UMA AVALIAÇÃO EXPERIMENTAL DA PLATAFORMA *PARALLELLA* UTILIZANDO CONTROLE PREDITIVO BASEADO EM MODELO COMO UM ESTUDO DE CASO

MARLON MARQUES SOUDRÉ

DISSERTAÇÃO DE MESTRADO EM SISTEMAS MECATRÔNICOS

DEPARTAMENTO DE ENGENHARIA MECÂNICA

UNIVERSIDADE DE BRASÍLIA - UnB
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS MECATRÔNICOS

UMA AVALIAÇÃO EXPERIMENTAL DA PLATAFORMA *PARALLELLA* UTILIZANDO CONTROLE PREDITIVO BASEADO EM MODELO COMO UM ESTUDO DE CASO

Dissertação apresentada ao Departamento de Engenharia Mecânica da Faculdade de Tecnologia da Universidade de Brasília como parte dos requisitos necessários para obtenção do grau de Mestre em Sistemas Mecatrônicos

MARLON MARQUES SOUDRÉ

APROVADA POR

_____

Prof. Dr.Carlos Humberto Llanos Quintero.
(PPMEC- Orientador)

_____

Prof. Dr.Ricardo Pezzuol Jacobi.
(PPMEC- Co-Orientador)

_____

Prof. André Murilo de Almeida Pinto.
(PPMEC, Avaliador Interno)

_____

Prof. Alba Cristina Magalhães Alves de Melo.
(CIC-UnB, Avaliadora Externa)

BRASÍLIA/DF, 04 DE ABRIL DE 2017

FICHA CATALOGRÁFICA

SOUDRÉ, MARLON MARQUES

Uma Avaliação Experimental da Plataforma Parallella Utilizando Controle Preditivo Baseado em Modelo Como um Estudo de Caso, [Distrito Federal] 2017.

No. 119, 117p., 210 x 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2017). Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia. Departametno de Engenharia Mecânica.

1. Plataforma *Parallella* 2. MPC

3. Sistemas Embarcados 4. Arquitetura *Multicore*

REFERÊNCIA BIBLIOGRÁFICA

SOUDRÉ, M. M. (2017). Uma Avaliação Experimental da Plataforma Parallella Utilizando Controle Preditivo Baseado em Modelo Como um Estudo de Caso. Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM-119/17, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 117p.

CESSÃO DE DIREITOS

AUTOR: Marlon Marques Soudré

TÍTULO: Uma Avaliação Experimental da Plataforma Parallella Utilizando Controle Preditivo Baseado em Modelo Como um Estudo de Caso.

GRAU: Mestre ANO: 2017

_____

Marlon Marques Soudré

RESUMO

UMA AVALIAÇÃO EXPERIMENTAL DA PLATAFORMA PARALLELLA UTILIZANDO CONTROLE PREDITIVO BASEADO EM MODELO COMO UM ESTUDO DE CASO.
Autor: Marlon Marques Soudré
Orientador: Prof. Carlos Humberto Llanos Quintero
Programa de Pós-Graduação em Sistemas Mecatrônicos
Brasília, 04 de Abril de 2017

Nas últimas décadas, o poder computacional de sistemas embarcados têm crescido de forma muito rápida. Em geral, tais sistema são projetados para operar sob restrições como portabilidade (peso e tamanho), consumo de recursos, baixo consumo de energia e dissipação de potência. Assim, motivado pelos fatores supracitados e pelo avanço tecnológico, assim como pela demanda crescente de desempenho por parte das aplicações embarcadas, têm surgido vários processadores e plataformas de *hardware* que fazem uso de arquiteturas *multicore*, com destaque para a *Parallella*, uma plataforma de alto desempenho e baixo consumo energético. Nesse sentido, o presente trabalho traz a proposta de se avaliar tal plataforma sob uma abordagem experimental, como foco em seu coprocessador *Epiphany* de 16 *cores*, quando utilizada como um acelerador em *software* para aplicações de controle preditivo baseado em modelo como um estudo de caso, devido sua relevância para o grupo de pesquisa do LEIA (Laboratório de Sistemas Embarcados e Aplicações de Circuitos Integrados – Universidade de Brasília). Os resultados mostram que, apesar de restrições críticas como o tamanho da memória local dos *cores*, a plataforma *Parallella* se apresenta como uma arquitetura em potencial, podendo ser vista como uma alternativa à aceleração de algoritmos em *hardware*. Melhorias futuras como a expansão do número de núcleos do MPSoC *Epiphany* e da memória local dos mesmos, como previsto pelos fundadores do projeto, poderão alavancar ainda mais o uso de tal arquitetura em aplicações embarcadas.

Palavras-chave: Sistemas Embarcados, Arquitetura *Multicore*, Plataforma *Parallella*, MPC.

ABSTRACT

In the last decades, the computational power of embedded systems has grown very fast. In general, such systems are designed to operate under constraints such as portability, resource consumption, low power consumption and power dissipation. Thus, due to the aforementioned factors and technological advances, as well as the increasing demand for performance by embedded applications, there have been several processors and hardware platforms that make use of multicore architectures, with emphasis on a Parallella, a platform of high performance and low consumption. In this sense, the present work presents a proposal to evaluate such platform in an experimental approach, focusing on its Epiphany 16-core co-processor, when used as a software accelerator for model-based predictive control applications as a case study, due to its relevance to the research group of LEIA (Laboratory of Embedded Systems and Applications of Integrated Circuits - University of Brasilia). The results show that, despite critical constraints such as the local memory size of the cores, a Parallella platform presents itself as a potential architecture and can be seen as an alternative to accelerating hardware algorithms. Future improvements such as the expansion of the number of MPSoC Epiphany cores and their local memory, as predicted by the founders of the project, can leverage the use of this architecture in embedded applications.

Keywords- Embedded Systems, Multicore Architecture, Platform Parallella, MPC

Sumário

Lista de Figuras

Lista de Tabela

Siglas

MPC	*Model Predictive Control*
FPGA	*Field Programmable Gate Array*
NoCs	*Network − On − Chip*
DMA	*Direct Memory Access*
PID	*Proportional Integral Derivative*
RST	*Reference Signal Tracking*
SISO	*Single Input Single Output*
RISC	*Reduced Instruction Set Computer*
SDRAM	*Synchronous Dynamic Random Access Memory*
SoC	*System on Chip*
MSB	*Most Significant Bits*
HDF	*Hardware Description File*
LDF	*Linker Description File*
IDE	*Integrated Development Environment*
ULA	*Arithmetic Logic Unit*
FPU	*Floating Point Unit*
SISD	*Single Instruction, Single Data*
MISD	*Multiple Instrucrion, Single Data*
SIMD	*Single Instruction, Multiple Data*
MIMD	*Multiple Instruction, Multiple Data*
DMM	*Distribuited Memory Machines*
SMM	*Shared Memory Machine*
SMP	*Simetric Multiprocessors*
eSDK	*Epiphany Software Development Kit*
HDMI	*High − Definition Multimidia Interface*
VNC	*Virtual Network Computing*
SSH	*Secure Shell*
IDE	*Integrated Development Environment*
CSP	*Communicating Sequential Processes*
VFPU	*Vector Floating Point Units*
FMADD	*Fused Multiply Add*
FFT	*Fast Fourier Transform*
LTI	*Linear Time Invariant*
QP	*Quadratic Programming*

Capítulo 1

Introdução

Sistemas embarcados são componentes de *hardware* e *software* integrados em uma solução adequada, com o objetivo de cumprir várias funcionalidades de um produto específico. Em geral, tais sistemas são aplicados em diferentes áreas da indústria, por exemplo: (a) sistemas de controle e automação, (b) processamento de sinais, (c) comunicações e telefonia; entre outros [1].

Quanto à classificação, os sistemas embarcados podem ser do tipo (a) *reativo* ou (b) do tipo *tempo real*. Nos sistemas de *tipo reativo*, os mesmos respondem a eventos externos, que podem ser periódicos (caso de sistemas rotacionais ou de controles de laço) ou assíncronos (com entradas acionadas por botões ou dispositivos similares). Por outro lado, nos sistemas do tipo *tempo real*, existem limites ou restrições de tempo para executar cada tarefa (por exemplo, leitura de sensores ou emissão de sinais para um atuador). Este tipo de operação, por ser cíclico, não depende necessariamente de entradas ou sinais para executar as atividades, sendo capaz de tomar decisões na ausência das mesmas [82].

Os sistemas de tempo real são divididos em: (a) *soft real time*, onde tarefas podem ser executadas em um intervalo de tempo específico, sem consequências graves se este limite de tempo não for cumprido (por exemplo, sistemas administrativos objetivando a área bancária), ou (b) *hard real time*, onde tarefas devem ser executadas em um tempo específico, com consequências graves se qualquer tarefa falhar (por exemplo, sistemas de controle aplicados na aviação, transporte, ou aplicações em engenharia biomédica) [81].

Nas últimas décadas, o poder computacional de sistemas embarcados e sistemas móveis têm crescido de forma muito rápida. Como um exemplo deste fato, celulares atuais incluem processadores embarcados com diversos núcleos, executando em frequências da ordem de giga-hertz. Neste sentido, como um exemplo de caso, o esforço no mercado de processadores móveis tem sido sempre focado em obter circuitos de menor tamanho, com menor consumo de energia (e, portanto, maior duração da bateria), onde usualmente os processadores são integrados em projetos do tipo SoC (do inglês, *System on Chip*) e/ou MPSoC (do inglês, *Multi-Processor System on Chip*). Neste mercado, companhias como *Qualcomm*, *Texas Instruments* e *Nvidia* estão competindo na maioria de negócios relacionados com *smartphones* de última geração, a partir de projetos compatíveis com processadores ARM [83].

