aplicação também apresenta esta complexidade.

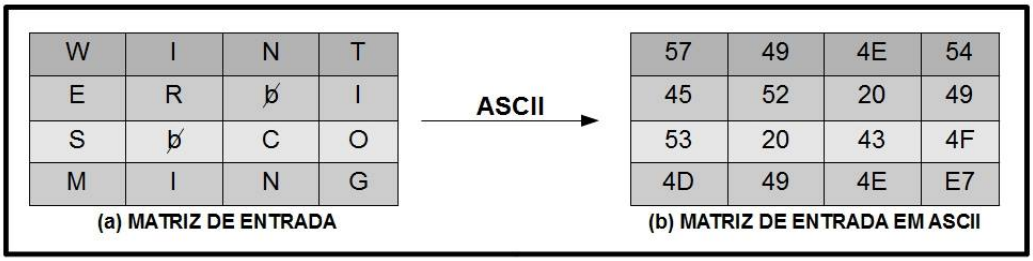
2.3.3 AES

O algoritmo AES, também conhecido como algoritmo de *Rijndael* (DAEMEN; RIJMEN, 1999), é um algoritmo criptográfico que pode ser usado para proteger dados eletrônicos (STANDARD, 2001). O algoritmo foi proposto por Joan Daemen e Vincent Rijmen, sendo vencedor de uma chamada pública realizada pelo Instituto Nacional de Padrões e Tecnologia (NIST - do inglês *National Institute of Standards and Technology*) para a definição de um novo padrão de esquema criptográfico de chave secreta.

O AES foi projetado para realizar a criptografia em sequências de 128 bits. A entrada para o algoritmo AES pode ser representada por uma matriz, onde a sequência de bits é divida em blocos e cada bloco de 8 bits (1 byte) é representado como um elemento

da matriz. Esta matriz é exemplificada na Figura 15 (a), e seus elementos são convertidos em hexadecimal (tabela ASCII apresentada na Figura 15 (b)), para que as operações de criptografia possam ser realizadas.

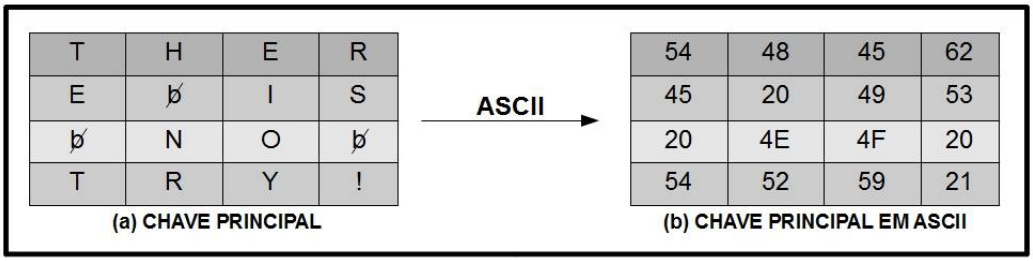
Figura 15 – Matriz de Blocos de Entrada AES.



Fonte: Próprio Autor.

O AES pode utilizar cifras de comprimentos de 128, 192 ou 256 bits (DAEMEN; RIJMEN, 1999). Assim como a entrada para o algoritmo, a chave de codificação também pode ser representada como uma matriz retangular com quatro linhas, como apresentado na Figura 16 (a). Esta matriz também deve ser convertida para hexadecimal, tal como visto na Figura 16 (b).

Figura 16 – Exemplo de Chave Criptográfica.



Fonte: Próprio Autor.

O procedimento de encriptação é composto por diferentes métodos, os quais são executados em rodadas. O número de rodadas é dependente do tamanho da chave utilizada. Na Tabela 1 são apresentadas as combinações de tamanho de bloco, chave e número de rodadas, de acordo com o comprimento da chave criptográfica, sendo Nk = tamanho da chave; Nb = tamanho do bloco; e Nr = número de rodadas.

O primeiro procedimento realizado na aplicação AES é a geração de diferentes sub-chaves para cada rodada. As sub-chaves derivam da chave principal e são compostas

Tabela 1 – Combinações de bloco, chave e rodada - AES.

	Tamanho da Chave (*Nk palavras*)	**Tamanho do Bloco (*Nb palavras*)**	**Número de rodadas (*Nr*)**
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Fonte: Adaptado de (DAEMEN; RIJMEN, 1999)

por uma sequência de 4 bytes, representadas por um único vetor w. Primeiramente, é realizada a concatenação das linhas da chave principal, e os valores são armazenados nas primeiras Nk posições de w. Após são realizadas várias operações, que são descritas a seguir:

- **RotWord**: Rotaciona a palavra em uma posição (1 byte) a esquerda, e o byte mais significativo é empurrado para o final. Por exemplo, rotação da sequência $0x0914dff4$ é $0x14dff409$.

- **SubWord**: Substitui cada byte da palavra através de uma tabela de substituição, chamada S-box, exemplificada na **??**. Por exemplo, na substituição da sequência $0x14dff409$, o primeiro byte (14) é substituído pelo conteúdo da linha 1 e coluna 4 da S-box, o qual corresponde ao byte fa. Logo, o resultado desta operação para a sequência exemplificada é $0xfa9ebf01$.

- **EXOR**: É realizada uma operação XOR (ou exclusivo) entre a palavra gerada na operação anterior e uma constante armazenada em um vetor chamado *Rcon*. *Rcon* contém os valores dados por $[x^{i-1}, 00, 00, 00]$, com x^{i-1} sendo potências de x no campo $GF(2^8)$ (DAEMEN; RIJMEN, 1999). Assim, uma operação XOR entre a sequência $0xfa9ebf01$ e a constante $0x01000000$ resulta na sequência $0xfb9ebf01$.

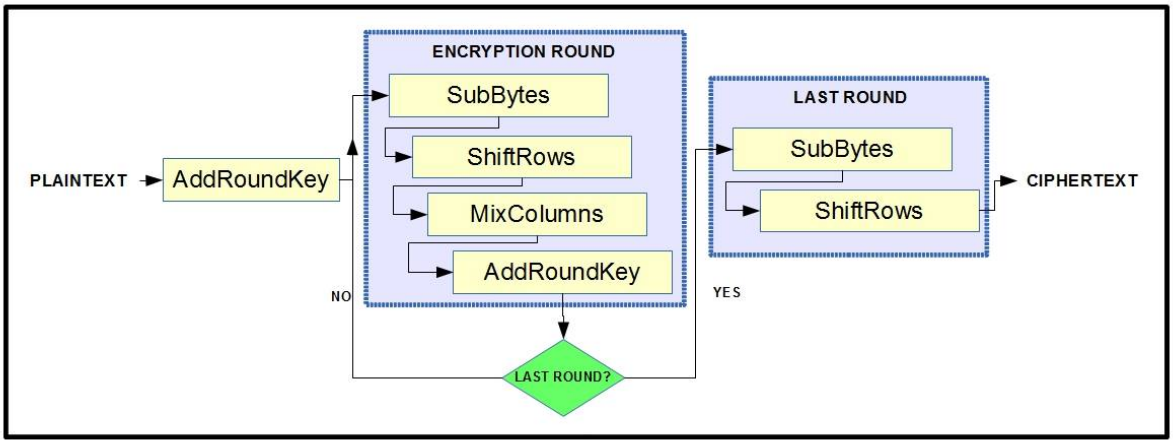
O Listing 2.7 apresenta o pseudocódigo da função *KeyExpansion*, a qual gera as sub-chaves da aplicação AES. A função tem como entrada a chave de criptografia, um vetor w que armazena as subchaves e a variável Nk que representa o número de colunas da chave.

Após a geração das sub-chaves são realizados os procedimentos exibidos na Figura 17. Nestas operações são realizadas alterações em uma matriz, chamada Estado, que tem como elementos iniciais os elementos da matriz de entrada. Como observado na Figura 17, a partir da primeira até a penúltima rodada $(Nr - 1)$, as funções são executadas na ordem: *SubBytes, ShiftRows, MixColumns e AddRoundKey.* E na última rodada (Nr') são executadas apenas as funções: *SubBytes* e *ShiftRows.*

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    WORD temp
    i = 0
    /*concatenacao*/
    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Listing 2.7 – Pseudocódigo para a Geração de Sub-chaves. Fonte: (STANDARD, 2001)

Figura 17 – Modelagem AES.



Fonte: Adaptado de (REZENDE; CAIMI, 2013).

Cada uma das funções, apresentadas na Figura 17, é descrita a seguir:

- **AddRoundKey**: Para cada rodada, uma sub-chave é derivada da chave principal. Nesta operação a sub-chave é somada com o estado atual utilizando a operação

XOR. Por exemplo, o caractere M (0x4D em hexadecimal) e a sub-chave $0x54$, resultam no hexadecimal 19. No Listing 2.8, é apresentado o algoritmo desta etapa.

```
1     for(i=0;i<4;i++){
2       for(j=0;j<4;j++){
3             subkey[j] = (w[i] >> 24 - (j*8)) & 0xFF;
4       }
5       for(j=0;j<4;j++){
6         state[j][i] ^= subkey[j];
7       }
8     }
```

Listing 2.8 – Algoritmo AddRoundKey. Adaptado de (REZENDE; CAIMI, 2013)

.

- **SubBytes**: Realiza a transformação de todos os bytes da matriz Estado utilizando a tabela S-box. Se o elemento do estado for $0x00000021$, então este valor será substituído por $0x000000fd$, que corresponde ao valor contido na linha 2 e coluna 1 da S-box. O Listing 2.9 apresenta como é realizada esta substituição.

```
1 for  (i=0;i<4;i++){
2     for(j=0;j<4;j++){
3
4         state[i][j] = sbox[state[i][j] >> 4][state[i][j] & 0x0F];
5
6     }
7 }
```

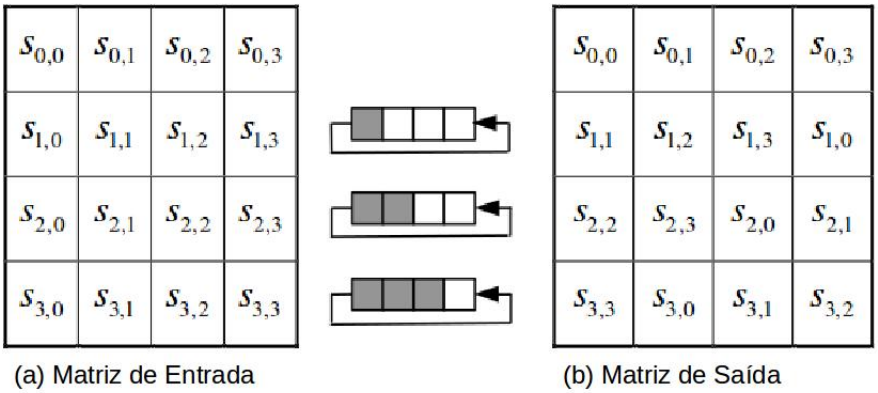
Listing 2.9 – Algoritmo SubBytes.

- **ShiftRows**: nesta operação, as linhas da matriz, que representa o bloco, são rotacionadas de forma circular para a esquerda por 0, 1, 2 e 3 posições, respectivamente. A Figura 18 demonstra como é realizada esta operação, onde a Figura 18(a) corresponde a matriz de entrada e a Figura 18 (b) corresponde a matriz de saída.

- **MixColumns**: este passo realiza uma transformação linear em cada coluna da matriz, como apresentado na Figura 19.

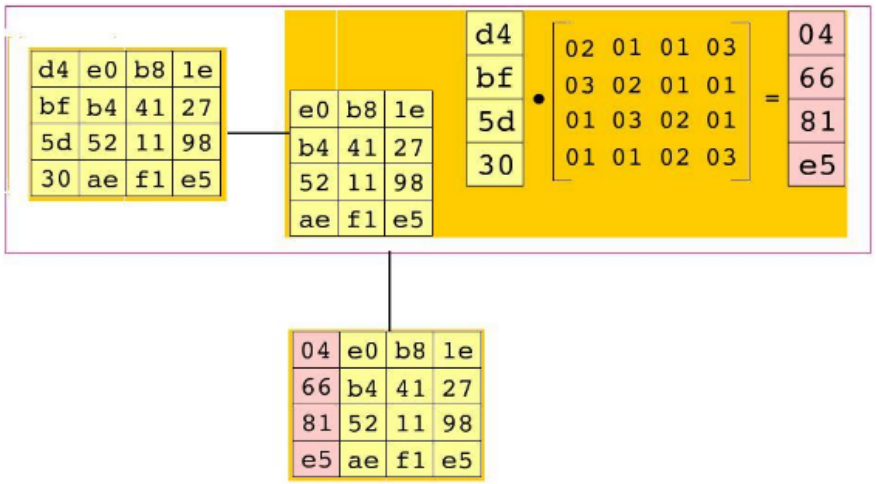
O módulo de decifragem envolve as mesmas funções que realizam transformações equivalentes contrárias. Cada função faz a operação inversa de sua representante no módulo de cifragem.

Figura 18 – Exemplo de Entrada e Saída - ShiftRows.



Fonte: (DAEMEN; RIJMEN, 1999).

Figura 19 – Exemplo de Transformação MixColumns.



Fonte: (MISAGHI, 2017).

3 TRABALHOS RELACIONADOS

Neste Capítulo são apresentados 6 trabalhos relacionados, dentre eles 3 trabalhos investigam os paradigmas de programação em arquiteturas paralelas, enquanto a outra metade explora o desenvolvimento de aplicações em MPSoCs. A tabela abaixo sintetiza os trabalhos relacionados, exibindo para cada trabalho a arquitetura utilizada, seu objetivo e a principal contribuição para este trabalho. Após são apresentadas duas Seções que realizam a comparação entre os trabalhos encontrados.

Tabela 2 – Síntese dos Trabalhos Relacionados.

Referência	**Arquitetura**	**Objetivo da Pesquisa**	**Principal Descoberta**
(RAEDER et al., 2011)	Cluster de computadores pessoais.	Utiliza métodos analíticos para escolher o paradigma mais adequado antes de iniciar a codificação.	O paradigma pipeline se mostrou ineficiente, enquanto os paradigmas divisão e conquista e mestre-escravo foram mais eficientes para entradas menores e maiores, respectivamente.
(CHI; JURLINK; MEENDERINCK, 2010)	MPSoC Cell com 6 PEs.	Encontrar a melhor maneira de aproveitar todo o potencial em um MPSoC Cell.	A aplicação desenvolvida em Mestre-Escravo, apresentou menor desempenho devido a sobrecarga na comunicação do nó mestre.
(LANGE; BOHRER, 2010)	MPSoC HeMPS.	Propor uma adaptação de uma arquitetura MJPEG, baseada em pipeline, para operar em uma arquitetura MPSoC.	O comportamento da NoC Hermes se mostrou adequado para os módulos da aplicação MJPEG.
(REBONATTO et al., 2002)	Cluster de máquinas virtuais.	Comparar os modelos Divisão e Conquista e Mestre-Escravo em uma máquina paralela virtual heterogênea.	O paradigma de Divisão e Conquista não se mostrou eficiente em ambientes heterogêneos.
(SHEE; ERDOS; PARAMESWARAN, 2006)	Processador multicore.	Desenvolver e analisar uma aplicação JPEG implementada utilizando os paradigmas mestre-escravo e pipeline.	Para aplicações que podem ser divididas em estágios o paradigma pipeline pode ser bastante eficiente.
(WANG et al., 2014)	MPSoC.	São analisados o comportamento de quatro aplicações em três MPSoCs com topologias de NoC distintas.	Utilizar o maior número possível de recursos de um processador não garante a melhora de desempenho ou eficiência.

3.1 Paradigmas de Programação: Vantagens e Desvantagens

Em Raeder et al. (2011) são utilizados métodos analíticos para a avaliação de desempenho de aplicações paralelas, como uma alternativa para auxiliar no processo de escolha das melhores estratégias de paralelização. Utilizando redes de autômatos estocásticos, o trabalho modela e avalia o desempenho de aplicações desenvolvidas principalmente para máquinas agregadas (*clusters*). Na comparação dos resultados obtidos na modelagem realizada com os tempos de execução reais, foi utilizada a aplicação Multiplicação de Matrizes em 3 paradigmas diferentes: mestre-escravo, divisão e conquista e *pipeline*. Foram testadas dimensões de matriz de ordens 10000, 20000 e 30000, enquanto o número de processos variou entre 2 e 7 para o paradigma mestre-escravo e entre 2 e 6 para os demais. O paradigma *pipeline* se mostrou ineficiente para este tipo de aplicação, uma vez que esta não apresenta um número grande de estágios. Enquanto que para os paradigmas divisão e conquista e mestre-escravo, o aumento de processos resultou em um maior desempenho. Comparando os dois últimos, pode-se analisar que para a aplicação mestre-escravo obteve um melhor desempenho para matrizes de ordem 20000, enquanto que para as demais, o paradigma divisão e conquista foi mais eficiente.