A integração de subsistemas de um produto em SoC ou MPSoC é, portanto, uma alternativa para otimizá-lo em termos de desempenho e consumo de energia. Nesse contexto, o projeto de sistemas embarcados pode adotar técnicas de co-projeto de *hardware*/*software* (*hardware*/*software codesign* [3]), onde os diversos processos que compõem a solução são analisados visando a determinação da tecnologia de implementação mais adequada. Por esse motivo surge a necessidade de se verificar quais partes do sistema a ser projetado são críticas nos quesitos de desempenho, área, consumo de potência, robustez, tolerância a falhas, flexibilidade, portabilidade, entre outras.

A análise experimental de desempenho da aplicação (denominada de *profiling*) permite levantar estas características, que servem de subsídio para a otimização do sistema. Em termos de desempenho, processos críticos podem ser acelerados através da implementação de seus algoritmos diretamente em *hardware* reconfigurável, (tais como FPGAs – do inglês, *Field Programmable Gate Array*), ou por meio de arquiteturas paralelas com múltiplos núcleos (*many* ou *multicore* [4]) ou ainda, em casos mais extremos, através do desenvolvimento de circuitos dedicados em silício – ASICs (do inglês, *Application Specific Integrated Circuits*).

No caso dos sistemas do tipo MPSoCs (envolvendo múltiplos núcleos), há o suporte a redes intra-chip (NoC- do inglês, *Network on Chip*). Nos últimos anos, a ideia de se utilizar NoCs como comunicação viável *on-chip* para sistemas MPSoCs vem ganhando força, se apresentando como uma tentativa de se aplicar os conceitos de redes em grande escala, adaptando-os a sistemas embarcados baseados em SoCs [5]. Ao contrário das tradicionais arquiteturas de comunicação *on-chip* baseadas em barramento, as NoCs usam pacotes para rotear dados da fonte para o componente de destino, através de uma rede constituída de *switches* (roteadores) e *links* de interconexão, mostrando-se como uma solução com potencial de atender aos requisitos de escalabilidade e QoS (do inglês, *Quality of Service*) de SoCs complexos[6].

Em geral, todos os tipos de sistemas embarcados são projetados para operar sob restrições de portabilidade (peso e tamanho), consumo de recursos, baixa frequência de *clock*, desempenho adequado, baixo consumo de energia e dissipação de potência. Normalmente é desejável uma frequência de *clock* tão baixa quanto possível, embora existam sistemas embarcados trabalhando a frequências da ordem de giga-hertz, tal como acontece nas plataformas de telefonia celular. Entretanto, trabalhar a frequências altas termina por penalizar em alguns casos a autonomia das baterias deste tipo de sistemas, por um incremento no consumo de energia (problema conhecido como *power-wall* [7]). Por esse motivo, são desenvolvidas soluções de *hardware* que permitam explorar o paralelismo intrínseco dos algoritmos a ser embarcados, permitindo assim obter implementações com bom desempenho, operando também com baixas frequências de *clock*, aliviando assim o problema de *power-wall*.

Um ponto importante na proposta deste trabalho é a necessidade do desenvolvimento de sistemas embarcados aplicados nas áreas de automação e controle, robótica e visão

computacional, como parte da vocação e objetivos do LEIA (Laboratório de Sistemas Embarcados e Aplicações de Circuitos Integrados), e que vem trabalhando dentro do GRACO (Grupo de Automação e Controle), pertencentes à Universidade de Brasília. Neste contexto, e dentro das novas atividades do LEIA, a área de Controle Preditivo Baseado em Modelo (MPC, do inglês, Model Predictive Control) vem se tornando uma nova atividade de pesquisa, tendo em conta novos projetos envolvendo a construção de um exoesqueleto, para aplicações de reabilitação de pessoas com deficiências motoras.

1.1 Plataformas Comerciais de Sistemas Embarcados

Motivado pelos fatores supracitados e pelo avanço tecnológico, assim como pela demanda crescente de desempenho por parte das aplicações embarcadas, têm surgido vários processadores e plataformas de *hardware* que fazem uso de arquiteturas *multicore* (por exemplo, *Intel Xeon Phi* [8], *Recore Xentium RFD* [9], TILE64 [10]), trazendo dentre seus muitos desafios, o de aliar um bom desempenho computacional com facilidade de desenvolvimento a um custo reduzido.

Neste cenário, um projeto lançado pela Adapteva [11], uma empresa *startup* de semicondutores fundada em 2008 e conduzida por Andreas Olofsson, Roman Trogan e Yaniv Sapir, traz como promessa o desenvolvimento de um *hardware* de baixo custo, assim como ferramentas de desenvolvimento *OpenSource* que tornem a computação paralela facilitada aos desenvolvedores (aplicada na área de sistemas embarcados). Inspirado em grandes comunidades como o *Raspbarry Pi* e *Arduino*, o projeto visa criar uma plataforma de alto desempenho e baixo consumo energético, democratizando o acesso a programação paralela. Tal projeto foi concebido sobre os seguintes pilares:

- Acesso aberto a documentos de arquiteturas e SDKs (do inglês, *Software Development Kit*);
- Ferramentas de desenvolvimento e bibliotecas *OpenSource*;
- Custo acessível (tipicamente abaixo de US$100).

A plataforma em questão (denominada *Parallella*) tem como pontos centrais: (a) um coprocessador *Epiphany multicore* (do tipo MPSoC) com versões iniciais de 16 e 64 *cores* em um único *chip*; (b) uma rede *intra-chip* (NoCs - *Network on Chip*), interligando os processadores do *Epiphany*, disponibilizando um sistema escalável, com o objetivo de expansão tanto no número de núcleos de processamento quanto no tamanho da memória interna dos mesmos, como visto na Figura 1.1.1, mostrando a pretensão dos desenvolvedores no início do projeto [11] e (c) um processador ARM A9 Dual Core e um FPGA dentro de um núcleo Zynq (da Xilinx [12]), sendo que o FPGA é pré-configurado para viabilizar a comunicação entre o ARM e o *Epiphany* [62].

O grande diferencial da plataforma *Parallella* com respeito a outras plataformas *single core*, está na presença de um coprocessador *multicore* (do tipo MPSoC), possibilitando a exploração dos recursos da programação paralela. Assim, ao invés de competir com outras arquiteturas, a *Parallella* traz uma abordagem diferente, com o enfoque no paralelismo de aplicações visando o ganho de desempenho computacional, mostrando um grande potencial para o desenvolvimento de sistemas embarcados (neste caso, aproveitando os recursos e técnicas da computação paralela).

Entretanto, quando comparada a arquiteturas populares como *Raspbarry Pi*, a plataforma *Parallella* apresenta algumas desvantagens, como por exemplo um processador ARM A9 *Dual Core* de 667MHz, frente a um *Quad Core* ARM –A53 de 1.2GHz, maior custo e ausência de recursos gerais como módulo *bluetooth* e *wi-fi*. Além de tais fatores, o *Raspbarry* conta com uma grande comunidade de desenvolvedores, o que facilita a melhoria constante da plataforma em questão, ao passo que a *Parallella* é uma arquitetura nova, porem emergente.

Nesta mesma direção, existem outras arquiteturas multicore poderosas, tais como o *Intel Xeon Phi*, já citado anteriormente, com até 72 *cores*, 16GB de memória e operando a 1.5GHz. No entanto, este último apresenta um custo bem mais elevado que a plataforma em estudo neste trabalho, além de maior consumo energético, inviabilizando seu uso para muitas das aplicações baseadas em sistemas embarcados que tenham tais fatores como restrições.

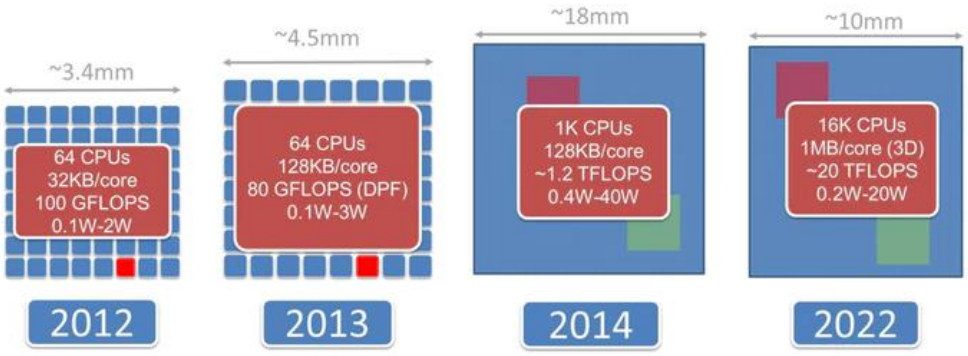

Figura 1.1. 1 Previsão de evolução do coprocessador Epiphany multicore Adapteva [11].

Do ponto de vista histórico, a primeira placa *Parallella* foi produzida no ano de 2013, contando com Zynq-7020 *dual-core* ARM A9 CPU e versões do coprocessador *Epiphany* com 16 e 64 *cores* com 32KB/*core* de memória interna, sendo esses ligados por uma NoC, além de uma memória compartilhada de 1GB e outros periféricos.

Desde então, muitos trabalhos têm surgido com o intuito de explorar essa arquitetura, atraídos pelo custo (sendo US$99,00 para sua versão mais simples), assim como pelo fato de ser essa uma plataforma flexível; reunindo componentes de grande uso em sistemas embarcados como um FPGA, processador ARM, uso de NoCs e de um paralelismo via sistema do tipo MPSoC,

além da promessa de baixo consumo energético, alto rendimento e da facilidade de implementação (ferramentas *OpenSource* e programação facilitada em linguagem C/C++).

Neste contexto, o número de trabalhos envolvendo a plataforma *Parallella* cresce a cada ano, como mostrado pelo gráfico da Figura 1.1.2. Apesar das primeiras placas *Parallella* terem sido produzidas no ano de 2013, os trabalhos referentes a 2010 [13] e 2011 [14] são de autoria dos próprios criadores do projeto, apresentando os conceitos da arquitetura *multicore Epiphany* e realizando análises de consumo energético e desempenho.

Os trabalhos envolvendo a plataforma *Parallella* estão divididos entre estudos iniciais da plataforma (apresentando sua arquitetura, principais características e configurações, [16] e [17]), passando por trabalhos cujos objetivos estão na análise de pontos gerais [18] e específicos da placa, tais como a sua eficiência energética [19], até trabalhos com um estudo de caso envolvendo aplicações reais [20].

Dentre estes últimos, as áreas de aplicação variam entre visão computacional [21], processamento de imagens [22], processamento de sinais [23], *Deep Learning* [24], Redes Neurais Artificiais (RNA) [25], algoritmos bio-inspirados [26], UAN's (*Unmanned Aerial Vehicle*) [27], IoT (do inglês, *Internet of Things*) [28], segurança da informação [29], dentre outros.

Em se tratando do uso da plataforma *Parallella* para a área de controle, são encontradas poucas referências na literatura, indicando que este ainda é um ramo pouco explorado para esta arquitetura em questão. Trabalhos como [30] utilizam a placa *Parallella* na implementação do controle central de um robô humanoide, enquanto que em [31] tal plataforma é utilizada para a proposição de uma implementação de controle baseado em regras *fuzzy*.

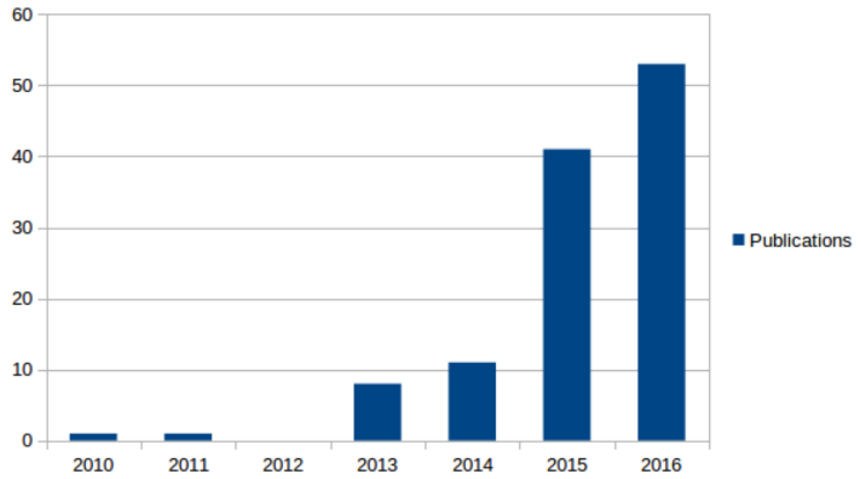

Figura 1.1. 2 Gráfico do crescimento do número de trabalhos publicados que se utilizam da plataforma Parallella e arquitetura multicore Epiphany [15].

Outras abordagens, como o controle preditivo baseado em modelos (MPC, do inglês, Model *Predictive Control*) que possam vir a fazer uso da plataforma *Parallella* são desconhecidas na literatura especializada. Essa técnica de controle ganhou aceitação principalmente em aplicações industriais, permitindo lidar com uma enorme variedade de problemas de controle, com múltiplas variáveis e sujeitos a restrições sobre os sinais de controle, estados internos e saídas [32]. Sua filosofia consiste basicamente em calcular, para cada intervalo de amostragem, um conjunto de ações de controles futuras que irão minimizar uma função custo.

Apesar de sua popularidade na indústria, a demanda computacional intensa (imposta pelo MPC), mostrou-se um fator crítico para aplicações com exigência de um rápido tempo de resposta (desempenho em *timing*) ou que devem ser executadas em plataformas embarcadas de recursos limitados, baixo custo e consumo energético [34]. No entanto, a adoção de tal técnica em aplicações embarcadas, tem crescido cada vez mais nos últimos anos, embalada pelo avanço tecnológico das plataformas de *hardware/software* e, para o caso linear, pela viabilidade em se utilizar algoritmos de otimização quadrática e restrições lineares, conhecidos como QP (do inglês, *Quadratic programming*) [35].

Os problemas QP incorporados ao MPC embarcado e aos recursos de *hardware*, como FPGA's e arquiteturas *multicore* citadas no início do capítulo, estimulou uma extensa pesquisa na comunidade MPC desde a última década, apresentando assim um amplo leque de aplicações sobre as quais se faz uso do MPC.

Nesse sentido, na referência [36] foi desenvolvida uma pesquisa a respeito da utilização em aplicações industriais do MPC (abordado em sistemas embarcados) cujos resultados são ilustrados na Figura 1.1.3.

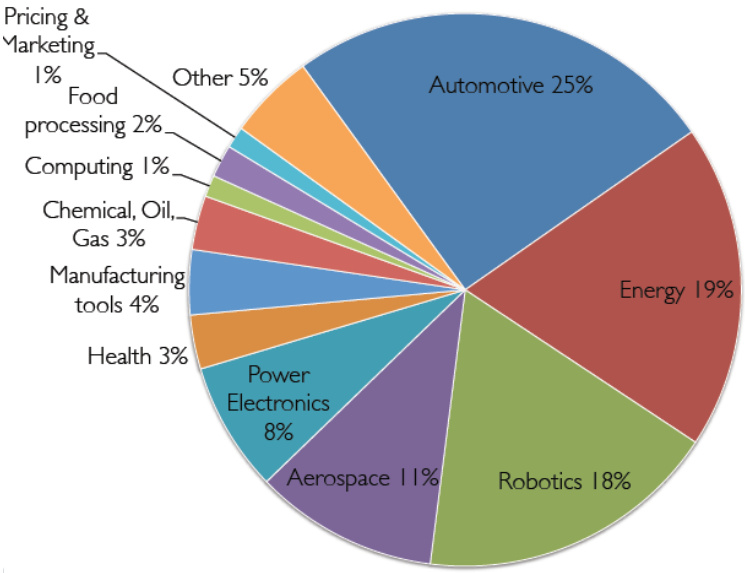

Figura 1.1. 3 Aplicações industriais baseadas em sistemas embarcados [36].

1.2 Motivação do Trabalho

A motivação principal deste trabalho está em explorar os recursos da plataforma *Parallella*, tendo em conta algumas características importantes, tais como: (a) flexibilidade e potencialidade (processador ARM, chip FPGA e coprocessador *Epiphany multicore* em uma única placa), (b) facilidade de desenvolvimento com ferramentas *OpenSource*, (c) eficiência computacional com exploração dos recursos de paralelismo mediante os *cores Epiphany* interligados por NoC, (d) custo reduzido e (e) promessa de baixo consumo energético. Esta ideia de pesquisa vem subsidiada pelo fato de não terem sido encontradas na literatura, aplicações da plataforma *Parallella* na área de MPC.

Neste contexto, o presente trabalho foi concebido no Laboratório LEIA (*Laboratório de Sistemas Embarcados e Aplicações de Circuitos Integrados*) da Universidade de Brasília, que desenvolve atividades de pesquisas voltadas a área de *sistemas embarcados* aplicados a diversos problemas de engenharia, tais como robótica, processamento de imagens, engenharia biomédica e quadrirrotores.

Muitos desses projetos demandam a implementação de algoritmos de controladores sobre plataformas de *hardware* restritas, característica típica dos sistemas embarcados. Nesse sentido, o controle preditivo baseado em modelo (MPC) tem se mostrado um campo de interesse para o grupo de pesquisa do LEIA, embalado pela grande aceitação por parte da comunidade científica, e de sua capacidade de lidar com uma enorme variedade de problemas de controle (com múltiplas variáveis) e sujeitos a restrições, como apresentado no início do deste Capítulo.

O laboratório em questão (LEIA) conta atualmente com dois grandes projetos que possibilitaram a aplicação das técnicas MPC em problemas específicos, a saber: (a) "*Desenvolvimento de um Sistema de Controle em MPSoC, Implementado em FPGA, para um Exoesqueleto Projetado para a Reabilitação de Pessoas com Deficiências Motoras*", financiado pelo CNPq e (b) "*Desenvolvimento de uma Plataforma Reconfigurável para um Sistema de Controle Embarcado para Robótica Móvel",* financiado pela FAPDF.

No contexto dos projetos supracitados, foram desenvolvidos trabalhos iniciais tais como [37], com foco na aceleração em *hardware*, incluindo a implementação de uma Rede Neural Artificial (RNA) em uma FPGA, a fim de tratar o caso do MPC não linear. Os resultados obtidos foram animadores, comprovando o potencial desse tipo de abordagem. No entanto, implementações de arquiteturas de *hardware* demandam, no geral, um tempo maior de desenvolvimento que uma aceleração via *software* (por exemplo usando plataformas *multicores* do tipo MPSoCs) devido a maior dificuldade envolvida em sua concepção. Embasado por tal afirmativa e pelo surgimento de novas plataformas de *hardware multicore*, a aceleração via *software* tem se tornado uma opção interessante a se considerar, devido a sua maior flexibilidade e simplicidade de implementação supracitadas.

Nessa perspectiva a plataforma *Parallella* se mostra promissora, unindo em uma única placa recursos de *hardware* de grande utilização pela comunidade de sistemas embarcados, como o processador ARM e um chip FPGA's, além de trazer como foco um coprocessador *multicore* cujos núcleos são ligados por uma NoC. Tais fatores oferecem uma grande flexibilidade ao desenvolvedor, aliada ainda a facilidade de implementação defendida pelos fabricantes da mesma, com ferramentas de desenvolvimento *OpenSource* e programação facilitada dos núcleos de processamento (linguagem C/C++ e *OpenCL*, dentre outras). Por fim, a plataforma *Parallella* ainda apresenta como ponto forte a eficiência energética e o baixo custo, colocando-a como uma plataforma de *hardware* de grande interesse ao LEIA, não só para os problemas MPC, mas também para as demais aplicações de engenharia [11].