Em Rebonatto et al. (2002) foram comparados os paradigmas divisão e conquista e mestre-escravo em uma máquina paralela virtual heterogênea. Foi escolhida, como alvo da pesquisa, uma aplicação responsável por encontrar conjuntos de números-primos. Esta aplicação foi desenvolvida usando linguagem de programação *C* e a biblioteca de programação paralela MPI. Para comparação foram empregadas as medidas de desempenho: *Speedup* (aceleração na execução do programa paralelo) e eficiência (fração do tempo que foi efetivamente aproveitado em processamento). Devido ao ambiente heterogêneo, o paradigma divisão e conquista não se mostrou eficiente, já que os processadores com maior poder computacional terminaram suas tarefas antes e precisaram esperar os outros, retardando assim, o tempo de execução do programa. Este problema é menor no paradigma mestre-escravo visto a sua capacidade de adaptação dinâmica da carga enviada para cada nodo.

Em Shee, Erdos e Parameswaran (2006) foi implementada uma aplicação de compressão JPEG, utilizando os paradigmas de aplicação mestre - escravo e *pipeline*, sobre um processador *multicore*. A aplicação JPEG implementada foi particionada em vários estágios que poderiam ser alocados e diferentes processadores. Na implementação mestre-escravo, cada processador escravo implementa as mesmas funções (as que podem ser paralelizadas), sendo que a carga de trabalho é divida pelo processador mestre entre os escravos. Enquanto que na implementação *pipeline* cada processador executa um estágio do programa, e cada estágio precisa esperar pelos dados da etapa anterior. O estágio de codificação de *Huffman* foi considerado o mais crítico pelo autor para ambos paradigmas, devido a impossibilidade de paralelizá-lo. Por fim, o tempo de execução se mostrou menor para o paradigma *pipeline*, enquanto que a área aumentou com o número de núcleos, mas

foi similar para ambos os paradigmas

Dentre os trabalhos que realizam a comparação entre paradigmas, o paradigma mestre-escravo é mais comumente utilizado do que os demais. Pode-se dizer que esta é uma vantagem, pois uma maior quantidade de técnicas e ferramentas são desenvolvidas com foco neste paradigma.

Realizando uma análise sobre os trabalhos de Shee, Erdos e Parameswaran (2006) e Raeder et al. (2011), pode-se observar que o desempenho de uma aplicação implementada sobre vários paradigmas é dependente do tipo da aplicação. Em Shee, Erdos e Parameswaran (2006), por exemplo, a aplicação implementada possui vários estágios, o que contribui para que um melhor desempenho seja alcançado através do paradigma *pipeline*, onde cada processador executa uma funcionalidade diferente. De maneira oposta, a aplicação implementada em Raeder et al. (2011), possui apenas 3 estágios, o que dificulta a paralelização em *pipeline*, porém facilita uma paralelização de dados entre os processadores, o que ocorre nos paradigmas mestre-escravo, e divisão e conquista onde os processadores implementam as mesmas funções sobre dados distintos.

3.1.1 Programação em MPSoCs

Nesta Seção são discutidos como foi realizada a implementação de aplicações dos trabalhos que utilizam MPSoCs na Tabela 1. Como estas arquiteturas são dotadas de características específicas, observa-se diferentes formas de implementações em cada trabalho.

Em Chi, Juurlink e Meenderinck (2010), o desenvolvimento de aplicações paralelas é citado como principal desafio de arquiteturas MPSoC. Com o objetivo de encontrar a melhor maneira de aproveitar todo o potencial que essas arquiteturas oferecem, foram desenvolvidos, em uma arquitetura Cell, duas aplicações paralelas de decodificação H264: *Ring Line* e *Task Pool*. Essas aplicações foram implementadas explorando o paralelismo em nível de macrobloco, onde a aplicação *Task Pool* foi desenvolvida baseada no paradigma mestre-escravo enquanto a *Ring Line* aborda uma estratégia distribuída, onde os núcleos processam linhas inteiras de macroblocos em vez de blocos únicos. O processo de implementação em macroblocos se mostrou bastante complexo, no entanto, a aplicação *Ring Line* apresentou um melhor desempenho em comparação com a *Task Pool*, devido a sobrecarga de comunicação no nó mestre.

Em LANGE e BOHRER (2010), é proposta uma adaptação da aplicação JPEG para uma arquitetura SoC, a qual emprega a NoC Hermes. A prototipação do sistema proposto foi realizado em uma placa FPGA, o que demandou a utilização de diversas ferramentas. O decodificador JPEG é formado por núcleos IPs, conectados por uma estrutura pipeline. A integração entre a NoC e os núcleos IPs, envolve o desenvolvimento de interfaces que façam a conexão e interpretem os protocolos de ambos os lados. Assim, na primeira fase desta integração foram utilizados dois módulos do processo de decodificação

JPEG e entre eles foram inseridos dois roteadores da NoC Hermes. A segunda fase, que envolveria todos os módulos não foi implementada. A aplicação JPEG foi desenvolvida na linguagem *C*. Através da simulação realizada pode-se confirmar a integridade dos dados resultantes. Assim, a NoC Hermes, também empregada neste trabalho, se mostrou adequada para o desenvolvimento dos módulos da aplicação JPEG.

Em Wang et al. (2014), são analisados sistematicamente os comportamentos de computação e comunicação de quatro aplicações em MPSoCs com base em três topologias de NoC: torus, malha e árvore gorda. Primeiramente, as aplicações são modeladas formalmente como grafos de comunicação de tarefa. Após foram alocados recursos de memória adequados para cada aplicação, e realizado o mapeamento de tarefas em processadores distintos. As aplicações foram descritas em *C* e executadas em um simulador MPSoC. Foram analisadas a distribuição de carga, o tráfego na comunicação das aplicações, o desempenho e a eficiência energética de cada implementação. Para a NoC baseada em malha, aplicações com grande número de dados apresentam menos atrasos de pacote quando o número de processadores do MPSoC é aumentado. Estas aplicações podem se beneficiar muito da grande quantidade de recursos de processamento, enquanto outras aplicações não necessitam que todos os recursos sejam utilizados para atingir o ápice de desempenho.

Cada um dos três trabalhos descritos acima, aborda uma forma diferente de implementar suas aplicações. Em Chi, Juurlink e Meenderinck (2010), as aplicações foram implementadas no video-game PlayStation 3, o qual possui um processador Cell de 6 PEs, enquanto que em LANGE e BOHRER (2010) e em Wang et al. (2014) foram realizadas simulações em FPGA e em SystemC, respectivamente. A vantagem de realizar uma implementação diretamente em um MPSoC é de que os resultados são precisos e o tempo de execução menor. Por outro lado, o número de núcleos que estão presentes em MPSoCs que temos contato, como o PlayStation 3, é muito pequeno. Logo, a vantagem de simular um MPSoC está na flexibilidade do número de núcleos, tamanho do buffer de comunicação, e tamanho de memória.

Em outros trabalhos, comprova-se essa grande variedade de maneiras de implementar aplicações em MPSoCs. Alguns utilizam APIs como em Saldana e Chow (2006), onde é empregado MPI (Message Passing Interface), e em Chapman et al. (2009), onde o OpenMP é utilizado. Ou até mesmo, alguns fabricantes como a empresa Tilera, fornecem bibliotecas e ferramentas especificas para o desenvolvimento de aplicações em seus MPSoCs. Neste trabalho é utilizada a simulação do MPSoC HeMPS em SystemC.

A NoC da plataforma HeMPS foi utilizada no trabalho de LANGE e BOHRER (2010). Apesar de algumas aplicações já estarem implementadas, o trabalho de LANGE e BOHRER (2010) foi o único encontrado que tem como foco o desenvolvimento de uma aplicação sobre o MPSoC HeMPS.

4 DESENVOLVIMENTO

Para o desenvolvimento das aplicações no MPSoC HeMPS, foram utilizados os seguintes paradigmas: mestre-escravo, *pipeline* e divisão e conquista. Nas subseções seguintes são descritas as aplicações - Multiplicação de Matrizes, Manipulação de Imagens e AES - implementadas em cada paradigma. Não foi possível implementar as aplicações no paradigma fases paralelas devido a limitações no MPSoC HeMPS que resultam em situações de *deadlock*.

Para cada implementação foram criadas modelagens que descrevem as principais funções das tarefas e a distribuição dos dados. As aplicações foram implementadas utilizando os recursos do MPSoC HeMPS descritos na subseção 2.1.2.2. Em todas as aplicações desenvolvidas, os dados de entrada são armazenados em vetores unidimensionais. A comunicação entre as tarefas é realizada através de troca de mensagens, utilizando as primitivas *Send*() e *Receive*().

A distribuição de dados é realizada através de uma função que determina a quantidade de dados que cada tarefa (*thread*) irá processar, de forma circular. Ou seja, se a quantidade de elementos paralelizados for 10, e a quantidade de *threads* disponíveis for 3, as *threads* 2 e 3 receberão 3 elementos e a *thread* 1 receberá 4 elementos. As tarefas escravas (paradigma mestre-escravo), as tarefas folhas (paradigma divisão e conquista) e os fluxos de execução (paradigma *pipeline*) representam as *threads* utilizadas.

Nos paradigmas mestre-escravo e divisão e conquista, a aplicação inicia e termina na mesma tarefa, e o recebimento dos resultados oriundos das tarefas escravas/filhas é realizado sempre na mesma ordem, ou seja, primeiro serão recebidos os dados da tarefa 1, depois da tarefa 2, e assim por diante. Mesmo que outra tarefa termine a execução primeiro, o MPSoC não permite essa dinamicidade.

No paradigma divisão e conquista foi estipulado que cada tarefa deve ter duas tarefas filhas. As tarefas folhas realizam o processamento, e as tarefas intermediárias (que estão entre a raiz e as tarefas folhas), dividem os dados recebidos pela tarefa raiz, e integram os dados recebidos pelas tarefas folhas. É importante ressaltar que, dependendo do número de *threads*, a árvore que representa a aplicação pode ficar em desequilíbrio (um lado maior do que o outro), e assim os dados podem ser divididos de maneira desigual entre as *threads*, o que pode impactar no desempenho da aplicação.

No paradigma *pipeline* as aplicações foram divididas em estágios, os quais foram organizados em fila, para cada nova *thread* são adicionados novos fluxos de execução na forma de *pipeline*. As aplicações sempre são finalizadas após o término da execução da última tarefa do *pipeline*.

Todas as aplicações implementadas neste trabalho encontram-se disponíveis em: <https://github.com/geaninne/TCC/Aplicacoes>.

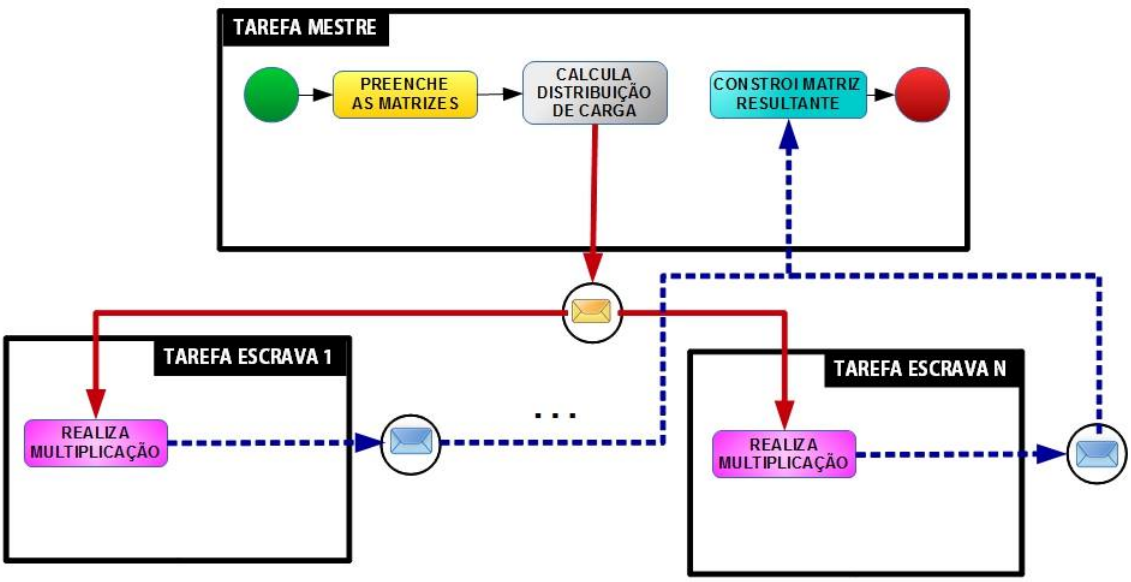
4.1 Multiplicação de Matrizes

Na implementação da aplicação Multiplicação de Matrizes, as matrizes de entrada A e B são quadradas, de ordem N e são preenchidas com elementos inteiros pelas tarefas da aplicação. A distribuição de carga é realizada sobre as linhas da matriz A, uma vez que todos os elementos de uma determinada linha devem ser enviados para uma mesma tarefa. A paralelização ocorre através da multiplicação realizada por diferentes tarefas escravas, onde cada tarefa escrava multiplica diferentes linhas de A por B.

4.1.1 Paradigma Mestre-Escravo

A Figura 20 apresenta uma visão geral da implementação da aplicação no paradigma mestre-escravo, onde são representadas a comunicação entre a tarefa mestre e as tarefas escravas e as suas principais funções. As setas indicam o fluxo da aplicação e os círculos verde e vermelho indicam o início e o fim da aplicação, respectivamente. A tarefa mestre preenche as matrizes, realiza a distribuição de carga sobre A, e constrói a matriz resultante. Enquanto que as tarefas escravas executam a multiplicação paralelamente. As mensagens indicam a transmissão de dados entre a tarefa mestre e as tarefas escravas.

Figura 20 – Modelagem da Aplicação Multiplicação de Matrizes no Paradigma Mestre - Escravo.

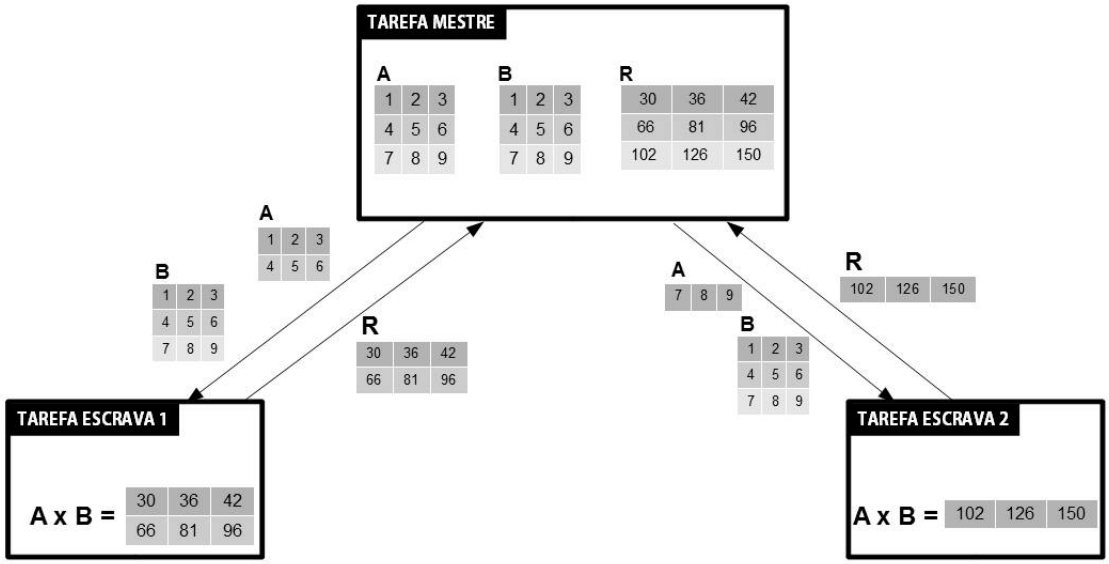


Fonte: Próprio Autor.