Cabe salientar que a plataforma escolhida tem como restrição a dificuldade do uso da FPGA (para uma aplicação), tendo em conta que a mesma vem pré-configurada para viabilizar a comunicação entre o SoC (ARM) e o MPSoC (*Epiphany*). Neste contexto, este trabalho ficou focado na tentativa (usando uma metodologia experimental) de paralelizar exemplos de MPC (como um estudo de caso) usando os recursos da plataforma escolhida (ARM + *Epiphany*).

Diante de todo o cenário descrito acima, o presente trabalho traz como proposta a avaliação da plataforma *Parallella* mediante a utilização do controle preditivo baseado em modelo (MPC) como um estudo de caso, adotando a metodologia de aceleração via *software* sobre uma abordagem de paralelização experimental.

1.3 Objetivos

De modo a facilitar o entendimento e contribuir para a organização do trabalho, foram divididos os objetivos do mesmo em geral e específicos, como mostrado nas subseções 1.3.1 e 1.3.2 respectivamente.

1.3.1 Objetivo Geral

O objetivo geral deste trabalho está em avaliar a plataforma de arquitetura *multicore Parallella* mediante a utilização do controle preditivo embarcado baseado em modelo como estudo de caso, adotando a metodologia de aceleração via *software* (em um MPSoC) sobre uma abordagem de paralelização experimental.

1.3.2 Objetivos Específicos

A fim de cumprir seu objetivo principal, esse trabalho visa atingir os seguintes objetivos específicos:

- Avaliar características gerais de desempenho da plataforma via *benchmarks*.

- Desenvolver uma estratégia de paralelização do algoritmo MPC proposto, mediante uma abordagem experimental.

1.4 Aspectos Gerais da Metodologia Utilizada

Como forma de alcançar os objetivos propostos, desenvolveu-se uma metodologia baseada em etapas sequenciais com o intuito de se construir o conhecimento necessário para se avaliar a plataforma *Parallella* mediante a paralelização de um algoritmo MPC proposto em [38] sobre uma abordagem experimental. Tais etapas consistem em:

Etapa 1 - Estudo da plataforma *Parallella* (vide Capítulo 2), levantando características de interesse para as aplicações propostas, bem como avaliando as possibilidades de implementação fornecidas pelo *hardware*, além de se analisar dados de *benchmarks* realizados pelo próprio fabricante e relatados na literatura.

Etapa 2 - Criar *benchmarks* (vide Capítulo 4) com o objetivo de se testar operações específicas do *hardware (tais* como operações matriciais) medindo seu desempenho e identificando possíveis gargalos, realizando comparações com resultados divulgados na literatura referente à placa *Parallella*.

Etapa 3 - Realizar um *profile* de uma aplicação de MPC específica (o integrador triplo) a fim de extrair as características de desempenho em uma arquitetura serial (processador ARM).

Etapa 4 - A partir dos resultados do *profile*, definir algumas estratégias de paralelização do algoritmo MPC proposto.

Etapa 5 - Utilizar o exemplo do integrador triplo (pela sua simplicidade) para avaliar as possíveis estratégias de paralelização.

Etapa 6 - Uma vez escolhia da estratégia com melhor desempenho, validar a mesma com outras aplicações MPC.

Vale ressaltar que o algoritmo MPC utilizado foi proposto por [38], sendo seus detalhes discutidos no Capítulo 2. Um ponto importante da aplicação da metodologia proposta é que no *profiling* (vide Etapa 4) foi detectado que a parte mais crítica do algoritmo MPC utilizado era justamente o passo de otimização da função custo (vide Capítulos 2 e 4).

1.5 Estrutura do Trabalho

O presente trabalho divide-se da seguinte maneira:

- **Capítulo 2: Revisão da Literatura** – Aborda as características da plataforma *Parallella*, com destaque a arquitetura *multicore Epiphany*, ressaltando seus principais componentes, seu modo de funcionamento e os trabalhos correlatos, encontrados na literatura. Além disso, são tratados os princípios gerais do modelo de controle preditivo, ressaltando sua filosofia, seus principais elementos e os passos necessários à concepção do algoritmo que efetuará os cálculos dos sinais de controle, além de uma breve revisão dos trabalhos envolvendo o MPC para sistemas embarcados.

- **Capítulo 3: Avaliação das Característi**cas **da Arquitetura *Multicore Epiphany* via *Benchmarks*** - O objetivo deste capítulo é trazer uma série de testes que explorem as características da plataforma de *hardware* utilizada, explorando pontos específicos de funcionamento da mesma que sejam de interesse direto do algoritmo MPC adotado.

- **Capítulo 4: Um Estudo de Caso Utilizando Aplicações de Controle Preditivo Baseado em Modelo** - Esse capítulo traz uma aplicação de MPC baseada no integrador triplo desenvolvido em [38]. O objetivo é utilizar tal aplicação na avaliação de estratégias de paralelização propostas que sejam eficientes do ponto de vista de tempo computacional, aplicando a estratégia de melhor desempenho em uma aplicação do controle de atitude de uma plataforma de satélites.

- **Capítulo 5: Resultados** – Apresenta os resultados obtidos da implementação da aplicação MPC do triplo integrador e da aplicação de controle de atitude de uma plataforma de satélites artificiais.

- **Capítulo 6: Conclusões e Trabalhos Futuros** – Apresentam os principais pontos de discussão frente aos resultados obtidos e aos métodos empregados para obtenção dos mesmos e levanta sugestões de continuidade da pesquisa.

Capítulo 2

Revisão da Literatura

Este capítulo traz uma breve revisão dos conceitos utilizados no presente trabalho. Nas seguintes seções tratar-se-ão as características principais da plataforma de *hardware Parallella*, abordando aspectos de sua arquitetura e utilização pela comunidade cientifica. Em seguida, serão apresentados os conceitos gerais do controle preditivo baseado em modelo e do algoritmo MPC utilizado, proposto em [38], bem como de trabalhos correlatos na área de MPC aplicado a sistemas embarcados.

2.1 Plataforma de Hardware *Parallella* e Arquitetura *Multicore* Epiphany

Existem inúmeras plataformas de desenvolvimento para sistemas embarcados, variando em tamanhos, capacidade de processamento e custos, dentre outros fatores que devem ser levados em conta com base nas especificidades da aplicação que se deseja embarcar. Nesta pesquisa foi utilizada a plataforma *Parallella* [11], contando com uma arquitetura de *hardware* voltada para a implementação de programas paralelizáveis. Os aspectos mais relevantes de tal arquitetura são mostrados nas subseções a seguir.

2.1.1 Descrição da Plataforma *Parallella*

A *Parallella* é uma plataforma desenvolvida com foco em desempenho computacional e energético, baseada no coprocessador *Epiphany multicore* da *Adapteva* (caracterizando um sistema MPSoC), podendo ser usada como um computador *standalone*, um dispositivo embarcado ou em conexão com outras placas de modo a se formar um *cluster*. A placa inclui um chip *Zynq* (do tipo SoC) composto por um ARM A9 *Dual Core* e uma FPGA, suportando várias distribuições populares do Linux como o Ubuntu [61].

O coprocessador Epiphany consiste em uma matriz de processadores RISC (do inglês, *Reduced Instruction Set Computer*) programáveis em linguagem *C/C++* ou por algum *framework* de programação paralela, tal como *OpenCL*, MPI e *OpenMP*. Os núcleos são ligados por uma NoC (do inglês, *Network on Chip*) junto a uma arquitetura de memória compartilhada. A Figura 2.1.1 traz uma visão geral dos componentes (MPSoC e SoC) presentes na plataforma *Parallella*.

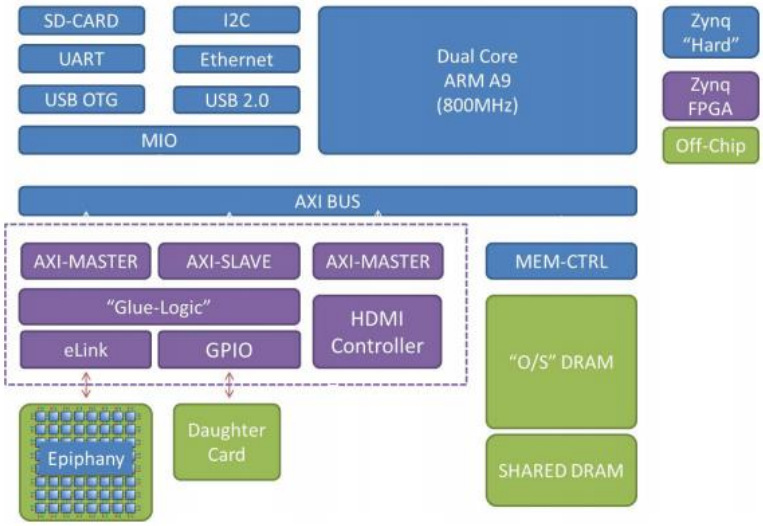

Figura 2.1. 1 Visão geral da arquitetura Parallela [62].

Um resumo das características encontradas nos modelos comerciais da *Parallella* pode ser visto na Tabela 2.2.1, tendo em conta quesitos tais como capacidade de memória, tipos de memória, interfaces, dimensões dos *cores*, entre outras.

Tabela 2.2. 1 Placas Parallella Comerciais [7].

Model	P1600	P1601	P1602
Mnemonic	"Microserver"	"Desktop"	"Embedded"
Host Processor	Xilinx Zynq Dual-core ARM A9 XC7Z010		Xilinx Zynq Dual-core ARM A9 XC7Z020
Coprocessor	Epiphany 16-core CPU E16G301		
Memory	1 GB DDR3		
Ethernet	Gigabit Ethernet		
Boot Flash	128Mb QSPI Flash		
Power	5V DC		
Storage	Micro-SD		
USB	No	USB 2.0 Host Port	
HDMI	No	Micro HDMI	
GPIO Pins	0	24	48
eLink Connectors	0	2	2
FPGA Logic	28K Logic Cells 80 DSP Slices	28K Logic Cells 80 DSP Slices	80K Logic Cells 220 DSP slices
Weight	1.3 oz (36 grams)	1.4 oz (38 grams)	
Size	3.5" x 2.1" x 0.625" (90mmx55mmx18mm)		
SKU	P1600-DKxx	P1601-DKxx	P1602-DKxx
HTS Code (Schedule B)	8471.41.0150	8471.41.0150	8471.41.0150

2.1.2 Arquitetura da Placa *Parallella*

A placa *Parallella* tem como componentes principais um SoC *Zynq* e um coprocessador *Epiphany*. A Figura 2.1.2 ilustra esses dois *chips* juntamente com alguns outros componentes que a compõe, tais como memoria SDRAM (do inglês, *Synchronous Dynamic Random Access Memory*) e conectores.

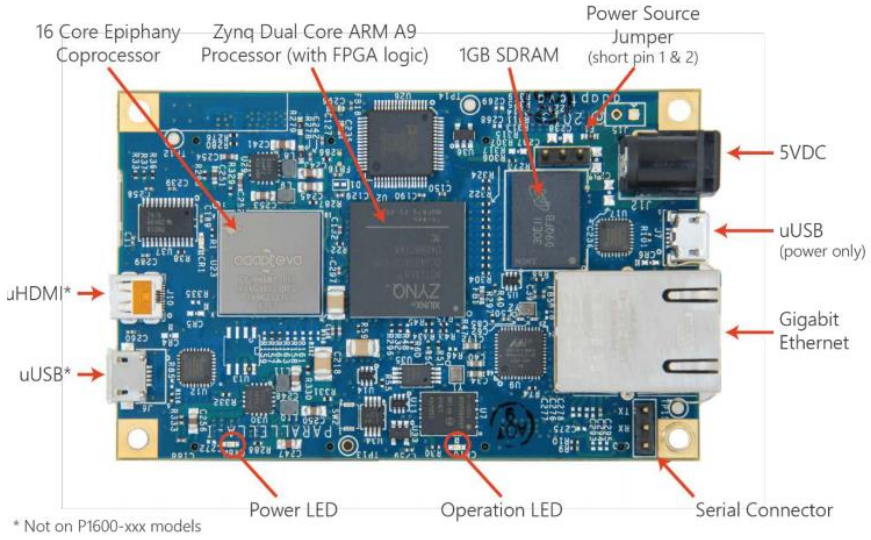

Figura 2.1. 2 Visão superior da placa Parallella [63].

Além dos componentes supracitados, a placa *Parallella* ainda conta com quatro conectores de expansão que permite a comunicação da mesma com outros dispositivos, via protocolo *eLink*, possibilitando a formação de clusters, por exemplo, como mostrado na Figura 2.1.3.

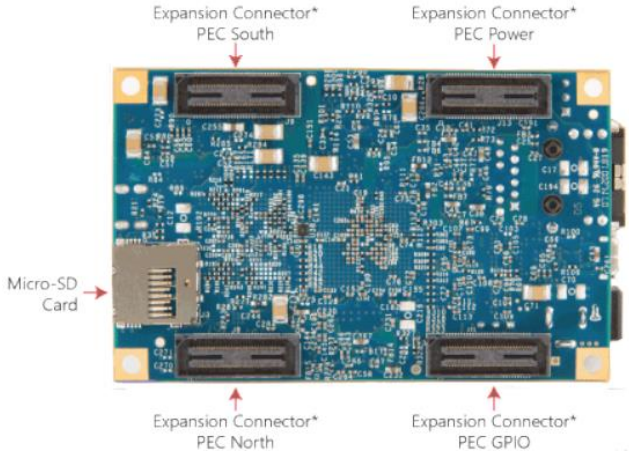

Figura 2.1. 3 Visão Inferior da placa Parallella [63].

A Figura 2.1.4 ilustra as conexões internas a placa *Parallella* do ponto de vista do SoC (do inglês, *System on Chip*) *Zynq*.

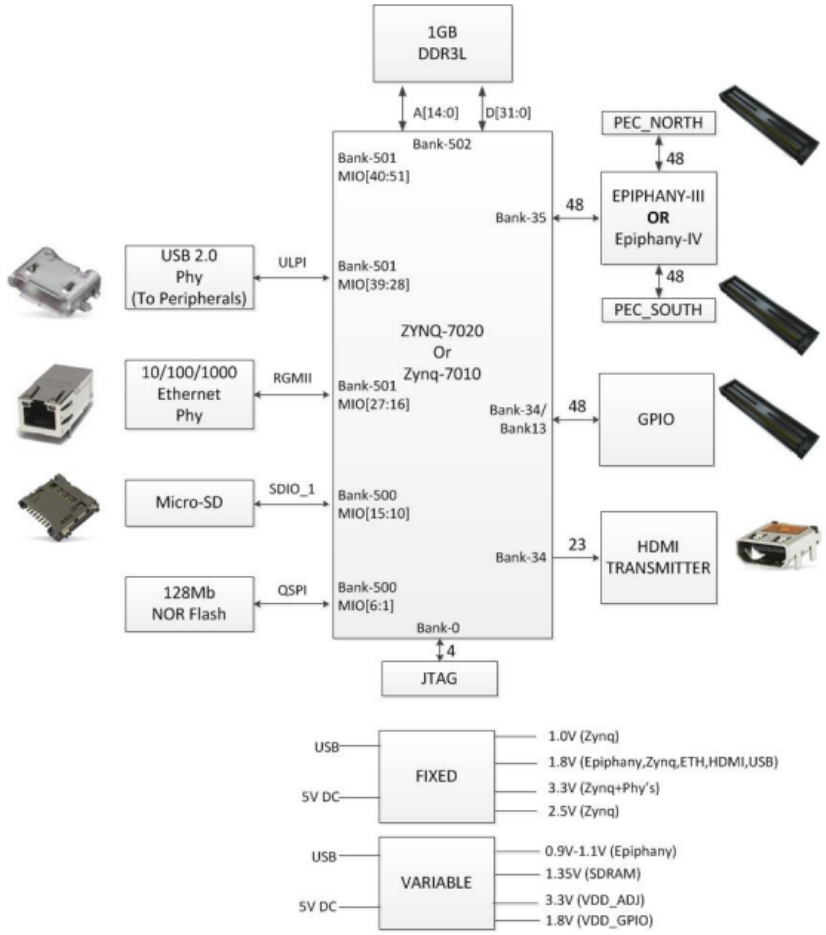

Figura 2.1. 4 Diagrama de conexões internas da placa Parallella [8].

Em uma visão de mais alto nível, a arquitetura da placa *Parallella* pode ser observada como ilustrado na Figura 2.1.5, em que são apresentados os componentes do SoC *Zynq* e do coprocessador *Epiphany*.

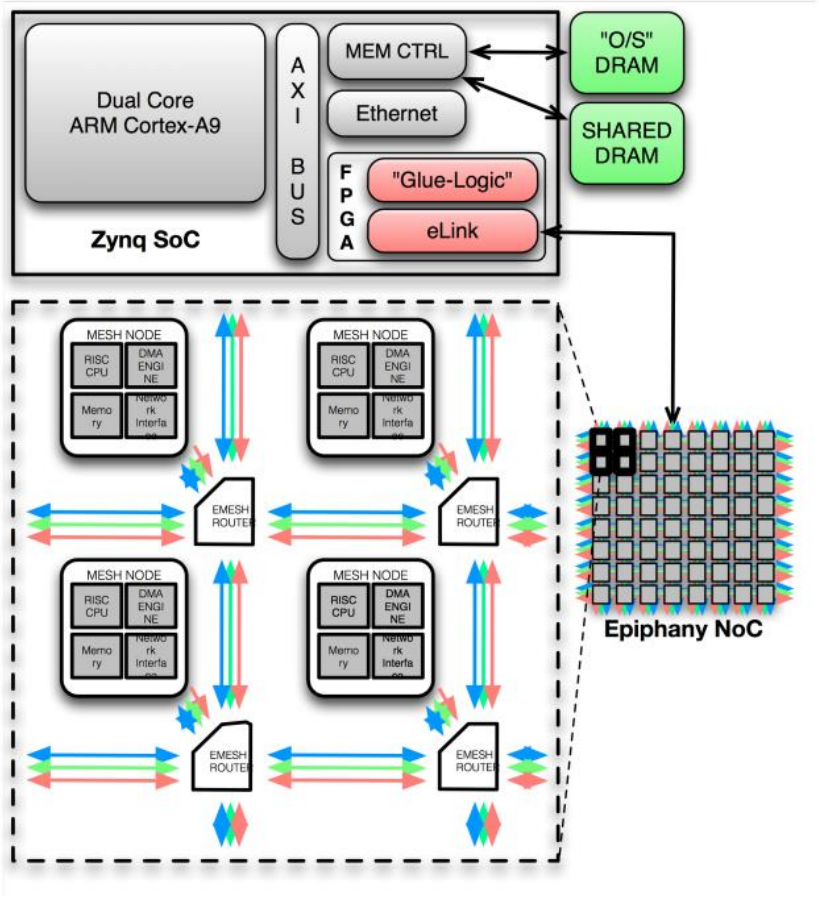

Figura 2.1. 5 Visão em alto nível dos componentes da placa Parallella (Epiphany na versão 64 eCores) [64].