Na Figura 20, não foi representado o envio da matriz B, a mesma é transmitida, em sua totalidade, para as tarefas escravas, como demonstrado na Figura 21. A Figura 21

apresenta a distribuição dos dados entre as tarefas. Neste exemplo, a aplicação é paralelizada em duas tarefas escravas, e as matrizes de entrada possuem ordem 3. De acordo com o algoritmo de distribuição de carga aplicado, as duas primeiras linhas de A são enviadas para a tarefa escrava 1 e a última linha é enviada para a tarefa escrava 2. Então, após a multiplicação, as matrizes resultantes são enviadas para a tarefa mestre que constrói a matriz R final.

Figura 21 – Distribuição dos Dados da Aplicação Multiplicação de Matrizes no Paradigma Mestre- Escravo.



Fonte: Próprio Autor.

As funções utilizadas nesta implementação são descritas a seguir:

- **Preencher Matrizes**: Preenche as matrizes A e B com números inteiros. A função que calcula a distribuição de carga só inicia seu processamento após as matrizes serem completamente preenchidas;

- **Calcular a distribuição de carga**: Determina, de forma circular, as linhas da matriz A que devem ser atribuídas a cada tarefa escrava; e

- **Construir a matriz resultante**: Recebe os resultados oriundos das tarefas escravas e constrói a matriz resultante final.

- **Multiplicação**: Realiza a multiplicação de uma parcela de A por B.

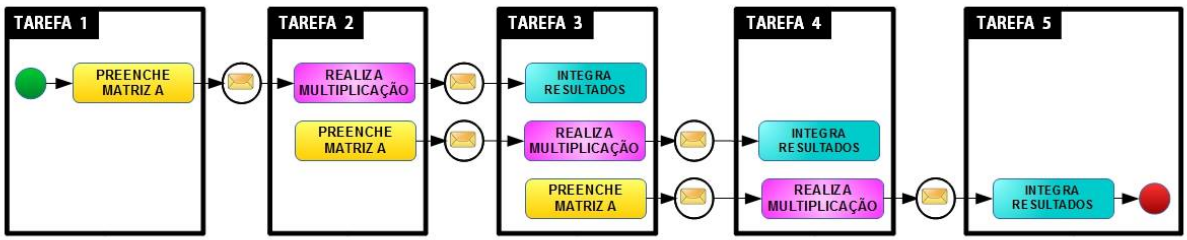
Os algoritmos pertinentes a esta aplicação são apresentados no Apêndice (**??**).

4.1.2 Paradigma *Pipeline*

Neste paradigma a aplicação é dividida nos seguintes estágios: preencher matriz A, multiplicar, e integrar resultados. Na Figura 22 é apresentada a modelagem desta implementação paralelizada em três *threads*, o que significa que cada fluxo de execução executa os estágios sobre um terço das linhas da matriz A. A seguir são descritas as funções utilizadas nesta implementação:

- **Preencher Matriz A**: Preenche, com valores inteiros, uma linha de A por vez, e a envia para a próxima tarefa.
- **Multiplicar**: Multiplica cada linha recebida por todos os elementos de B, e envia a linha resultante para a próxima tarefa.
- **Integrar Resultados**: Recebe os resultados oriundos da tarefa anterior e os integra em uma matriz resultante.

Figura 22 – Modelagem da aplicação Multiplicação de Matrizes no Paradigma *Pipeline*.



Fonte: Próprio Autor.

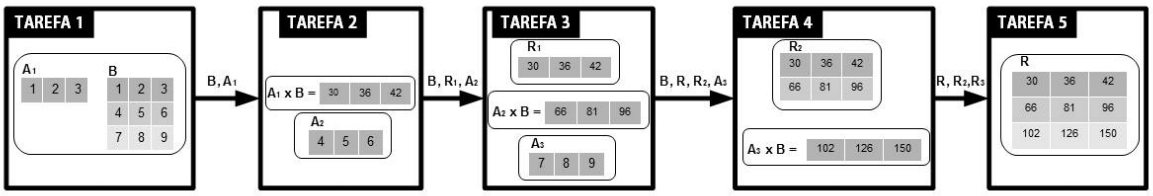
Na Figura 23 é demonstrada a distribuição dos dados, onde as matrizes de entrada possuem ordem 3. A matriz B é transmitida em sua totalidade, seguindo a ordem de fila, da tarefa 1 para a 4. As linhas da matriz A (A_1, A_2 e A_3), assim que preenchidas pelas tarefas 1, 2 e 3, respectivamente, são enviadas para a próxima tarefa. E a matriz R é construída em partes nas tarefas 3 e 4, e concluída na tarefa 5.

De acordo com (RAEDER et al., 2011), este paradigma não é apropriado para este tipo de aplicação, uma vez que cada fluxo é realizado quase que sequencialmente, e a aplicação possui apenas um estágio que demanda um grande poder de processamento.

4.1.3 Paradigma Divisão e Conquista

A Figura 24 apresenta a modelagem de uma implementação da aplicação no paradigma divisão e conquista, paralelizada em 4 *threads*. Primeiramente, são preenchidas as

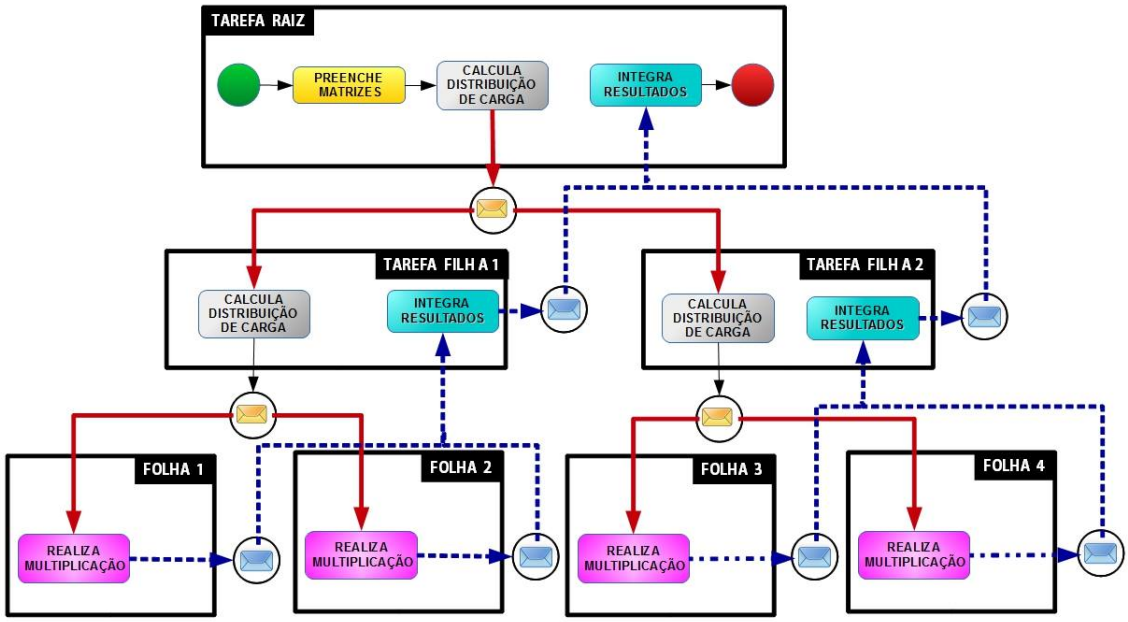
Figura 23 – Distribuição dos Dados da Aplicação Multiplicação de Matrizes no Paradigma *Pipeline*.



Fonte: Próprio Autor.

matrizes A e B pela tarefa raiz. Então a raiz realiza o ajuste de carga, e envia os dados para suas filhas. Este processo de divisão se repete, até que os dados cheguem em uma tarefa folha. As tarefas folhas realizam a multiplicação, e enviam o resultado de volta. O resultado é integrado até chegar na tarefa raiz que construirá a matriz resultante.

Figura 24 – Modelagem da Aplicação Multiplicação de Matrizes no Paradigma Divisão e Conquista.

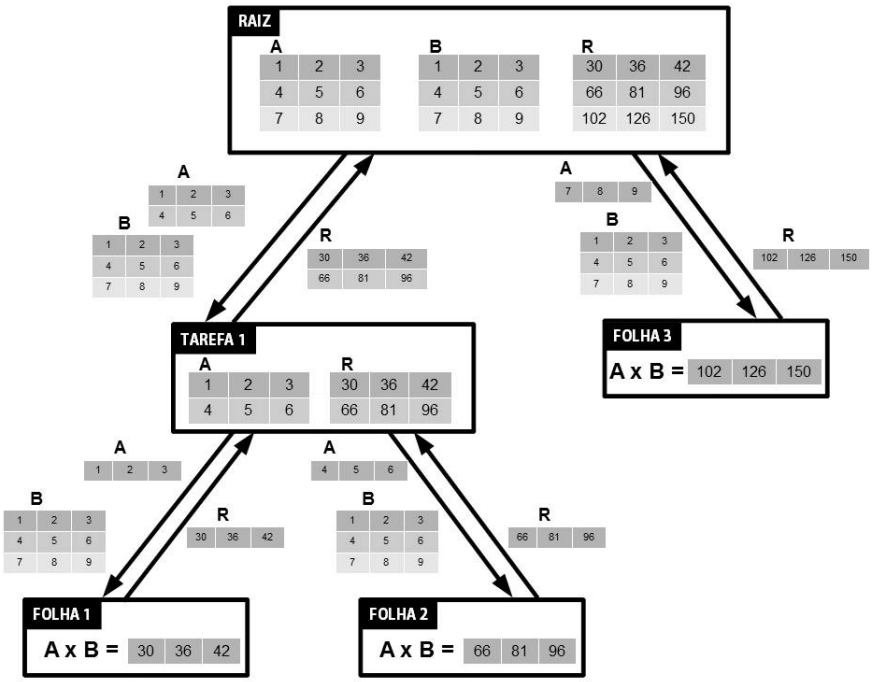


Fonte: Próprio Autor.

Um exemplo de distribuição de carga desta implementação é apresentado Figura 25. Neste exemplo a raiz realiza a distribuição de carga para as duas tarefas filhas,

tarefa 1 e tarefa folha 3. A tarefa 1 recebe os dados, realiza novamente a distribuição de carga, e envia os dados para as tarefas folhas 1 e 2 que realizam o processamento, enquanto que a tarefa folha 3 recebe os dados diretamente da raiz e multiplica os dados. Após ao processamento das folhas 1 e 2, a tarefa 1 integra os resultados e envia para a tarefa raiz. Então a tarefa raiz integra os dados recebidos e constrói a matriz R final.

Figura 25 – Distribuição de Dados da Aplicação Multiplicação de Matrizes no Paradigma Divisão e Conquista.



Fonte: Próprio Autor.

A funções contidas nesta implementação são semelhantes à implementação com base no mestre-escravo. Neste caso, o que difere as implementações é, essencialmente, a estrutura da aplicação.

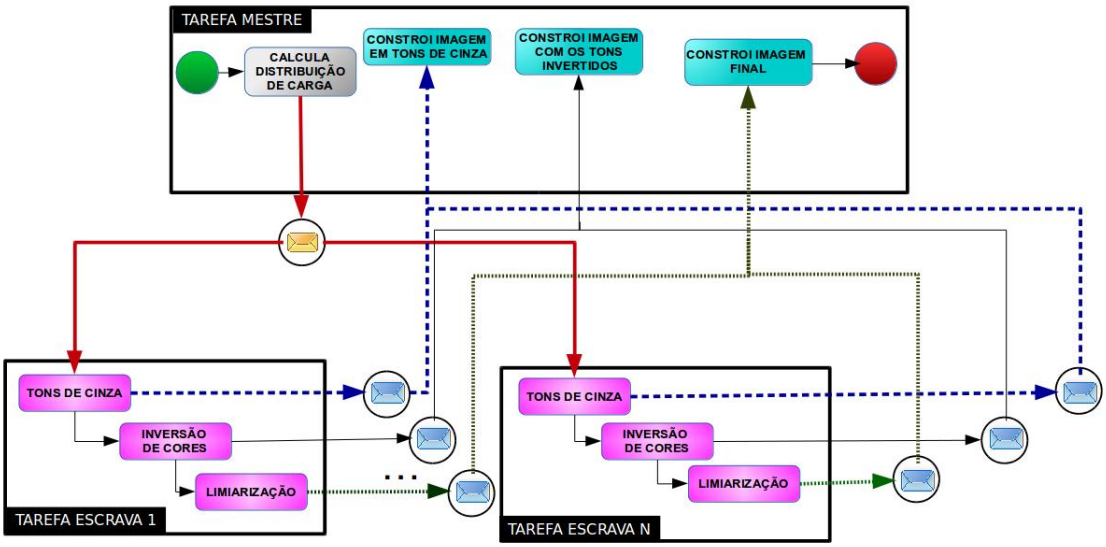
4.2 Manipulação de Imagens

Na implementação desta aplicação foi utilizado um vetor de entrada, contendo elementos do tipo *struct pixel*, estrutura que armazena 3 valores inteiros que representam os valores RGB de um pixel. A distribuição de carga é realizada sobre os pixeis da imagem, uma vez que os valores RGB de cada pixel devem ser enviados para uma mesma tarefa. A paralelização ocorre com a distribuição dos dados para diferentes *threads*, que realizam as transformações em uma parte do vetor.

4.2.1 Mestre-Escravo

A Figura 26 apresenta a modelagem da implementação da aplicação no paradigma mestre-escravo, onde a tarefa mestre realiza a distribuição de carga, e envia os valores RGB para as tarefas escravas. Então as tarefas escravas realizam cada transformação (tons de cinza, inversão e limiarização), de forma sequencial, e e enviam o resultado de cada uma para a tarefa mestre, que constrói a imagem resultante de cada transformação.

Figura 26 – Modelagem da Aplicação Manipulação de Imagens no Paradigma Mestre - Escravo.

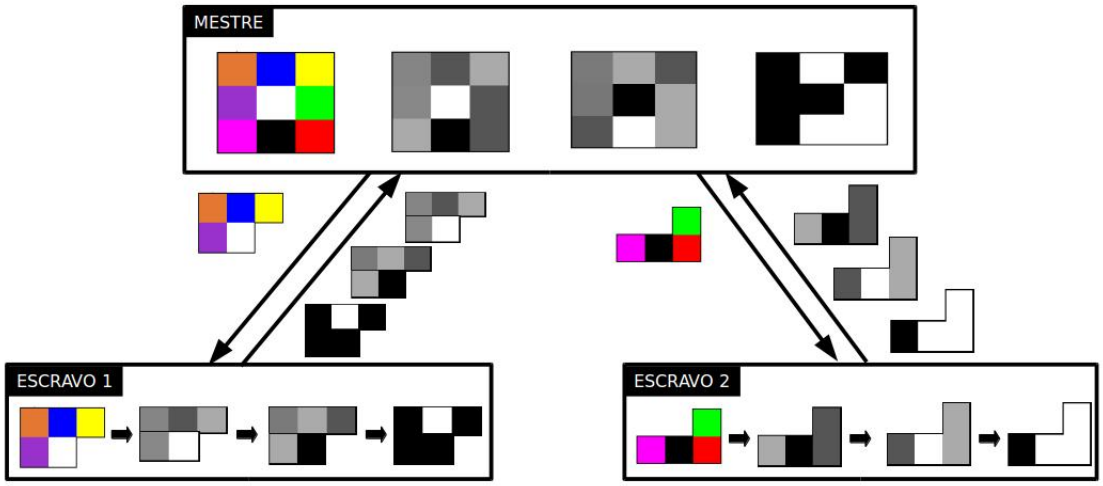


Fonte: Próprio Autor.

Na Figura 27 é apresentada a distribuição dos dados entre as tarefas. A distribuição de carga é realizada de acordo com o número de pixeis da imagem de entrada, uma vez que os valores de cada pixel devem ser enviados para uma mesma tarefa. Neste exemplo, a aplicação é paralelizada em duas tarefas escravas, e a imagem de entrada possui 9 pixeis. De forma que, 5 pixeis da imagem de entrada são enviados para a tarefa escrava 1 e os outros 4 são enviados para a tarefa escrava 2. Logo, são enviados 15 valores RGB ao total para a tarefa escrava 1 e 12 valores RGB para a tarefa escrava 2. Após cada transformação os pixeis resultantes são enviados para a tarefa mestre, que constrói uma imagem para cada transformação.