Devido ao aquecimento apresentado principalmente pelo coprocessador Epiphany, a plataforma *Parallella* vem acompanhada de um dissipador térmico que deve ser fixado à placa antes de seu uso. Além desse recurso, é recomendada pelo fabricante a utilização de um *cooler* sobre o dissipador, dando maior eficiência ao processo de resfriamento e garantindo o bom desempenho da placa. A Figura 2.1.6 ilustra o aparato supracitado.

Figura 2.1. 6 Placa Parallella Desktop E16G301 com dissipador térmico e cooler.

2.1.3 Arquitetura *Epiphany*

O elemento principal da placa *Parallella* é o coprocessador *Epiphany* (representando uma arquitetura do tipo MPSoC [33]), contendo núcleos com CPU's RISC superescalar e de ponto flutuante, ligadas por uma NoC. Os modelos existentes no mercado podem vir com 16 ou 64 núcleos. A Figura 2.1.4 da subseção anterior traz uma visão de alto nível acerca da arquitetura *Epiphany* com detalhe para a estrutura interna do núcleo de processamento.

a) Arquitetura de Memória

A arquitetura *Epiphany* usa uma memória interna composta por 2^{32} endereços de 8-*bits*. Cada campo de endereço é composto por uma palavra de 32 *bits*, ou seja, a junção de 4 dos espaços de endereço supracitados. Por exemplo, um endereço *A* consiste de 4 bytes de endereços alinhados, *A, A+1, A+2 e A+3*. Desse modo tem-se 2^{30} espaços de endereços com palavras de 32-*bits*, levando a um espaço final de 1GB.

Cada nó da rede é mapeado em um espaço de 1MB dessa memória, possuindo uma memória interna de 32KB (divididos em 4 bancos de 8KB), um espaço reservado e uma área de memória para mapeamento de registradores. Além disso, cada nó também conta com um identificador (ID), que permite comunicação do mesmo com os outros nós do sistema. O ID do nó é composto por 12 bits sendo 6 correspondentes ao número da linha e 6 ao número da coluna em que se encontram. Esses 12 bits são situados no endereço mais significativo MSB (do inglês, *Most Significant Bits*) do espaço de endereços. A Figura 2.1.7 mostra uma visão geral do espaço de memória da *Epiphany*.

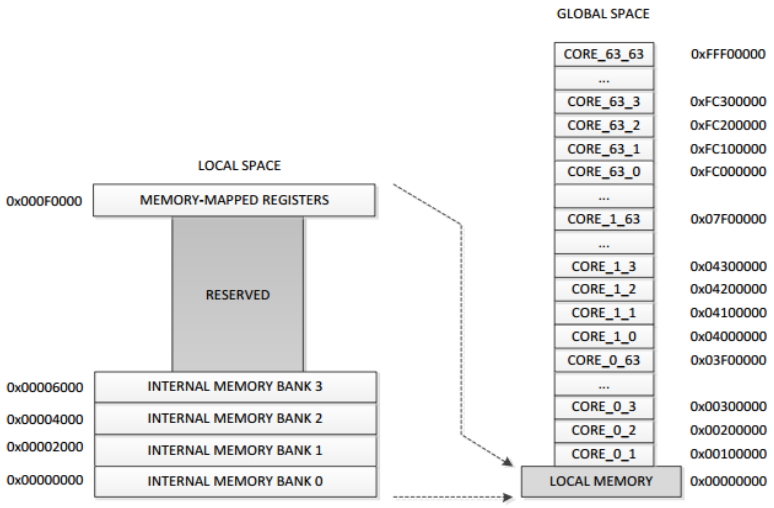

Figura 2.1. 7 Mapa de memória global do coprocessador Epiphany [63].

Os dados são armazenados no banco 0 da memória interna de cada *eCore*.A Figura 2.1.8 mostra o mapeamento de memória para os nós do sistema, com destaque para o endereço de memória dos mesmos e seu mnemônico (linha, coluna).

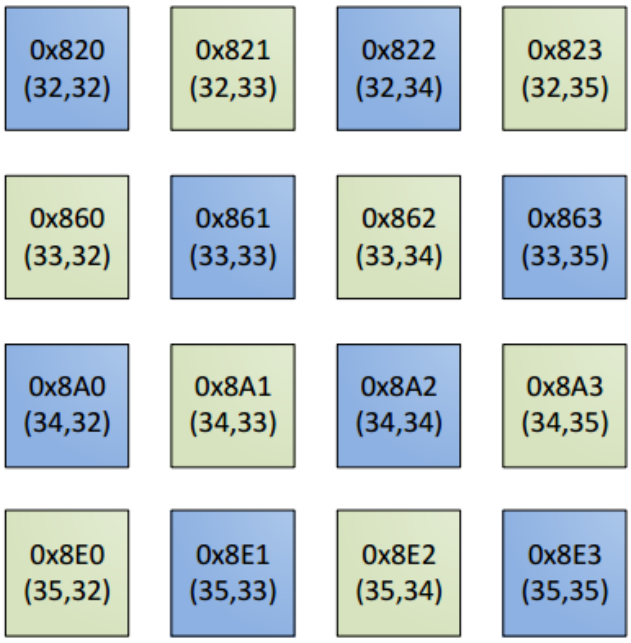

Figura 2.1. 8 Mapeamento de memória para os nós do coprocessador Epiphany (Adaptado [63]).

b) Network on Chip - eMesh

A rede em *chips* (NoC) implementada, denominada de *eMesh*, possui topologia 2D com conexão direta somente entre vizinhos. Cada roteador da rede é conectado ao norte, sul, leste, oeste e ao nó da rede. Transações de escrita na *eMesh* ocorrem com uma latência de 1,5 ciclos de *clock* por etapa de roteamento.

A *eMesh* é dividida em três estruturas de rede separadas e ortogonais, servindo aos seguintes propósitos:

cMesh – Usada para transações de escrita realizadas entre os nós internos a rede.

rMesh – Usada por todas as requisições de leitura.

xMesh – Usada para transações de escritas externas ao chip.

A Figura 2.1.9 ilustra a estrutura da *eMesh* e seus nós.

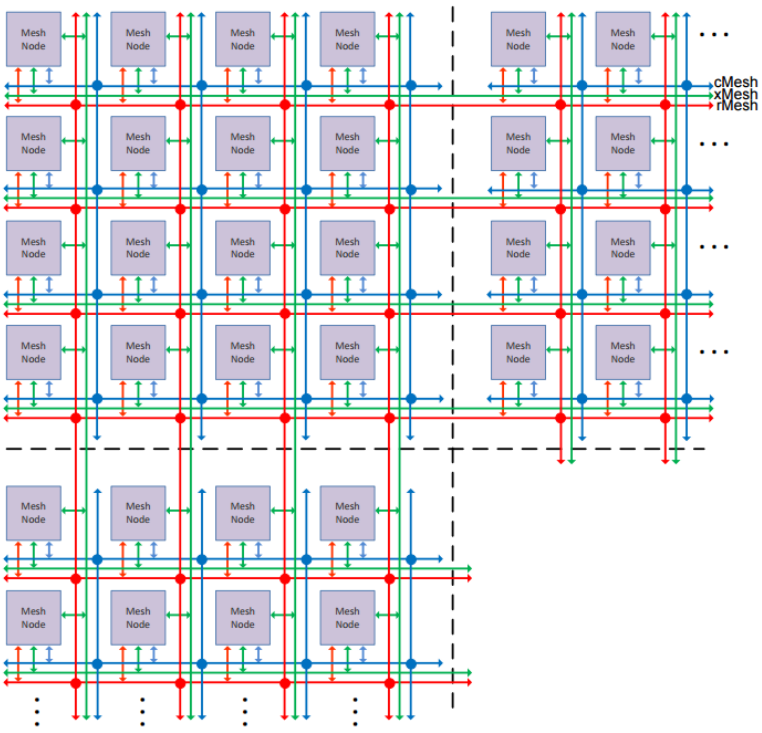

Figura 2.1. 9 Estrutura da NoC *eMesh* [64].

O protocolo de roteamento da rede supracitada é baseado nas coordenadas (x, y), de modo a comparar o ID de destino com o ID do núcleo em questão, fazendo-se primeiro a comparação da coluna seguido da comparação da linha. Dessa forma, a mensagem chega primeiramente à coluna de destino e posteriormente a linha, encontrando o nó desejado.

Na comparação da coluna, caso o ID da coluna de destino seja maior que o ID do núcleo em que a comparação está sendo feita, desloca-se para o nó a leste, caso contrário a oeste. O mesmo procedimento é feito para a linha, deslocando-se para sul e norte, até que se encontre o nó cujo ID corresponda com o ID de destino.

c) Estrutura de um núcleo de processamento

Cada núcleo que compõe a matriz de um coprocessador *Epiphany* é composto por uma CPU RISC, uma memória local de múltiplos bancos, uma *engine* DMA e uma interface de rede, que faz a conexão do núcleo com a rede em chip. A Figura 2.1.10 ilustra a visão geral de um núcleo da *Parallella*.

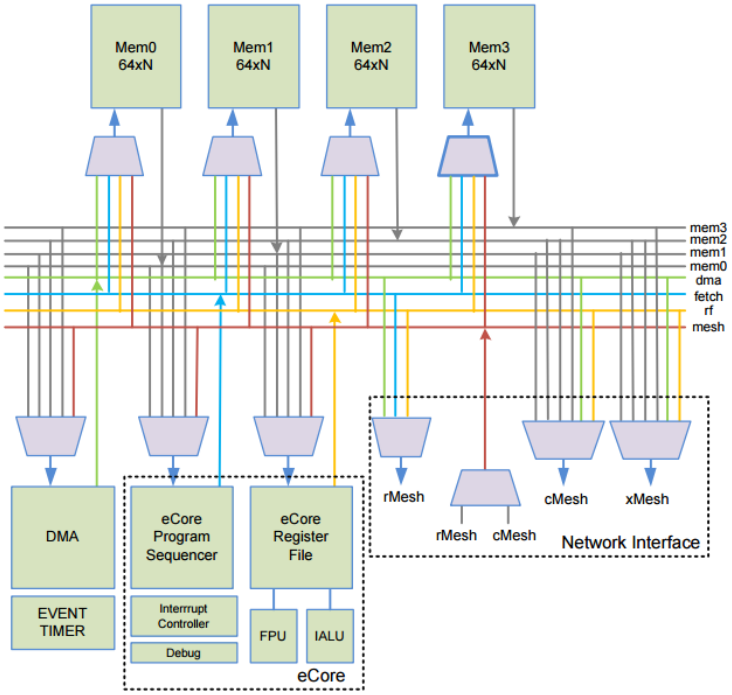

Figura 2.1. 10 Estrutura interna dos núcleos de processamento Epiphany [64].

- ***eCore***

  O coração do núcleo é a unidade de processamento, denominada *eCore* CPU, um microprocessador designado especificamente para processamento *multcore* e melhorado de forma a se achar uma relação entre eficiência energética e desempenho. O núcleo de processamento é comumente denominado *eCore* pela comunidade desenvolvedora.

- **Memória Local**

  A memória local de cada um dos núcleos *Epiphany* é dividida em 4 bancos de 8KB de tamanho (totalizando 32KB de memória local para cada núcleo), que podem ser acessados uma vez a cada ciclo de *clock* operando a uma frequência de 1GHz.

  Tal memória, também chamada de memória interna de cada núcleo, começa no endereço $0x00000000$ terminando em $0x00007fff$, sendo tais endereços vistos do ponto de vista do núcleo. Do ponto de vista externo ao núcleo os endereços são $0xXXX00000$ até $0xXXX07fff$, onde os primeiros 12 bits correspondem ao endereço de mapeamento dos núcleos por parte do dispositivo externo.

- **DMA *Engine***

  A DMA acelera a transferência de dados entre os nós de processamento através da estrutura *eMesh*. A *engine* foi projetada e customizada para a estrutura *eMesh* e opera a mesma velocidade da mesma.

- ***Event Timers***

Cada processador possui dois temporizadores de eventos independentes e de 32 bits que possibilitam monitorar eventos. Esses temporizadores podem ser utilizados para *debug*, otimização de programa, balanceamento de trafego, contadores de tempo, temporizadores de *watchdog* e diversos outros propósitos.

- **Interface de Rede**

  A interface de rede conecta o nó de processamento à estrutura da NoC eMesh. O decodificador da interface de rede é responsável por carregar e gravar instruções, endereço do contador de programas e endereços das transações DMA. Além disso, decide se a transação é destinada a seu processador, caso em que os bits [31:20] são zero, ou à própria *eMesh*.

2.1.4 Memória Compartilhada

A placa *Parallella* utiliza como dispositivo de memória compartilhada entre o SoC *Zynq* e o coprocessador *Epiphany* uma DRAM de 32MB (vide Figura 2.1.5). Tal memória possui endereço físico no intervalo $0x1e000000 \div 0x1fffffff$, visto pelo *host* (definido neste trabalho como o processador ARM A9), e $0x8e000000 \div 0x8fff$ffff visto pela *device* (definido neste trabalho como o coprocessador *Epiphany*).

Tomando como base o ponto de vista do coprocessador Epiphany, os primeiros 16MB da memória compartilhada, *0x8e000000-0x8effffff*, são destinados à biblioteca C, código, dados e pilhas, dentre outros.

Os outros 16MB são divididos em:

- $0x8f000000 - 0x8f7fffff$ = seção *shared-dram* -> 8MB
- $0x8f800000 - 0x8fffffff$ = seção *heap-dram* -> 8MB

Este último intervalo de endereço é dividido em 512KB por *eCore* (16*512KB = 8MB). Esses dados são definidos no *Hardware Description File* (HDF) e nos programas Epiphany no *Linker Description File* (LDF).

2.1.5 Comunicação entre o SoC *Zynq e* o coprocessador *Epiphany*

Os coprocessadores *Epiphany* são ligados a um dispositivo *host* (*Zynq* 7010 ou 7020), usando um ou mais de seus quatro *eLinks* (norte, sul, leste e oeste). Tal conexão se dá pela implementação pré-definida de uma "*Glue-Logic*" e do protocolo *eLink* via FPGA que compõe o SoC *Zynq*, como pode ser visto na Figura 2.1.1 no início do capítulo. Além disso, a FPGA implementa a interface AXI (do tipo mestre/escravo) e também um controlador HDMI.

Como já mencionado na seção anterior, o SoC Zynq e o coprocessador *Epiphany* compartilham uma memória de 32MB por meio da qual é realizada a comunicação entre os

mesmos. A memória compartilhada pode ser mapeada de maneira diferente do ponto de vista do Zynq e do dispositivo *Epiphany*. Na placa *Parallella*, tal memória possui endereço físico no intervalo *0x1e000000÷0x1fffffff*, visto pelo *Zynq*; e *0x8e000000÷0x8fffffff* visto pela *Epiphany*.

O envio e recebimento de mensagens por parte do *Zynq* pode ser feito através de funções específicas (como *e_read/e_write*) da biblioteca da camada de abstração de *hardware* da *Epiphany* (*eHAL*) definida no arquivo *header e-hal.h*.

Desse modo, a aplicação *Zynq* pode se comunicar com o dispositivo *Epiphany* acessando os espaços de memória interna de cada *eCore* ou usando *buffers* na memória compartilhada.

Quando o *Zynq* escreve em um endereço mapeado para um *eCore* da *Epiphany*, o mesmo envia os dados para o *eLink*, que o encaminha para *eMesh* a fim de ser roteado para o *eCore* de destino. Para tanto, deve ser passados os índices da linha e coluna do *core* desejado (além de um objeto do tipo *e_epiphany_t*).

Caso o *Zynq* queira escrever em um *buffer* da memória compartilhada, a fim de que este seja acessado pelos *eCores* posteriormente, este deve utilizar as mesmas funções de leitura e escrita da biblioteca *eHAL* descritas acima, além de um objeto indicando o destino da escrita (*e_mem_t)*. Neste caso, os índices de linha e coluna que compõe a assinatura da função serão desconsiderados.

O processo de leitura é análogo ao descrito acima para a escrita. Uma ressalva se dá por parte dos *eCores*, que não conseguem acessar diretamente a memória do *Zynq*, somente a memória interna à *Epiphany* e a memória compartilhada. Para tanto, podem ser utilizadas as funções específicas (*e_read/e_write*) presentes na biblioteca *eLib*, presente no arquivo *header e-lib.h*.

Denomina-se *workgroup* o conjunto de *eCores* ativos em uma dada aplicação. Caso se queira acessar a memória interna de um dos *eCores* pertencentes ao *workgroup*, deve-se passar como parâmetro da função o objeto do tipo *e_group_config*, juntamente com as coordenadas (linha, coluna) do *eCore*.

Em contrapartida, se usado como parâmetro um objeto do tipo *e_emem_config*, o endereço de destino é tomado relativamente à memória externa e os parâmetros de linha e coluna são desconsiderados.

Vale relembrar que o coprocessador *Epiphany* possui uma memória interna compartilhada (vide Figura 2.1.7), possibilitando que a comunicação entre os seus *eCore's* sejam realizadas mediante o uso de ponteiros, em alternativa ao uso das funções *e_read/e_write* supracitadas.

2.1.6 Modo de programação da placa *Parallella*

A arquitetura *Epiphany multicore* oferece suporte à linguagem *ANSI C/C++*, utilizando o GNU GCC e GDB. Esse é um grande ganho em se tratando de tempo e facilidade de desenvolvimento, uma vez que a linguagem C/C++ é bem difundida entre os desenvolvedores e, por se tratar de uma linguagem de alto nível, oferece maior facilidade e rapidez na construção de código e *debug* do mesmo; poupando o árduo trabalho de se escrever um código em *assembly* por exemplo. Essa é uma das maiores vantagens da *Epiphany* eSDK (do inglês, *Software Development Kit*).

A fim de trazer facilidade ao trabalho, os desenvolvedores podem contar ainda com:

- *OpenCL* SDK
- *Debug* GDB com suporte Multicore
- IDE (do inglês, *Integrated Development Environment*) Eclipse
- Biblioteca de abstração de *hardware* (e-hal) e configuração (e-Lib)
- Simulador Funcional *single core*.

O desenvolvimento de uma aplicação para a plataforma *Parallella* conta com a construção de dois programas distintos, (a) um para ser executado pelo *host* (ARM) e (b) outro para ser executado pelo *device* (*eCores Epiphany*).

Do lado do *host* a aplicação deve gerenciar todo o processo de comunicação com o coprocessador *Epiphany*, tais como a inicialização e *reset* do mesmo, a definição do *workgroup*, a carga dos programas a serem executados nos *eCores* e alguns pontos da gerência dessa execução.

Para tanto, utiliza-se das funções disponíveis na biblioteca de abstração de *hardware* disponibilizada pela SDK *Epiphany* (eSDK), *e-hal*. O *Zynq* também fica encarregado de alocar a memória compartilhada de modo a receber uma estrutura de comunicação, variável com o objetivo da aplicação em questão, que será utilizada para gerenciar o fluxo de execução dos *eCores* e transmitir e receber dados para e dos mesmos.