Como pode ser observado na Figura 26, o volume de comunicação entre a tarefa mestre e as tarefas escravas é maior do que a aplicação multiplicação de matrizes. No entanto como visto na subseção 2.3.2 a complexidade desta aplicação é menor do que aplicação multiplicação de matrizes. Assim, pode-se presumir que o desempenho será

Figura 27 – Distribuição dos Dados da Aplicação Manipulação de Imagens no Paradigma Mestre - Escravo.



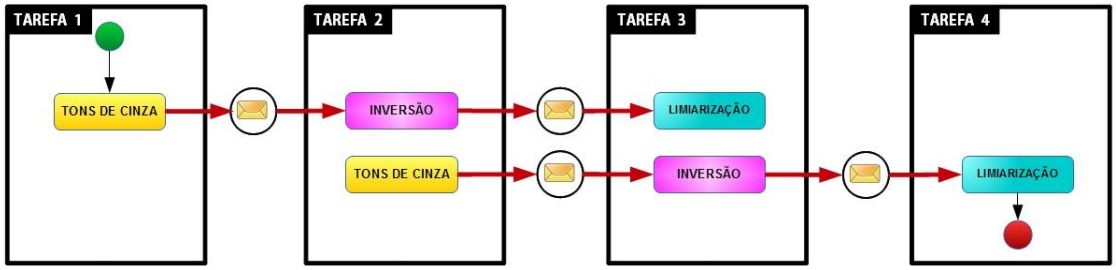
Fonte: Próprio Autor.

mais dependente da comunicação do que do processamento em si.

4.2.2 Paradigma Pipeline

No paradigma *pipeline* as funções de cada tarefa foram dividas de acordo com os estágios da aplicação. A Figura 29 apresenta a modelagem desta aplicação contendo 4 tarefas. Neste caso, os dados do vetor de entrada são divididos em duas *threads*. As transformações da primeira *thread* são realizadas pelas tarefas 1, 2 e 3, e as transformações da segunda são realizadas pelas tarefas 2, 3 e 4.

Figura 28 – Modelagem da Aplicação Manipulação de Imagens no Paradigma *Pipeline*.

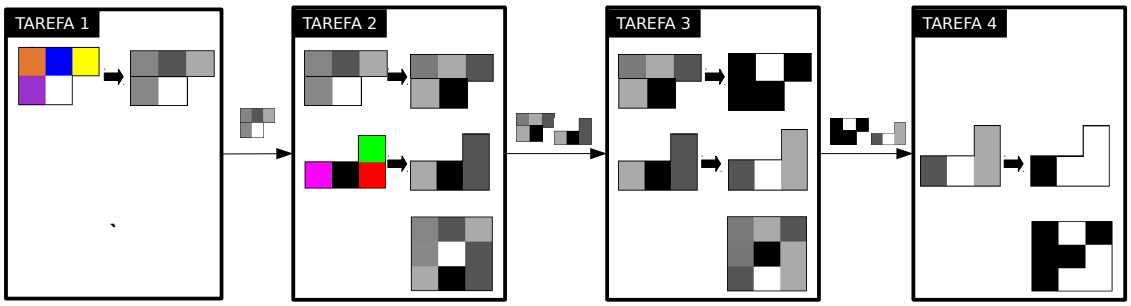


Fonte: Próprio Autor.

A Figura 28 apresenta a distribuição dos dados. Na primeira tarefa é realizada a transformação tons de cinza sobre os primeiros 5 pixeis da imagem. A tarefa 2 recebe os pixeis alterados, realiza a inversão sobre os mesmos e envia o resultado para a tarefa 3.

Após, a tarefa 2 ainda realiza a transformação para tons de cinza nos últimos 4 pixeis da imagem e constrói a imagem resultante desta transformação. A tarefa 3 transforma a primeira parte da imagem para preto e branco, realiza a transformação de inversão na segunda parte, envia os pixeis resultantes para a próxima tarefa, e constrói a imagem invertida. A ultima tarefa recebe transforma a segunda parte da imagem para preto e branco e monta a imagem final.

Figura 29 – Distribuição dos Dados da Aplicação Manipulação de Imagens no Paradigma Pipeline.



Fonte: Próprio Autor.

4.2.3 Paradigma Divisão e Conquista

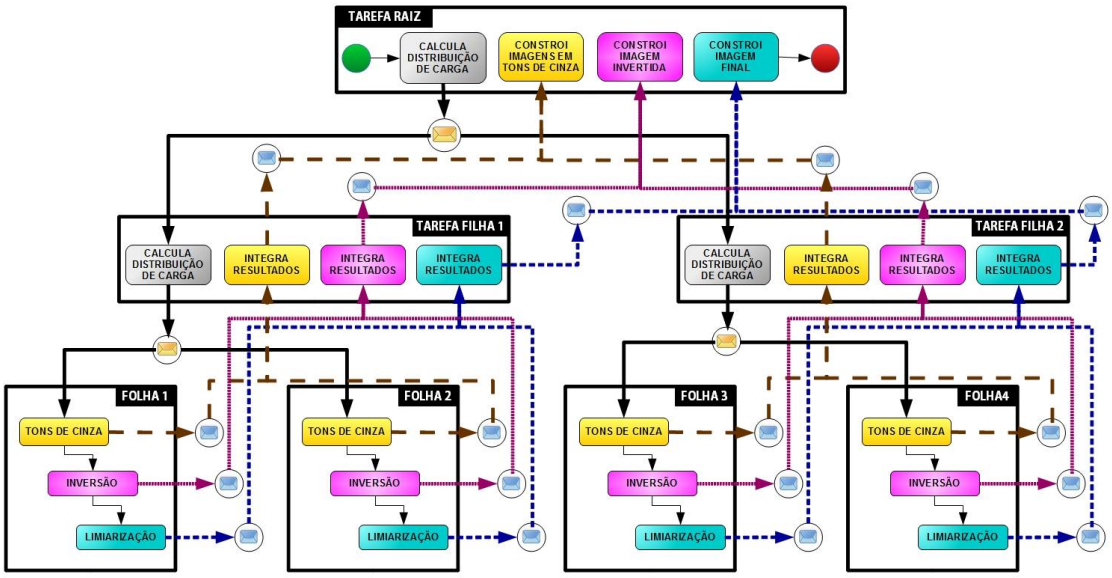
A modelagem da aplicação Manipulação de Imagens neste paradigma é apresentada na Figura 30. Neste exemplo, a aplicação é paralelizada em 4 tarefas folhas. A aplicação inicia na tarefa raiz que realiza a distribuição de carga e envia os pixeis para as tarefas filhas, que realizam a distribuição de carga novamente sobre os pixeis recebidos. As tarefas folhas recebem os pixeis, realizam as transformações nas imagens, e enviam os valores RGB de volta para as tarefas, que integram o resultado até chegar na tarefa raiz.

A Figura 31 apresenta um exemplo de distribuição de carga. Os pixeis são repartidos entre as tarefas filhas, até chegarem nas tarefas folhas. Observe que, como a árvore formada está desequilibrada, a tarefa folha 3 recebe uma carga maior (4 pixeis) do que as demais (tarefa folha 1 - 3 pixeis e tarefa folha 2 - 2 pixeis). Após o processamento de cada transformação os dados são enviados de volta e integrados pela tarefa 1 e pela tarefa raiz. As funções contidas nesta implementação são semelhantes à implementação com base no mestre-escravo.

4.3 AES

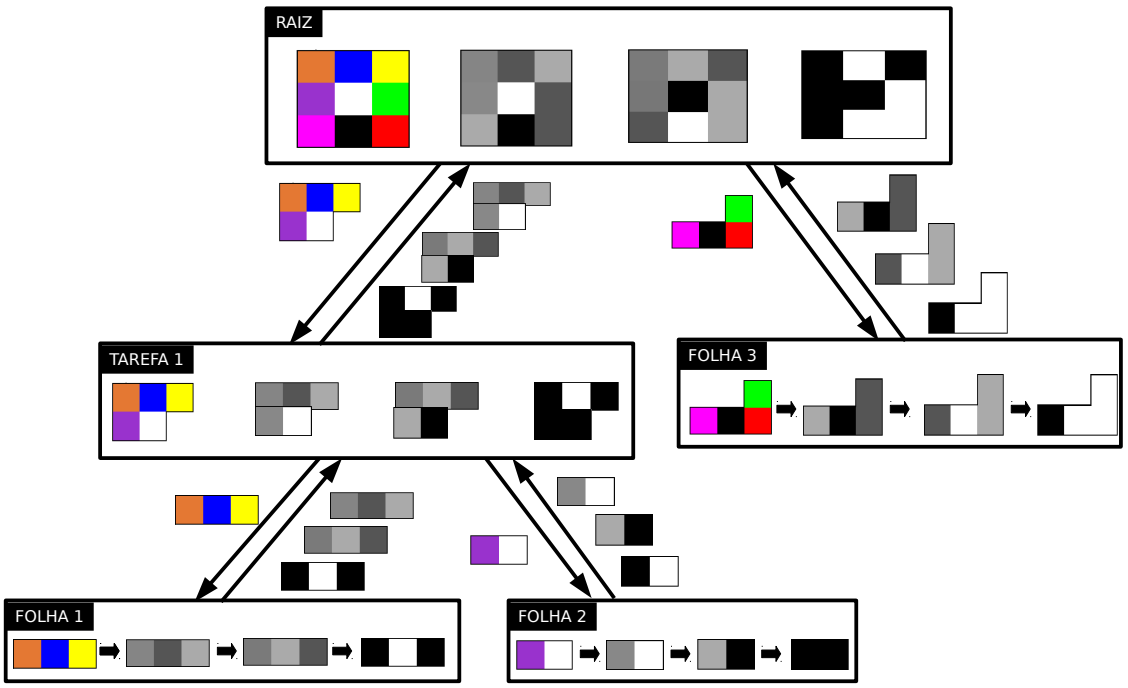
O desenvolvimento desta aplicação foi baseada no trabalho de (REZENDE; CAIMI, 2013), que implementou a aplicação no paradigma mestre-escravo. A aplicação tem como

Figura 30 – Modelagem da Aplicação Manipulação de Imagens no Paradigma Divisão e Conquista.



Fonte: Próprio Autor.

Figura 31 – Distribuição de Dados da Aplicação Manipulação de Imagens no Paradigma Divisão e Conquista.



Fonte: Próprio Autor.

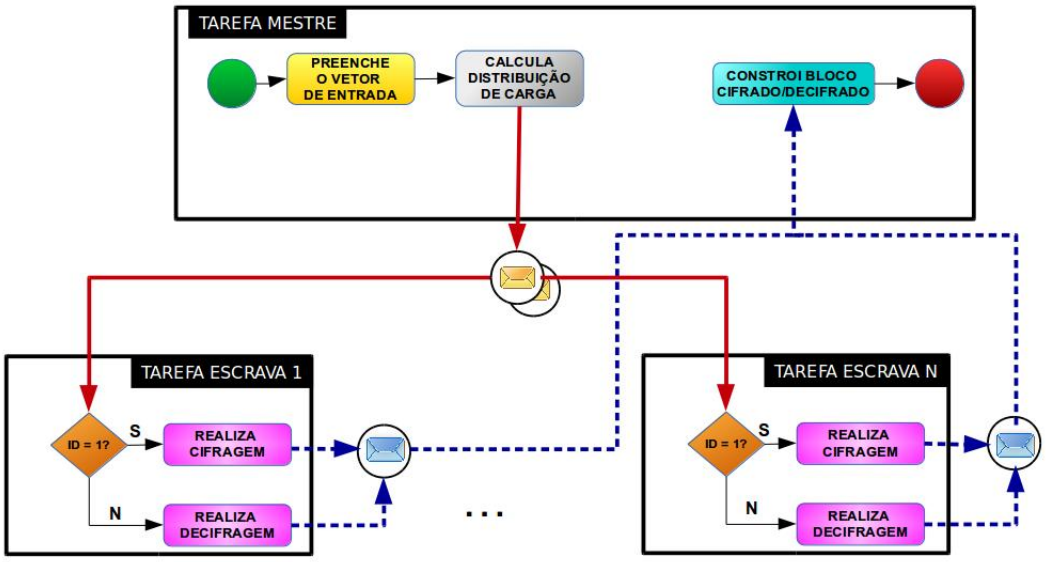
entrada uma mensagem que deve ser cifrada, esta mensagem é armazenada em um vetor de tamanho N, sendo N múltiplo de 128. O vetor de entrada é contido por uma sequência de letras, que se repetem 16 vezes. A chave de encriptação possui 256 bits.

A distribuição de carga é realizada da mesma maneira que as demais aplicações. Sendo que, neste caso, os dados são divididos em blocos de 16 bytes, e a distribuição de carga é realizada de acordo com o número de blocos.

4.3.1 Mestre-Escravo

A Figura 32 apresenta a modelagem desta implementação, onde primeiramente, a tarefa mestre preenche a mensagem que será cifrada ou decifrada, calcula a distribuição de carga e envia os dados para as tarefas escravas. Enquanto a tarefa mestre aguarda o resultado, as tarefas escravas recebem as mensagens, verificam se deve ser realizada uma operação de cifragem ou de decifragem, e executam a operação. Por fim, a tarefa mestre constrói o bloco cifrado/decifrado.

Figura 32 – Modelagem da Aplicação AES no Paradigma Mestre - Escravo.



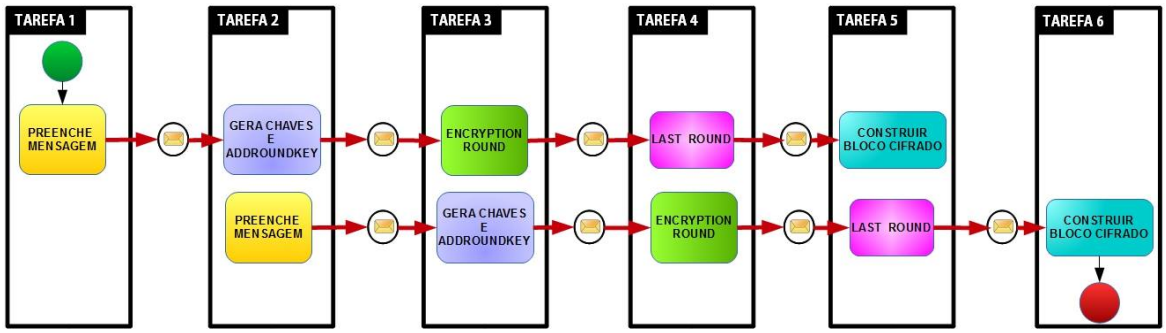
Fonte: Próprio Autor.

Diferentemente das demais aplicações implementadas neste paradigma, a operação de cifragem/decifragem é realizada em cada mensagem por vez, ou seja, para duas mensagens as tarefas escravas recebem a primeira mensagem, realizam a operação de cifragem, enviam o resultado e só então recebem a segunda mensagem e realizam o procedimento novamente.

4.3.2 Paradigma Pipeline

De acordo com o descrito na subseção 2.3.3, esta aplicação inicia com o preenchimento da mensagem, gera as sub-chaves, executa uma vez a função $AddRoundKey$, e inicializa as rodadas de encriptação, sendo a última rodada diferente das demais. Então, para a implementação do AES no paradigma *pipeline*, os estágios foram divididos conforme mostra a Figura 33, onde são demonstrados dois fluxos de execução. Cada fluxo de execução inicia com uma tarefa preenchendo a mensagem de entrada, então o bloco preenchido é enviado para uma segunda tarefa que gera as subchaves, realiza o método $AddRoundKey$, e envia o bloco gerado por este método para a próxima tarefa que inicializa a encriptação sobre o bloco. A encriptação é o estágio mais longo desta aplicação, e abrange os métodos $SubBytes$, $ShiftRows$, $MixColumns$ e $AddRoundKey$, executados em sequência em um *loop* que itera 13 vezes, para uma chave de 256 bits. Após, o resultado é enviado para a próxima tarefa que realiza a última rodada da encriptação, a qual engloba os métodos $SubBytes$, $ShiftRows$ e $AddRoundKey$, então o resultado é enviado para a última tarefa que constrói o bloco cifrado resultante.

Figura 33 – Modelagem da Aplicação AES no Paradigma *Pipeline*.



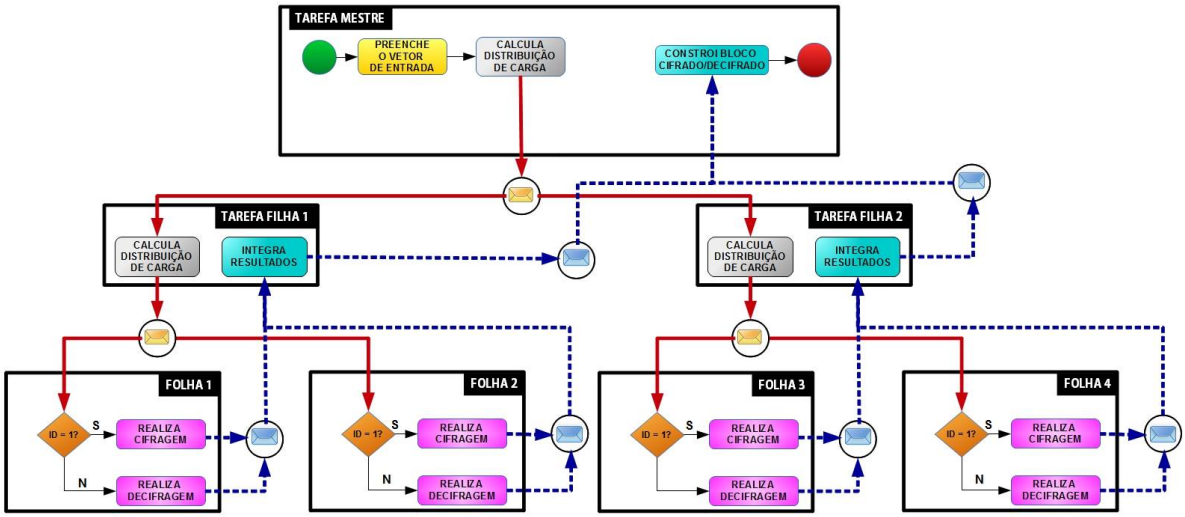
Fonte: Próprio Autor.

4.3.3 Paradigma Divisão e Conquista

Tal como as implementações anteriores baseadas no paradigma divisão e conquista, nesta implementação cada tarefa pode ter no máximo duas tarefas filhas. A mensagem de entrada é preenchida pela tarefa inicial, e a cifragem ou decifragem é realizada pelas tarefas folhas. As demais tarefas dividem os dados e integram os resultados.A Figura 34, apresenta a modelagem desta implementação com 7 tarefas disponíveis. O início e término da aplicação, indicados pelos círculos verde e vermelho, ocorrem na tarefa inicial.

As funções de preenchimento da mensagem, cálculo de distribuição de carga, integração dos resultados e cifragem ou decifragem são as mesmas da implementação no paradigma mestre-escravo. Contudo, a organização da aplicação difere, como pode ser

Figura 34 – Modelagem da Aplicação AES no Paradigma Divisão e Conquista.



Fonte: Próprio Autor.

observado na Figura 34. Outro ponto que pode ser observado como uma diferença, é que para 7 tarefas, no paradigma mestre-escravo, a entrada é dividida em 6 partes, enquanto que o mesmo número de tarefas em divisão e conquista, divide a entrada em 4 partes.

O conjunto de aplicações descrito nesta seção foi implementado em nível de software na plataforma HeMPS, utilizando a linguagem C. Um arquivo foi criado para cada tarefa apresentada. Além disso, foram criados arquivos de configuração da aplicação e da plataforma.

Conforme visto na seção 2.2, cada paradigma apresenta uma organização diferente. Deste modo, o número de tarefas necessárias para realizar uma paralelização pode divergir, dependendo do paradigma utilizado. Por exemplo, no paradigma mestre - escravo, o número de tarefas sempre será igual número de escravos mais um (tarefa mestre). Ao passo que no paradigma divisão e conquista este número dependerá do máximo de filhos que cada tarefa pode ter, por exemplo se o máximo de filhos for 2, para executar 4 *threads* (paralelizações), serão necessárias 7 tarefas ao total. No paradigma *pipeline*, o número de tarefas também depende do número de estágios da aplicação. A Tabela 3 apresenta, para cada aplicação, o número de tarefas que devem ser utilizadas, para uma determinada quantidade de *threads*, em cada paradigma.

O número de tarefas utilizados pode impactar no desempenho, consumo energético e área utilizada do MPSoC. Como as tarefas podem ser alocadas em PEs distintos, o tempo para comunicação, a energia consumida pela NoC e a área utilizada serão maiores. Assim, o número de tarefas utilizadas deve ser justificado pelo desempenho obtido no processamento paralelo realizado.

Tabela 3 – Número de Tarefas para Cada Paradigma

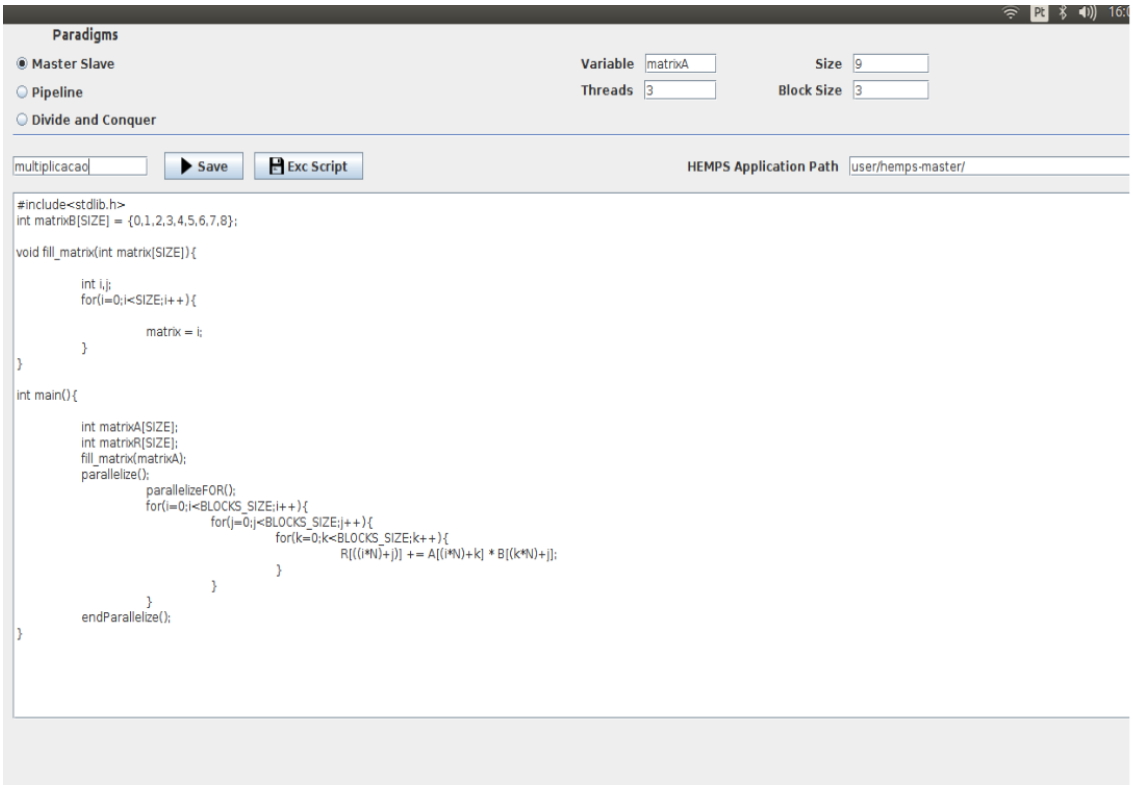
THREADS	**Multiplicação**			**Manip. de Imagens**			**AES**		
	ME	**DC**	**PP**	**ME**	**DC**	**PP**	**ME**	**DC**	**PP**
2	3	3	4	3	3	4	3	3	6
3	4	5	5	4	5	5	4	5	7
4	5	7	6	5	7	6	5	7	8
5	6	9	7	6	9	7	6	9	9
6	7	11	8	7	11	8	7	11	10
7	8	13	9	8	13	9	8	13	11
8	9	15	10	9	15	10	9	15	12
9	10	17	11	10	17	11	10	17	13
10	11	19	12	11	19	12	11	19	14

5 HEMPS PARALLEL PROGRAMMING

A programação no MPSoC HeMPS demanda um grande esforço, pois devem ser criados, manualmente, arquivos para cada tarefa e arquivos de configuração. O desenvolvedor consome tempo com trivialidades como replicação de tarefas e funções, e demora na depuração de problemas. As aplicações criadas não seguem um padrão e pode ocorrer falta de otimização dos recursos. Neste sentido, foi elaborada uma ferramenta com o objetivo de solucionar ou pelo menos mitigar estes problemas, realizando a criação das tarefas, e adicionando ao código as funções de divisão dos dados, e transmissão de mensagens.

A Figura 35 mostra a interface gráfica da HeMPS *Parallel Programming*, implementada na linguagem Java. Na área central, o usuário deve inserir o código sequencial da aplicação, juntamente com as funções descritas na Tabela 4. Na parte superior da ferramenta, o usuário deve indicar informações relevantes para a paralelização, as quais são descritas na Tabela 5. Por fim, clicando no botão *Save* (para salvar o arquivo), e posteriormente no botão *Execute*, o projeto da aplicação é gerado, através de códigos implementados em *ShellScript* que percorrem o caminho informado pelo usuário no $HeMPSApplicationPath$, e está pronto para ser simulado.

Figura 35 – HeMPS *Parallel Programming*.



Fonte: Próprio Autor.

Tabela 4 – API Desenvolvida.

Método e Descrição
parallelize() Inicia o trecho de código paralelização, e o código contido entre *parallelize* e *endParallelize* é replicado para as demais tarefas de acordo com o paradigma escolhido e número de *threads* utilizadas.
parallelizeFOR() O laço de repetição for é alterado, e percorre apenas a quantidade de elementos da tarefa escrava em questão
endParallelize() Finaliza o trecho de código paralelizado.

Tabela 5 – Parâmetros Inseridos pelo Usuário.

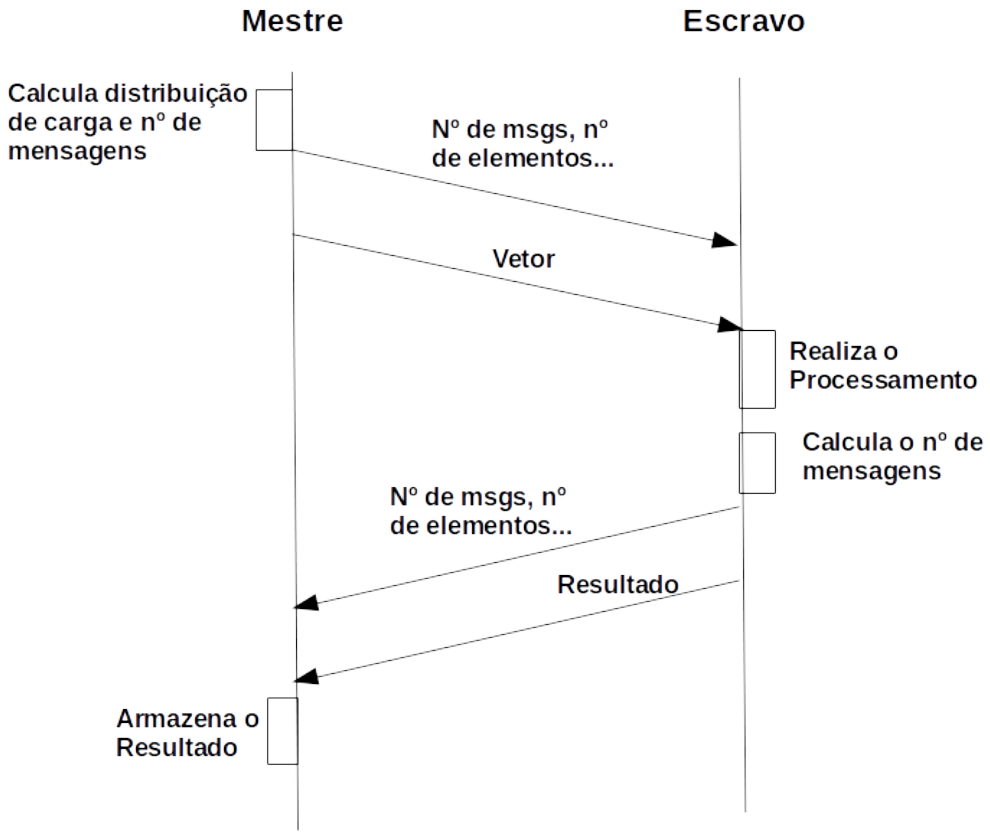
Parâmetro	**Descrição**
Paradigms	Deve ser indicado, em uma lista de seleção, o paradigma que deverá ser utilizado (mestre-escravo, divisão e conquista ou *pipeline*). Atualmente, apenas a ferramenta provê apenas a paralelização no paradigma mestre-escravo, no entanto em trabalhos futuros poderá prover a paralelização para os demais paradigmas
Variable	Deve ser inserido, no campo de texto indicado, o nome da variável que será paralelizada.
Threads	O número de threads utilizadas.
SIZE	Quantidade total de elementos da variável.
BLOCK_SIZE	Tamanho de um bloco, que não pode ser separado para tarefas diferentes.
HeMPS PATH	Caminho da HeMPS no computador do usuário.

No exemplo apresentado na Figura 35 é descrita uma aplicação de multiplicação de matrizes quadradas de ordem 3, que deve ser paralelizada no paradigma mestre-escravo, utilizando 3 tarefas escravas. Primeiramente é criada a biblioteca do projeto, que armazena as constantes $SIZE$, $BLOCK\_SIZE$ e $NUM\_THREADS$, passadas pelo usuário. A biblioteca também armazena a variável $matrixB$, uma vez que a mesma foi declarada como global na aplicação em sequencial. O trecho de código contido entre o início da função *main* e a chamada da função *parallelize*, é copiado para o arquivo que corresponde a tarefa mestre. Adicionalmente o código da tarefa mestre é composto pelas funções de distribuição de carga, cálculo, envio e recebimento de mensagens, armazenamento do resultado final e finalização da aplicação. Nas tarefas escravas, primeiramente são adicionados: a função de recebimento de mensagens e o trecho de código descrito entre as funções *parallelize* e *endParallelize*. Então, é realizado um rastreamento nos arquivos, verificando se a função $paralellizeFOR$ é chamada, e caso seja, o comando for

indicado é alterado em cada tarefa escrava para o valor correspondente a distribuição de carga. As tarefas escravas também compõem funções de cálculo e envio de mensagens.

A comunicação entre a tarefa mestre e as tarefas escravas é demonstrada na Figura 36, onde primeiramente é calculada a distribuição de carga e o número de mensagens necessárias para cada tarefa escrava. Então, informações como o número de mensagens e quantidade de elementos são enviados para as tarefas escravas. Após, é enviado cada parcela do vetor para a tarefa escrava correspondente. Então as tarefas escravas realizam o processamento e calculam o número de mensagens a serem enviadas e a quantidade de elementos do vetor resultante, e envia estas informações para a tarefa mestre. Por fim, envia o resultado para a tarefa mestre, que armazena o mesmo em um único vetor.

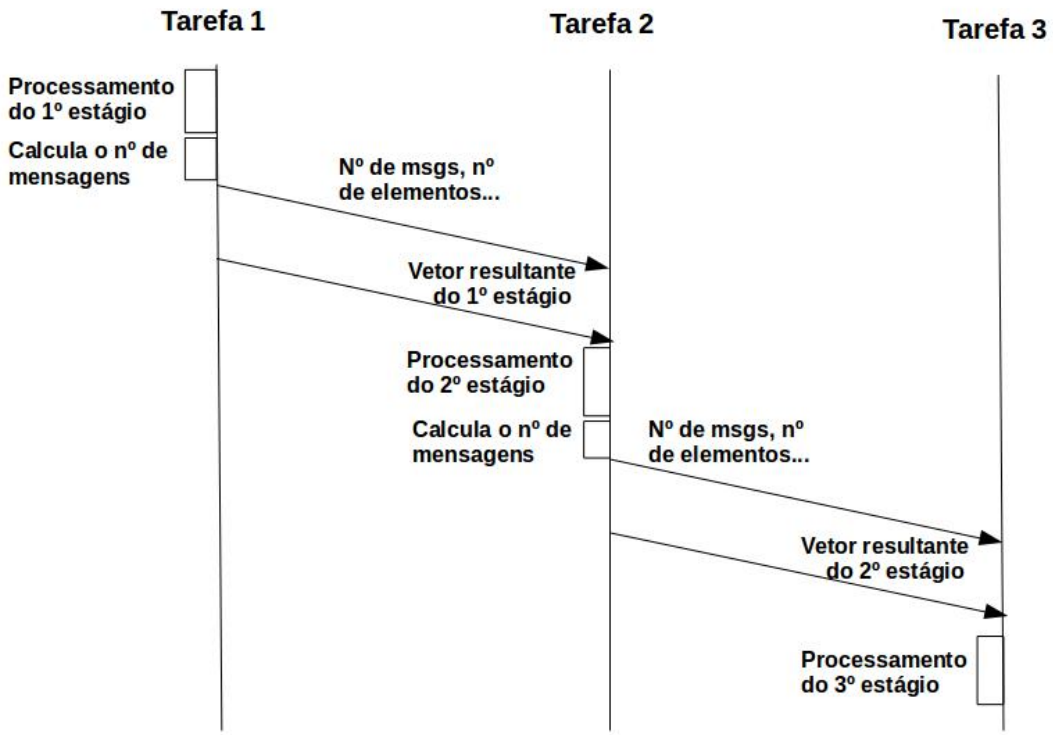
Figura 36 – Comunicação entre Mestre e Escravo.



Fonte: Próprio Autor.

No paradigma *pipeline*, os estágios são divididos de acordo com as funções existentes entre as diretivas *parallelize*, e *endParallelize*, sendo que o primeiro estágio corresponde a primeira parte da aplicação e o último armazena os resultados. Na Figura 37 é apresentada a comunicação entre as tarefas, onde a primeira tarefa processa o primeiro estágio, calcula o número de mensagens e envia o número de mensagens e o resultado do primeiro processamento para a próxima tarefa. A tarefa 2 repete os mesmos passos, e envia para a tarefa 3, a qual tem como estágio o armazenamento do vetor resultante.

Figura 37 – Comunicação no Paradigma Pipeline.

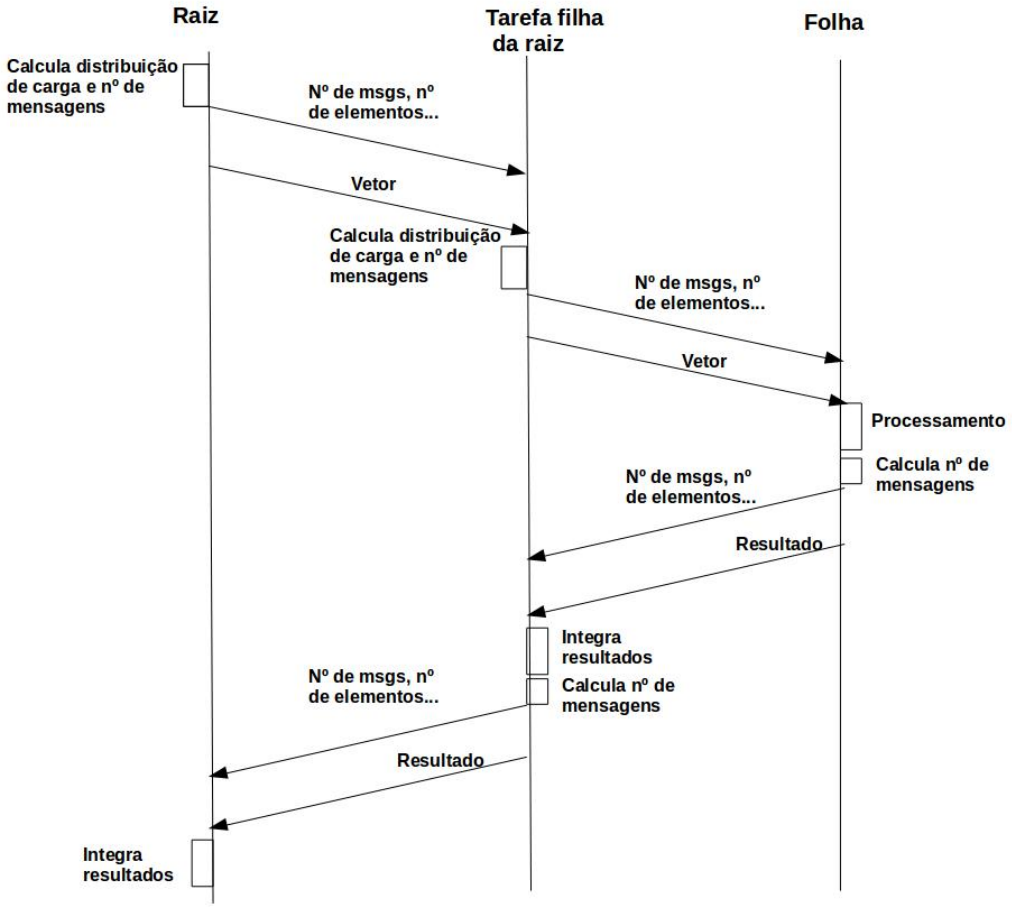


Fonte: Próprio Autor.

Para o paradigma divisão e conquista foi estabelecido um número máximo de filhos igual a 2, sendo que a árvore que forma a aplicação é preenchida da esquerda para a direita. A comunicação no paradigma divisão e conquista é demonstrada na Figura 38. A

distribuição de carga é realizada pela tarefa raiz, que envia as mensagens para as tarefas filhas. As tarefas filhas realizam o mesmo procedimento até chegar na tarefa folha, a qual realiza o processamento e envia de volta para a sua tarefa pai. Os resultados são integrados até chegar na tarefa raiz.

Figura 38 – Comunicação no Paradigma Divisão e Conquista.



Fonte: Próprio Autor.

A ferramenta criada apresenta algumas limitações, como prover a paralelização apenas de estruturas unidimesionais, e ainda está em processo de desenvolvimento. No entanto, a sua primeira versão já garante um menor tempo de desenvolvimento, e um número bastante reduzido de linhas de código.

6 RESULTADOS

Neste Capítulo são apresentados os resultados das simulações das três aplicações (Multiplicação de Matrizes, Manipulação de Imagens e AES) modeladas nos paradigmas Mestre-Escravo, Pipeline e Divisão e Conquista, assim como a análise e comparação destes resultados. A configuração do MPSoC HeMPS utilizada nas simulações é apresentada na seção 6.1. Os critérios utilizados na análise são descritos na seção 6.2. Na seção 6.3 são apresentados os resultados obtidos, juntamente com a análise e comparação destes.

6.1 Configuração da Plataforma

O MPSoC HeMPS utilizado nas simulações realizadas possui as seguintes características:

- Dimensão: 6 x 6, sendo 1 PE mestre e 35 PEs escravos;
- Número de clusters: 1;
- Tamanho da página dos PEs: 256 KB;
- Número de tarefas por PE: 1;
- Tamanho dos *buffers* da NoC: 8;
- Algoritmo de Mapeamento: estático (não considera a comunicação entre as tarefas);
- Algoritmo de Roteamento: XY;

6.2 Critérios de Avaliação

Os critérios de avaliação utilizados foram o tempo de execução e o consumo energético dos processadores plasma.

6.2.1 Tempo de Execução

O tempo de execução é o tempo necessário para concluir um caso de teste, expresso em milissegundos (ms). Para capturar este tempo, o usuário deve inserir no código da aplicação a função $GetTick()$, que retorna o valor do relógio atual. Assim, com a função $GetTick()$ inserida no começo e no fim da aplicação, o tempo de execução pode ser obtido através da subtração do tempo inicial do tempo final e esse resultado dividido por 100000, a fim de obter o tempo expresso em milissegundos.

6.2.2 Consumo Energético nos Processadores

No trabalho de Filho et al. (2012), verificou-se que instruções de uma dada categoria (lógica, saltos, aritmética, load/store, etc) possuem consumo energético similar. Então, as instruções executadas pelo Plasma, em uma frequência de $100MHz$ e para uma tecnologia de $65nm$, foram categorizadas conforme a Tabela 6, onde o consumo para cada categoria esta expresso em miliwatt.

Tabela 6 – Dissipação de potência das diferentes categorias de instruções do processador Plasma (frequência do processador: 100 MHz, tecnologia 65 nm), adaptado de (GUINDANI, 2014).

Categoria	**Instruções**	**Dissipação de Potência Média (mW)**
Aritmética	MULT, MULTU, DIV,ADD, ADDU, SUB, SUBU, SLT, SLTU, DADDU, ADDI, ADDIU,SLTI, e SLTIU	5,099
Saltos	JR, JALR, BLTZAL, BLTZ, BGEZ, BLTZALL, BLTZL, BGEZALL, BGEZL, JAL, J, BEQ, BNE, BLEZ, BGTZ, BEQL, BNEL, BLEZL e BGTZL	5,869
Load/Store	LB, LH, LW, LBU, LHU, SB, SH, SW, LL e SC	3,994
Lógica	AND, OR, XOR, NOR, ANDI, ORI, XORI e LUI	4,363
Move	MOVZ, MOVN, MFHI, MTHI, MFL0, MTL0 e COP0	3,143
Shift	SLL, SRL, SRA, SLLV, SRLV e SRAV	3,824
Demais	SYSCALL, BREAK, SYNC, TGEU, TLT, TLTU, TEQ, TNE, LWL, LWR, SWL, SWR, e SWC1	5,099

Para obter o consumo energético (potência dissipada) o número de cada categoria é contabilizado, separando as instruções do *kernel* e da tarefa. Então é realizada a multiplicação entre o total de instruções de determinada categoria pelo seu consumo energético, estipulado na Tabela 6. Assim é obtido o consumo tanto para as tarefas, como para o *kernel*. O total de energia consumido pelo processador é resultante da soma dos consumos energéticos do *kernel* e pelas tarefas. O modelo de consumo de energia do processador não avalia o consumo de acesso à memória, e somente o consumo devido a execução das instruções (GUINDANI, 2014).

6.3 Análise

Foram realizadas simulações das aplicações descritas, variando o tamanho de entrada e número de *threads*. Na subseção 6.3.1 são apresentados os resultados das simulações da aplicação Multiplicação de Matrizes para cada paradigma.

6.3.1 Multiplicação de Matrizes

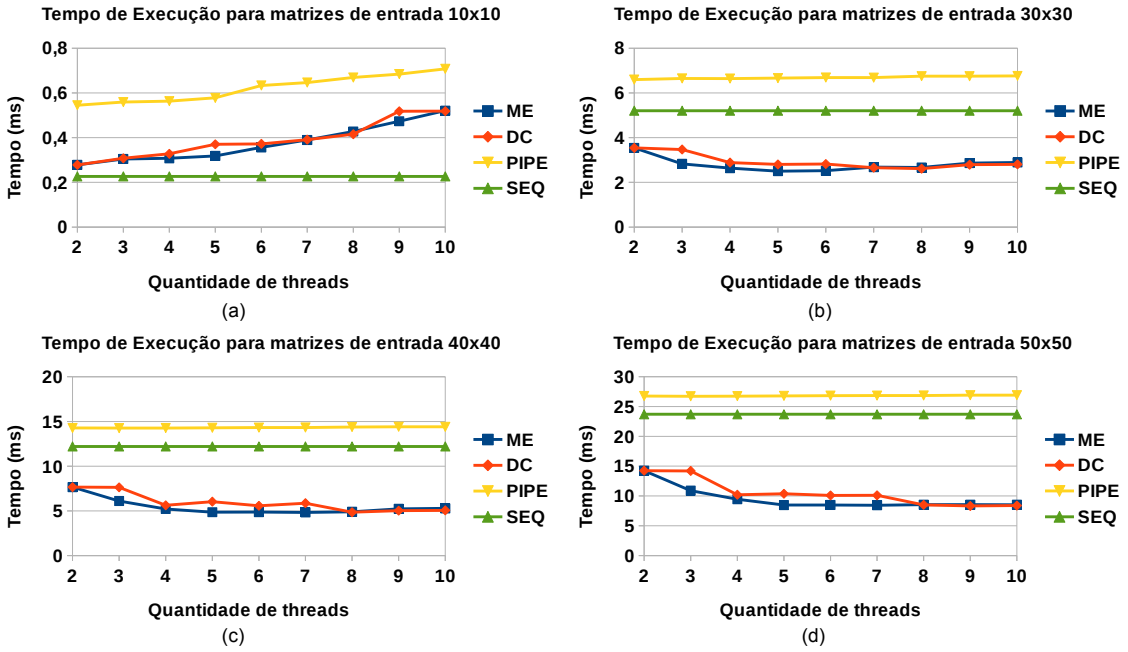
Nos casos de teste realizados para esta aplicação, o número de *threads* variou entre 2 e 10 e as dimensões utilizadas para as matrizes de entrada, A e B, foram: 10x10, 20x20, 30x30, 40x40 e 50x50. O MPSoC não suportou simulações para tamanhos maiores de matrizes de entrada.

Na Figura 39 são apresentados gráficos contendo os tempos de execução para diferentes dimensões de matrizes, onde no eixo x é apresentada a quantidade de *threads* e no eixo y o tempo obtido em milissegundos. A linha na cor verde mostra os tempos de execução para a aplicação na versão sequencial (SEQ). A linha na cor azul mostra os tempos para a aplicação implementada no paradigma mestre-escravo (ME). A linha na cor vermelha mostra os tempos para a aplicação implementada no paradigma divisão e conquista (DC) e a linha na cor amarela mostra os tempos para a aplicação implementada no paradigma *pipeline* (PIPE).

Pode ser observado no gráfico apresentado na Figura 39(a), que nenhum paradigma de programação paralela obteve ganho de desempenho em relação a implementação na versão sequencial para matrizes de ordem 10. Isto ocorre por causa do número baixo de elementos processados e o consequente pequeno tempo de execução. Assim, não há elementos suficientes que justifiquem a paralelização. Nas simulações realizadas com matrizes de ordem 30, (Figura 39(b)) e 40 (Figura 39(c)) houve ganho de desempenho com até 5 *threads* no paradigma mestre-escravo e com até 8 *threads* no paradigma divisão e conquista. Enquanto que para matrizes de entrada de ordem 50, houve ganho de desempenho utilizando até 7 *threads* no paradigma mestre-escravo e 9 *threads* no paradigma divisão e conquista. Enquanto o paradigma *pipeline* obteve um tempo de execução maior, em relação a versão sequencial, para todos os tamanhos de entrada, independente do número de *threads*.

Na maioria dos casos, o paradigma mestre-escravo apresentou melhor desempenho do que o divisão e conquista para as paralelizações em 3, 4, 5, 6 e 7 *threads*. Em contrapartida, para as paralelizações com 8, 9 e 10 *threads*, o paradigma divisão e conquista executou a aplicação em um tempo de execução menor, na maioria dos casos. Observa-se nos gráficos da Figura 39, que existe pouca diferença entre os tempos dos paradigmas mestre-escravo e divisão e conquista. Contudo com o uso de 4 e 8 *threads*, o tempo de execução do paradigma divisão e conquista reduziu consideravelmente. Isto ocorre pois com 4 e 8 *threads* a árvore formada fica balanceada e, consequentemente, os dados são

Figura 39 – Tempo de Execução para a Aplicação Multiplicação de Matrizes.



Fonte: Próprio Autor.

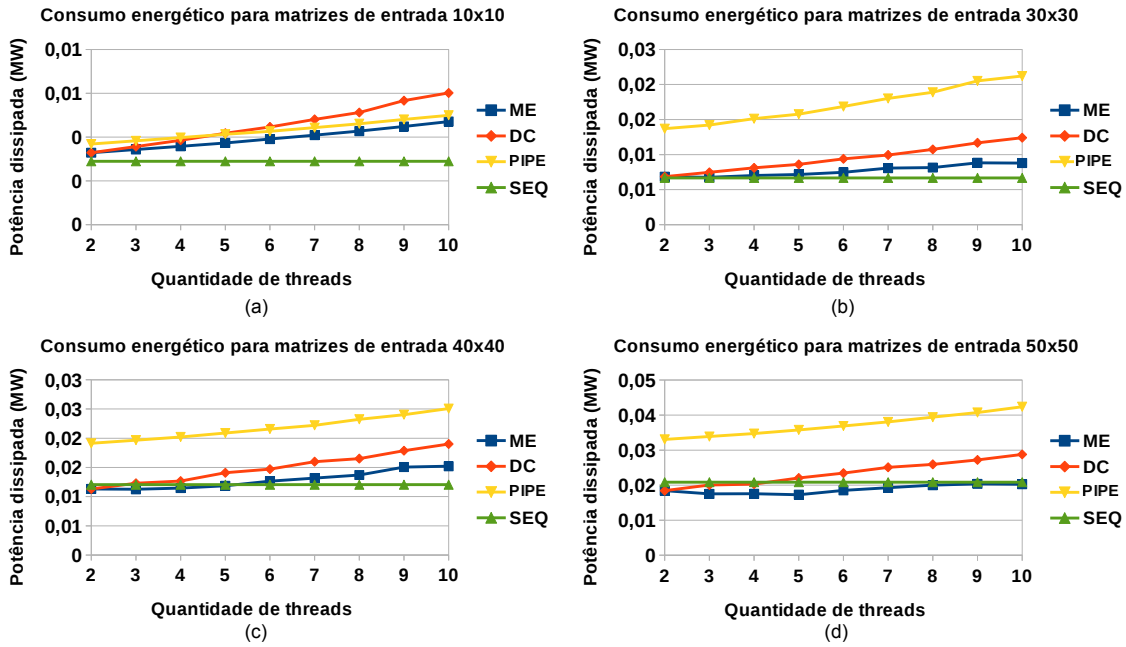
divididos igualmente entre as *threads*.

A Figura 40 apresenta gráficos contendo o consumo total de energia dos processadores plasma para diferentes dimensões de matrizes de entrada, onde o eixo x representa a quantidade de *threads* e o eixo y representa a potência dissipada pelo processadores em megawatt (MW).

Analisando somente os resultados dos paradigmas de programação paralela, o paradigma mestre-escravo teve um consumo energético menor em todos os casos. Em relação a versão sequencial, o paradigma mestre-escravo obteve um consumo energético menor utilizando até 5 tarefas escravas em matrizes de ordem 40 e em todos os casos para matrizes de ordem 50. Por consequência, espera-se que o paradigma mestre-escravo tenha menor consumo de energia em relação a versão sequencial para matrizes de ordem iguais ou superior a 50, considerando até 10 *threads*.

À medida que a ordem das matrizes de entrada aumenta, o consumo de energia do paradigma divisão e conquista se aproxima do consumo de energia da versão sequencial, chegando a ser menor para até 4 *threads* com matrizes de ordem 50. Desta forma, espera-se que o consumo de energia desse paradigma seja menor do que o consumo energético da versão sequencial para matrizes de ordem superior a 50.

Figura 40 – Energia Consumida nos Processadores Plasma pela Aplicação Multiplicação de Matrizes.



Fonte: Próprio Autor.

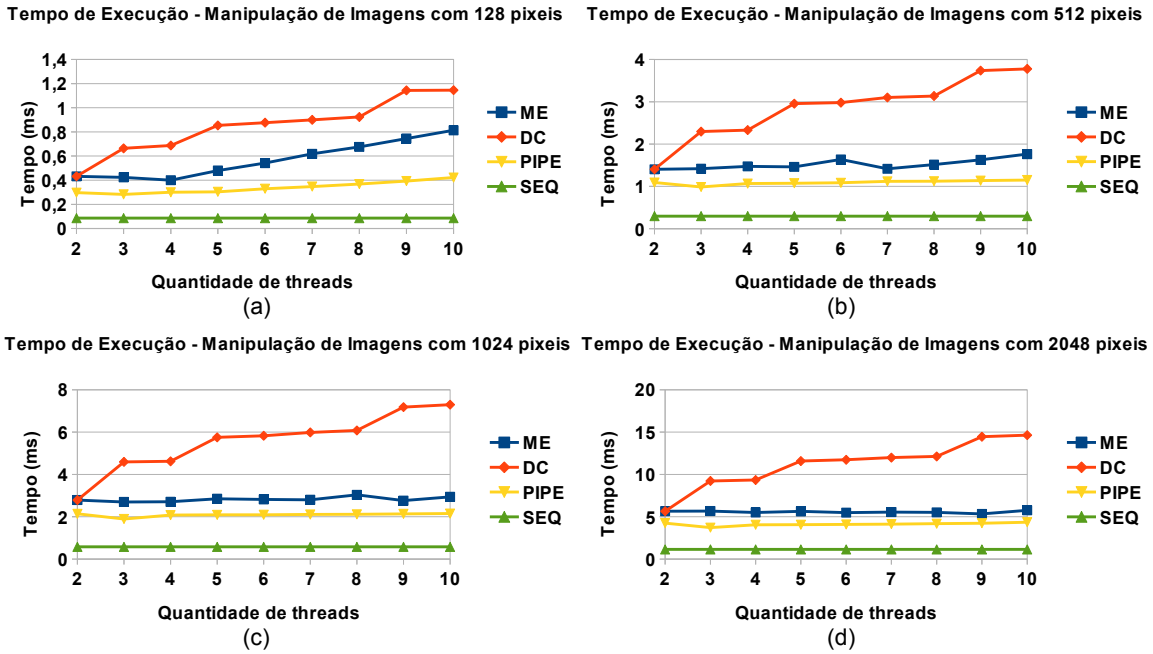
A exceção de matrizes de ordem 10, o paradigma *pipeline* consumiu mais energia do que os demais, não apresentando indícios de que seu consumo energético irá se aproximar ao valor da versão sequencial com o aumento da ordem das matrizes.

Considerando o desempenho, o paradigma que obteve o pior resultado, inclusive comparado a versão sequencial, foi o paradigma *pipeline*. O paradigma mestre-escravo obteve melhor desempenho, na maioria dos casos. No entanto, para matrizes de ordem superior a 50, a tendência é que o paradigma divisão e conquista obtenha melhores desempenhos que os demais paradigmas. Esses resultados são semelhantes aos obtidos por Raeder et al. (2011), que avaliaram o desempenho dessa aplicação em clusters de computadores pessoais, variando a ordem da matrizes de entrada de 10000, 20000 e 3000. O paradigma mestre-escravo também obteve o menor consumo energético em comparação aos demais paradigmas e, em alguns casos, menor do que a versão sequencial. Já os paradigmas divisão e conquista e *pipeline* apresentaram consumo de energia proporcional com o número de *threads*, sendo o paradigma *pipeline* menos eficiente.

6.3.2 Manipulação de Imagens

Nesta aplicação, os casos de testes alternaram o número de *threads* utilizadas entre 2 e 10 e tamanho da imagens de 128, 256, 512, 1024 e 2048 pixeis. A Figura 41 apresenta gráficos que evidenciam os tempos de execução obtidos para os diferentes tamanhos de entrada, onde o eixo x representa a quantidade de *threads* utilizadas e o eixo y representa o tempo em milissegundos.

Figura 41 – Tempo de Execução para a Aplicação Manipulação de Imagens.



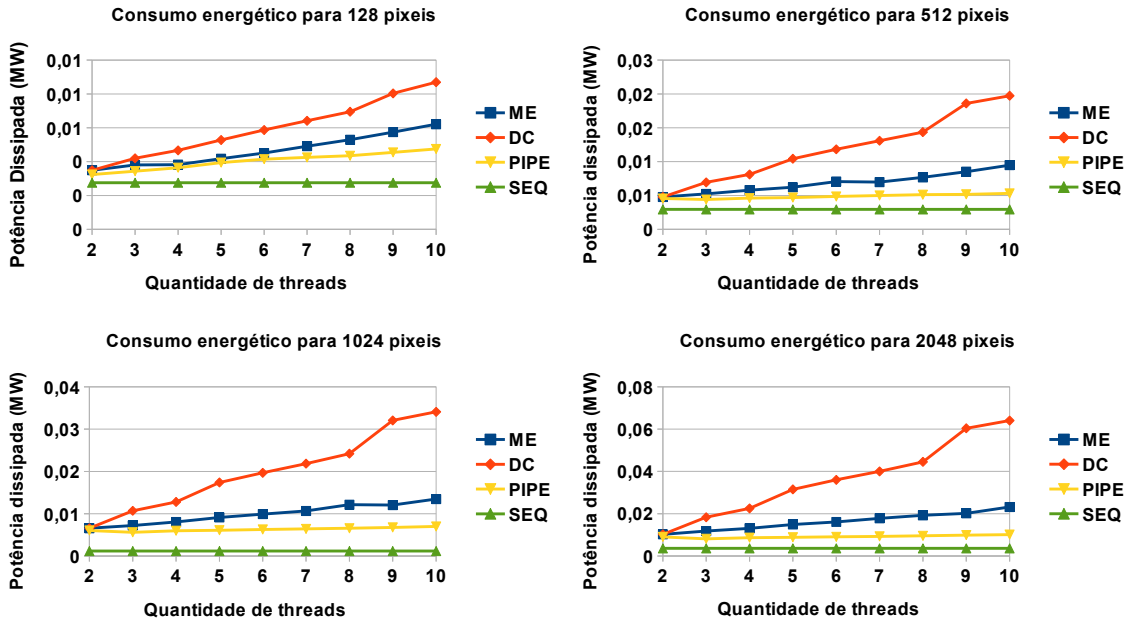
Fonte: Próprio Autor.

Analisando os resultados apresentados na Figura 41, o melhor desempenho do paradigma mestre-escravo foi obtido com 4 *threads* para imagem com 128 pixeis, com 7 *threads* para imagem com 512 pixeis, e com 9 *threads* para imagem com 1024 e 2048 pixeis. O melhor desempenho do paradigma *pipeline* foi obtido com 3 *threads*, independente do tamanho da imagem. Já o paradigma divisão e conquista apresentou perda de desempenho a medida que o número de *threads* aumentou. O paradigma com pior desempenho foi o divisão e conquista e o com melhor desempenho foi o *pipeline*. Entretanto, nenhum paradigma apresentou desempenho superior a versão sequencial devido a quantidade de processamento a ser realizada (somente foram analisadas imagens de pequeno tamanho).

A Figura 44 apresenta a energia consumida pelos processadores para executar a aplicação Manipulação de Imagens com diferentes tamanhos de entrada. No eixo x é

apresentada a quantidade de *threads* utilizadas, e no eixo y é apresentada a potência dissipada pelos processadores, em megawatt(MW).

Figura 42 – Energia Consumida nos Processadores Plasma pela Aplicação Manipulação de Imagens.



Fonte: Próprio Autor.

Observa-se na Figura 44 que o consumo energético aumentou em quase todos os casos a medida que mais *threads* foram adicionadas. No entanto, o aumento foi menor nos paradigmas mestre-escravo e *pipeline*, enquanto o paradigma divisão e conquista triplicou o consumo energético entre 2 e 10 *threads*. Nenhum dos paradigmas obteve menor consumo energético do que a versão sequencial. O paradigma divisão e conquista foi o que mais consumiu energia e o paradigma *pipeline* o que mais se aproximou do consumo da versão sequencial.

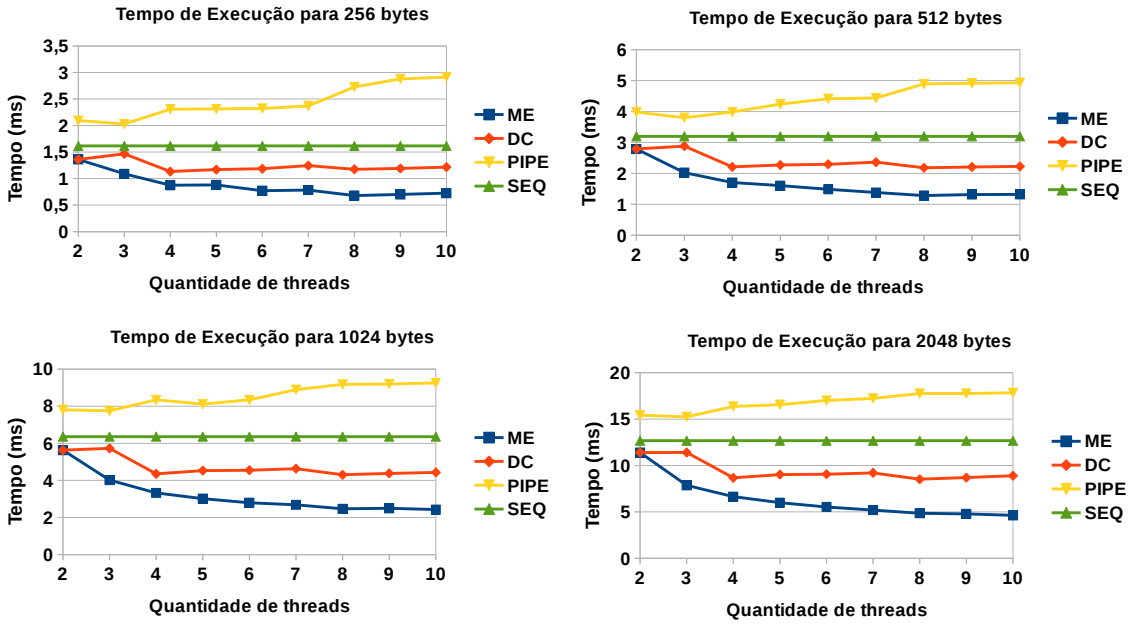
Pode-se concluir que, comparando somente os paradigmas de programação paralela, o paradigma *pipeline* obteve os melhores resultados tanto para o tempo de execução quanto para o consumo de energia. Isto ocorre porque a aplicação Manipulação de Imagens possui três estágios sequenciais e balanceados. Entretanto, os resultados desse paradigma foram piores que a versão sequencial. Assim, para os tamanhos de entrada simulados, a paralelização da aplicação Manipulação de Imagem não se justifica.

6.3.3 AES

Os casos de teste elaborados para essa aplicação alteraram o número de *threads* entre 2 e 10 e o tamanho da mensagem a ser criptografada de 256, 512, 1024 e 2048 bytes. O tempo de execução para os diferentes tamanhos de mensagem podem ser observados na Figura 44, onde o eixo x representa o número de *threads* e o eixo y representa o tempo em milissegundos.

Pode ser observado nos gráficos apresentados na Figura 44, que o paradigma mestre-escravo apresentou melhor desempenho, independente do tamanho da mensagem. Adicionalmente, o tempo de execução nesse paradigma diminui a medida que o número de *threads* aumenta. Para tamanho de mensagem maiores ou iguais a 512 bytes, O paradigma divisão e conquista apresentou melhor desempenho quando 8 *threads* foram utilizadas. Isto ocorre por causa do melhor balanceamento de carga de processamento entre as *threads*. Já o paradigma *pipeline* obteve o pior tempo de execução para todos os tamanhos de entrada, mesmo em comparação ao tempo da versão sequencial. Além disso, o tempo de execução desse paradigma piorou à medida que o número de *threads* aumentou.

Figura 43 – Tempo de Execução da Aplicação AES.

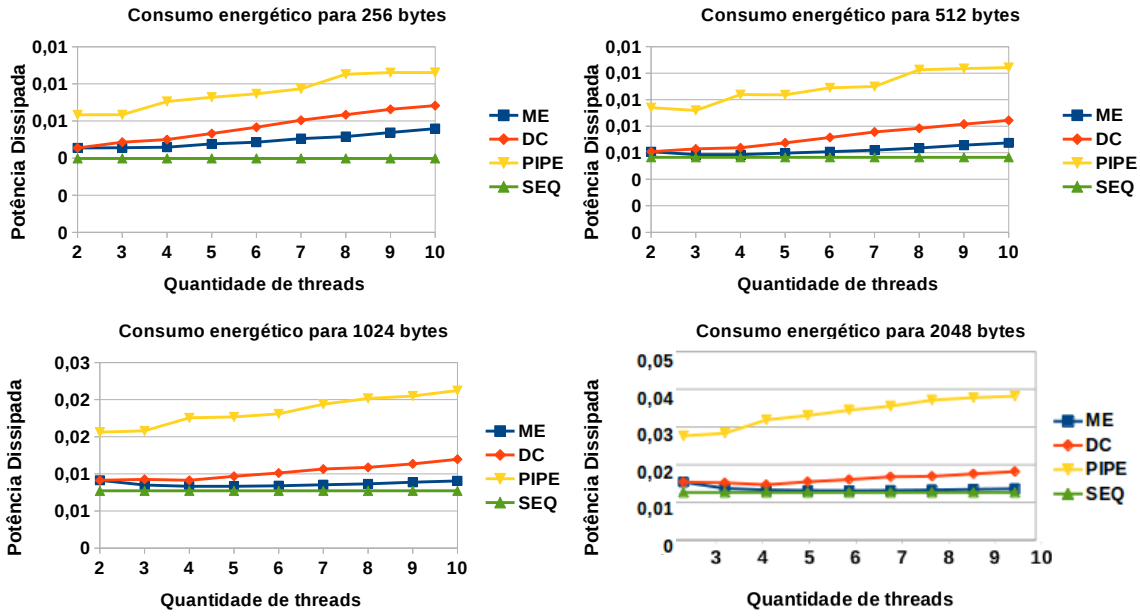


Fonte: Próprio Autor.

A Figura 44 apresenta gráficos contendo o consumo total de energia dos processadores plasma para os diferentes tamanhos de mensagem, onde o eixo x representa a

quantidade de *threads* e o eixo y representa a potência dissipada pelo processadores em megawatt (MW).

Figura 44 – Energia Consumida nos Processadores Plasma pela Aplicação AES.



Fonte: Próprio Autor.

Analisando os resultados apresentados na Figura 44, Observa-se que o consumo energético aumentou em quase todos os casos a medida que mais *threads* foram adicionadas. Comparando somente os paradigmas de programação paralela, o paradigma mestre-escravo teve um consumo energético menor em todos os casos.

Em relação a versão sequencial, o paradigma mestre-escravo obteve um consumo energético similar para tamanho de mensagem de 2048 bytes (Figura 44(d)). Desta forma, espera-se que o paradigma mestre-escravo tenha menor consumo de energia em relação a versão sequencial para tamanhos de mensagem superiores a 2048 bytes. À medida que o tamanho da entrada aumenta, o consumo de energia do paradigma divisão e conquista se aproxima do consumo de energia da versão sequencial. Por outro lado, o paradigma *pipeline* consumiu mais energia, não apresentando indícios de que seu consumo energético irá se aproximar ao valor da versão sequencial conforme o aumento da mensagem aumente.

Pode-se concluir que o paradigma mestre-escravo apresentou melhores resultados para a aplicação AES em relação ao tempo de execução e ao consumo energético em comparação aos demais paradigmas. O paradigma *pipeline* obteve piores desempenhos e maiores taxas de consumo energético, até mesmo em comparação a execução em se-

quencial. O paradigma divisão e conquista teve um consumo energético aproximado ao mestre-escravo. No entanto, apresentou um maior tempo de execução.

7 CONCLUSÕES

O objetivo geral do presente trabalho é investigar paradigmas de programação paralela sobre o MPSoC HeMPS. Para atingir tal objetivo, foi realizada uma investigação sobre MPSoCs e diferentes paradigmas de programação. Adicionalmente, uma pesquisa foi elaborada para identificar trabalhos que estudam a implementação de paradigmas de programação em arquiteturas paralelas (Clusters e Processadores Multicore) e outros que exploram o desenvolvimento de aplicações em MPSoCs. Com base nos trabalhos relacionados, foi definido o conjunto de paradigmas de programação que seriam avaliados, assim como os critérios.

Neste trabalho foram implementadas as aplicações: Multiplicação de Matrizes, Manipulação de Imagens e AES nos paradigmas de programação mestre-escravo, divisão e conquista e *pipeline*, considerando diferentes tamanho da entrada das aplicações e variando o número de *threads*. Assim foram analisados o tempo de execução e o consumo de energia nos processadores plasma para um total de 420 simulações.

Para a aplicação Multiplicação de Matrizes, o paradigma que obteve o pior desempenho, inclusive comparado a versão sequencial, foi o paradigma *pipeline*. O paradigma mestre-escravo obteve melhor desempenho, na maioria dos casos. No entanto, para matrizes de ordem superior a 50, a tendência é que o paradigma divisão e conquista obtenha melhores desempenhos que os demais paradigmas. Em relação ao consumo de energia, o paradigma mestre-escravo obteve o menor consumo energético em comparação aos demais paradigmas e, em alguns casos, menor do que a versão sequencial. Já os paradigmas divisão e conquista e *pipeline* apresentaram consumo de energia proporcional com o número de *threads*, sendo o paradigma *pipeline* menos eficiente. Assim, para manter o compromisso entre desempenho e consumo de energia, recomenda-se o uso do paradigma mestre-escravo para implementar essa aplicação em MPSoCs.

Os resultados para a aplicação AES foram semelhantes aos obtidos para a aplicação Multiplicação de Matrizes. O paradigma mestre-escravo apresentou melhores resultados tanto em relação ao tempo de execução quanto ao consumo energético, em comparação aos demais paradigmas. O paradigma *pipeline* obteve piores desempenhos e maiores taxas de consumo energético, até mesmo em comparação a execução em sequencial. O paradigma divisão e conquista teve um consumo energético aproximado ao mestre-escravo. No entanto, apresentou um maior tempo de execução. Consequentemente, recomenda-se o uso do paradigma mestre-escravo para a implementação da aplicação AES a fim de que o compromisso tempo de execução e consumo de energia seja mantido.

Para aplicação Manipulação de Imagens, o paradigma *pipeline* obteve os melhores resultados tanto para o tempo de execução quanto para o consumo de energia quando comparado somente aos demais paradigmas. Entretanto, seus resultados foram piores que a versão sequencial. Assim, para as variáveis simuladas, a paralelização da aplicação Manipulação de Imagem não se justifica.

As aplicações Multiplicação de Matrizes e AES possuem um estágio com carga de processamento maior do que os demais. Por esta razão, o paradigma *pipeline* não é adequado porque os estágios ficam desbalanceados, implicando em aumento do tempo de execução e consumo energético. Já a aplicação Manipulação de Imagens possui três estágios sequenciais, o que justifica o fato do paradigma pipeline ser mais adequado do que os demais. No entanto, esses estágios demandam pouco processamento, o que não justifica o uso de processamento paralelo para imagens com poucos pixeis.

A partir da análise dos resultados foi possível fazer algumas descobertas:

- O número de tarefas impacta no consumo energético, independente do paradigma de programação utilizado;
- No paradigma divisão e conquista, o número de tarefas folha deve ser 2^n para que a árvore esteja balanceada e, consequentemente, sejam obtidos os melhores resultados em termos de tempo de execução e consumo de energia;
- O paradigma *pipeline* é indicado apenas para aplicações que possuam estágios sequenciais e com carga de processamento balanceada;
- O paradigma mestre-escravo é indicado para as aplicações Multiplicação de Matrizes e AES, considerando os tamanhos de entrada simulados;
- O paradigma divisão e conquista tende a obter menores tempos de execução para tamanhos maiores de entrada. No entanto, o consumo de energia tende a continuar sendo maior devido ao número de tarefas desse paradigma;

Alguns desafios foram encontrados durante a execução deste trabalho, entre eles estão:

- Encontrar aplicações que pudessem ser implementadas no MPSoC HeMPS (por exemplo, funções com integral e operações trigonométricas não são suportadas);
- Compreender as diferentes camadas existentes no MPSoC HeMPS;
- Falta de mecanismos para depurar as aplicações paralelas executadas sobre o MPSoC HeMPS;
- Impossibilidade de simular aplicações com entradas maiores no MPSoC HeMPS; e
- Situações de *deadlock* impossibilitaram a simulação das aplicações no paradigma fases paralelas.

Cita-se como possíveis trabalhos futuros:

- Reavaliar os paradigmas utilizando mapeamentos que considerem o grafo de comunicação das tarefas;

- Explorar outras configurações do MPSoC, como: o número de tarefas por processador, o tamanho da página de memória e o número de clusters;
- Investigar o consumo energético da NoC para as diferentes aplicações e paradigmas;
- Descobrir o motivo das aplicações não estarem funcionando para entradas maiores.

Todas as aplicações implementadas neste trabalho e os resultados obtidos estão disponíveis em: <https://github.com/geaninne/TCC>.

REFERÊNCIAS

BALDO, L. J. et al. Predição de desempenho de aplicações paralelas para máquinas agregadas utilizando modelos estocásticos. Pontifícia Universidade Católica do Rio Grande do Sul, 2006. Citado na página 30.

BOHNENSTIEHL, B. et al. A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In: IEEE. **VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on**. Honolulu, HI, USA, 2016. p. 1–2. Citado na página 20.

CARARA, E. A. et al. Hemps-a framework for noc-based mpsoc generation. In: IEEE. **Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on**. [S.l.], 2009. p. 1345–1348. Citado 3 vezes nas páginas 20, 23 e 30.

CARVALHO, E. L. d. S. Mapeamento dinâmico de tarefas em mpsocs heterogêneos baseados em noc. Pontifícia Universidade Católica do Rio Grande do Sul, 2009. Citado 2 vezes nas páginas 20 e 30.

CASTILHOS, G. et al. A framework for mpsoc generation and distributed applications evaluation. In: IEEE. **Quality Electronic Design (ISQED), 2014 15th International Symposium on**. [S.l.], 2014. p. 408–411. Citado na página 26.

CHAPMAN, B. et al. Implementing openmp on a high performance embedded multicore mpsoc. In: IEEE. **Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on**. [S.l.], 2009. p. 1–8. Citado na página 46.

CHI, C. C.; JUURLINK, B.; MEENDERINCK, C. Evaluation of parallel h. 264 decoding strategies for the cell broadband engine. In: ACM. **Proceedings of the 24th ACM International Conference on Supercomputing**. [S.l.], 2010. p. 105–114. Citado 3 vezes nas páginas 43, 45 e 46.

CONSORTIUM, S. et al. **The SoCLib project: An integrated system-on-chip modelling and simulation platform**. [S.l.], 2003. Citado na página 20.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. aes algorithm submission, september 3, 1999. **URL http://www. nist. gov/CryptoToolKit**, 1999. Citado 4 vezes nas páginas 37, 38, 39 e 42.

DINECHIN, B. D. D. et al. Time-critical computing on a single-chip massively parallel processor. In: IEEE. **Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014**. Dresden, Germany, 2014. p. 1–6. Citado na página 20.

FILHO, S. J. et al. Suporte para aplicações dinâmicas em sistemas multiprocessados intra-chip homogêneos. Pontifícia Universidade Católica do Rio Grande do Sul, 2012. Citado na página 68.

FOCHI, V. M. **Técnicas de tolerância a falhas aplicadas a redes intra-chip**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul, 2015. Citado na página 30.

GALANTE, G. et al. Solução paralela de sistemas de equações lineares através de métodos de decomposição de domínio. **ERAD 2004**, 2004. Citado 2 vezes nas páginas 30 e 31.

GAPH, G. de Apoio ao Projeto de H. **HeMPS Hermes Multiprocessor System on Chip**. [S.l.], 2017. Disponível em: <https://www.inf.pucrs.br/hemps/getting_started.html>. Citado 3 vezes nas páginas 26, 27 e 28.

GUINDANI, G. M. Mecanismo de controle de qos através de dfs em mpsocs. Pontifícia Universidade Católica do Rio Grande do Sul, 2014. Citado 2 vezes nas páginas 13 e 68.

IEZZI, G.; HAZZAN, S. **Fundamentos de matemática elementar, 4: sequências, matrizes, determinantes, sistemas**. [S.l.]: Atual, 2004. Citado na página 33.

JERRAYA, A. A.; WOLF, W. The future of multiprocessor systems-on-chips. **Computer**, IEEE, v. 38, n. 2, p. 63–69, 2005. Citado 2 vezes nas páginas 19 e 20.

LANGE, A.; BOHRER, V. P. Adaptando sistemas para operar com redes intrachip: Decodificador m-jpeg/hermes. 2010. Citado 3 vezes nas páginas 43, 45 e 46.

LOPES, J. M. B. Cor e luz. **Texto elaborado para a disciplina de Computação Gráfica**, 2013. Citado na página 34.

MARTIN, G.; CHANG, H. **Winning the SoC revolution: experiences in real design**. [S.l.]: Springer Science & Business Media, 2012. Citado na página 19.

MEYER, L. A. V. C. Uma visão geral dos sistemas distribuídos de cluster e grid e suas ferramentas para o processamento paralelo de dados. **IBGE [sd]. Disponível em https://ww2. ibge. gov. br/confest_e_confege/pesquisa_trabalhos/CD/palestras/368-1. pdf. Acesso em**, v. 8, 2017. Citado 2 vezes nas páginas 30 e 31.

MEYER, V. Pipel: modelo de gerência da elasticidade para aplicações organizadas em pipeline. Universidade do Vale do Rio dos Sinos, 2016. Citado na página 34.

MISAGHI, M. Tópicos especiais em segurança da informação. 2017. Disponível em: <http://wiki.stoa.usp.br/images/c/c1/Aula2-TES.pdf>. Citado na página 42.

MORAES, F. et al. **HeMPS Platform**. [S.l.], 2016. Citado 4 vezes nas páginas 23, 24, 25 e 29.

MÜCK, T. R. et al. Projeto unificado de componentes em hardware e software para sistemas embarcados. 2013. Citado na página 19.

OLIVEIRA, A. S. de; ANDRADE, F. S. de. **Sistemas embarcados: hardware e firmware na prática**. [S.l.]: Editora Érica Ltda, 2006. Citado na página 19.

OPENCORES. **Plasma - most MIPS I(TM) opcodes :: Overview**. [S.l.], 2001. Citado na página 23.

PRESS, W. H. **Numerical recipes 3rd edition: The art of scientific computing**. [S.l.]: Cambridge university press, 2007. Citado na página 33.

RAEDER, M. et al. Performance prediction of parallel applications with parallel patterns using stochastic methods. **Sistemas Computacionais (WSCAD-SSC), XII Simpósio em Sistemas Computacionais de Alto Desempenho**, p. 1–13, 2011. Citado 5 vezes nas páginas 43, 44, 45, 50 e 71.

REBONATTO, M. T. et al. Comparação de algoritmos paralelos em uma rede heterogênea de workstations. 2002. Citado 2 vezes nas páginas 43 e 44.

REGO, R. S. d. L. S. **Projeto e implementação de uma plataforma MP-SoC usando SystemC**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2006. Citado na página 20.

REIS, C. Sistemas operacionais para sistemas embarcados. **Livro, 1º Ed. Editora: EDUFBA, Brasil**, 2004. Citado na página 19.

REZENDE, L.; CAIMI, L. **Desenvolvimento da Aplicacão AES para HeMPS**. [S.l.], 2013. Disponível em: <https://github.com/GaphGroup/hemps/blob/master/hemps8.5/applications/aes/aes_generator/Aplicacao_AES_para_HeMPS.pdf>. Citado 3 vezes nas páginas 40, 41 e 55.

RUARO, M. et al. Dmni: A specialized network interface for noc-based mpsocs. In: IEEE. **Circuits and Systems (ISCAS), 2016 IEEE International Symposium on**. [S.l.], 2016. p. 1202–1205. Citado na página 24.

SALDANA, M.; CHOW, P. Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. In: IEEE. **Field Programmable Logic and Applications, 2006. FPL'06. International Conference on**. [S.l.], 2006. p. 1–6. Citado na página 46.

SHEE, S. L.; ERDOS, A.; PARAMESWARAN, S. Heterogeneous multiprocessor implementations for jpeg:: a case study. In: ACM. **Proceedings of the 4th international conference on Hardware/software codesign and system synthesis**. [S.l.], 2006. p. 217–222. Citado 3 vezes nas páginas 43, 44 e 45.

STANDARD, A. E. Federal information processing standards publication 197. **FIPS PUB**, p. 46–3, 2001. Citado 2 vezes nas páginas 37 e 40.

TANURHAN, Y. Processors and fpgas quo vadis? **Computer**, IEEE, v. 39, n. 11, 2006. Citado na página 20.

TILERA. **Tile Processor Architecture Overview for the TILEGx Series**. [S.l.], 2012. Citado na página 20.

VIDAL, E. L.; MELLO, A. V. D. Um estudo sobre os desafios atuais de sistemas multiprocessados embarcados. **Anais do Salão Internacional de Ensino, Pesquisa e Extensão**, v. 9, n. 3, 2018. Citado na página 28.

WANG, Z. et al. A case study on the communication and computation behaviors of real applications in noc-based mpsocs. In: IEEE. **VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on**. [S.l.], 2014. p. 480–485. Citado 2 vezes nas páginas 43 e 46.

WOLF, W. Hardware/software interface codesign for embedded systems. IEEE, p. 682–685, 2004. Citado na página 20.