Do lado do *Epiphany*, os *eCores* recebem o código a serem executado e, mediante a autorização do *Zynq*, dão início a sua execução. Para auxílio nas obtenções da configuração de *hardware* é utilizada a biblioteca *eLib*. A comunicação com o *Zynq* é feita via memória compartilhada, seguindo o padrão de estrutura pré-configurado pelo mesmo.

A Figura 2.1.11 ilustra as camadas envolvendo os lados *Zynq* e *Epiphany* da plataforma *Parallella*

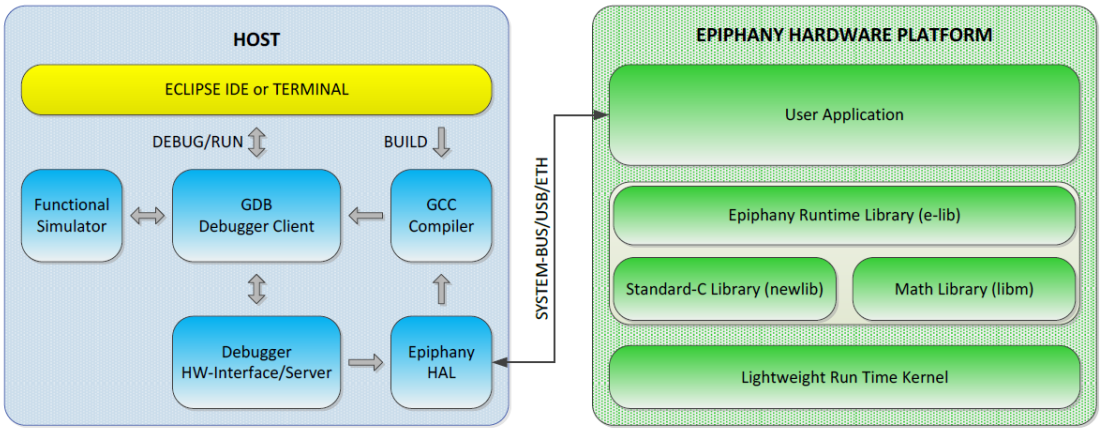

Figura 2.1. 11 Comunicação entre Zynq e Epyphany em aplicações embarcadas na placa Parallella [63].

De modo a facilitar o desenvolvimento em termos de compilação e *debug* para ambas as partes envolvidas, pode-se utilizar uma IDE de desenvolvimento. A documentação da eSDK sugere o uso da ferramenta ECLIPSE, todavia, neste trabalho foi utilizado o *Code::Blocks*, cujas configurações de *setup* estão resumidas no Anexo A.

Caso não haja interesse na utilização de IDEs, pode-se optar pelo desenvolvimento via terminal, com a criação de scripts e *makefiles* na compilação e *debug* dos códigos.

2.1.7 Sincronização entre *eCores*

Em se tratando de programação *multicore*, deve-se atentar de maneira especial para a sincronização entre os *eCores*, uma vez que os mesmos executam códigos em paralelo, concorrendo por recursos compartilhados. Desse modo, dois mecanismos são utilizados com a finalidade de se estabelecer tal sincronia, a saber, (a) o *barrier* e (b) o *mute*x.

a) Barreira Global

Barreiras (*Barriers)* são objetos compartilhados em um *workgroup* a fim de prover a sincronização global dos *eCores*. Quando um programa que está sendo executado em um *eCore* chega a um *barrier* ele interrompe sua execução e aguarda até que todos os outros *eCores* do *workgroup* alcancem esse ponto comum. Feito isso, todos os *eCores* continuam seus respectivos fluxos de execução. Essa é uma ferramenta importantíssima para prevenir os problemas de condição de corrida [79].

Pode ser observado que caso um *core* tenha terminado sua tarefa, o mesmo entra em estado de *wait*, a fim de cumprir os requisitos de sincronização. A biblioteca *eLib* fornece funções para inicialização e configuração de *barries*, cujo funcionamento descrito acima é verificado na Figura 2.1.12.

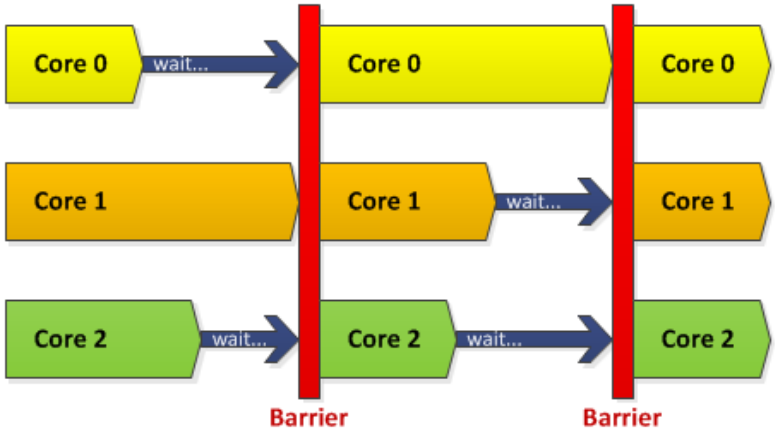

Figura 2.1. 12 Diagrama de funcionamento do recurso de barrier [64].

b) Mutex Lock

O *mutex lock* é um mecanismo utilizado para garantir o acesso único a um recurso compartilhado, ou seja, que somente um *eCore* por vez consiga acessar um recurso comum aos *eCores* do *workgroup*. Ele funciona com base na definição de um endereço na memória local *Epiphany* (para o *workgroup)*, consultado por cada um dos *eCore*s.

Assim, quando um *eCore* requer acesso a um determinado recurso compartilhado o mesmo checa o estado desse espaço de memória e verifica sua disponibilidade. Caso esteja bloqueado o *eCore* aguarda até sua liberação, caso esteja livre, o *eCore* muda seu *status* para bloqueado e faz uso do recurso em questão, liberando-o ao final para que outros *eCores* possa fazer uso desse mesmo recurso.

A exemplo dos *barries*, a biblioteca *eLib* também oferece funções de inicialização e configuração do *mutex* , cujo modo de operação é ilustrado pela Figura 2.1.13.

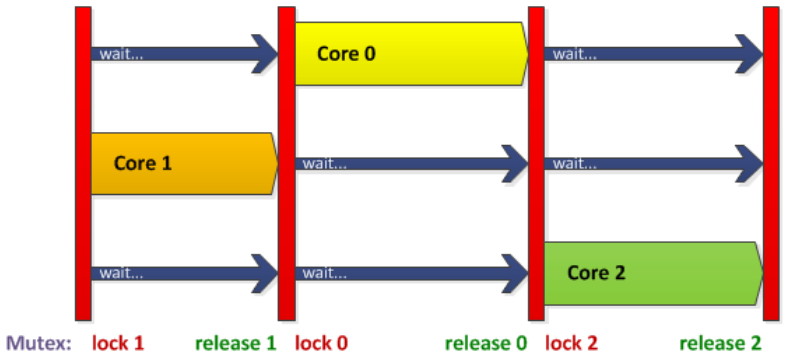

Figura 2.1. 13 Diagrama de funcionamento do mutex [64].

2.1.8 Elementos de Computação Paralela

Entende-se arquitetura paralela como o conjunto de elementos de processamento que cooperam e se comunicam buscando a resolução de grandes problemas rapidamente [86]. Sendo

assim, o desenvolvimento de aplicações sobre tais arquiteturas se dá mediante os princípios de programação paralela.

O principal intuito da programação paralela está em permitir um ganho de desempenho dos programas assim estruturados frente a um programa equivalente e implementado de forma serial. Tal ganho tem sido cada vez mais desejado por diversas áreas da ciência, variando desde o ramo comercial, com produtos que necessitam de um alto poder de processamento atrelado a um baixo custo, até aos pesquisadores que lidam com problemas de grandes dimensões e que exigem um alto esforço computacional.

Na teoria, o melhor desempenho, tomando como referência o tempo de execução, que se pode obter de um programa dividido em tarefas independentes a serem executadas em uma arquitetura composta por *p* unidades de processamentos, sendo que cada tarefa é executada em uma dessas unidades, é dado por:

$$T_{paralelo} = \frac{T_{serial}}{p} \qquad (2.1.1)$$

Em que $T_{paralelo}$ refere-se ao tempo total de execução do programa implementado de forma paralela, T_{serial} refere-se ao tempo de execução do programa implementado de forma serial e p se refere ao número de núcleos de processamento existentes [79].

Quando a relação expressa pela equação (2.1.1) acima é obedecida dizemos que o programa apresenta um aumento de velocidade linear, o que se conhece como *linear speedup*. Na prática não é possível conceber programas que obedeçam a esse ganho de velocidade linear, visto que a comunicação entre núcleos de processamento e suas tarefas incluem um custo extra em termos de tempo de processamento denominado tempo de *overhead*. Programas de memória compartilhada, por exemplo, sempre terão uma seção crítica que necessitará do uso de uma estrutura de controle de acesso aos recursos compartilhados como o *mutex lock*. A chamada ao *mutex* inclui um *overhead* que não estaria presente em uma versão serial do mesmo programa, ao passo que insere um ponto de serialização nesta seção, dada como crítica, no programa concebido sobre os preceitos do paralelismo [79].

Do mesmo modo, sistemas com arquitetura de memoria distribuída trazem a necessidade de se transmitir dados através da rede que interliga os processadores ou *cores*, o que é significativamente mais lento que o acesso à memória local. Fato que também não ocorre em programas seriais.

Assim, fica claro que em termos de aplicações práticas, o aceleramento obtido pela equação 2.1.1 se mostra inviável e inatingível, devido aos tempos de *overhead* necessários aos programas paralelizáveis. Esse *overhead* aumenta conforme o número de *threads* e processadores presentes no sistema.

- ***Speedup***

O ganho de desempenho em termos de velocidade de processamento de implementações paralelas frente a suas versões seriais, conhecido como *speedup,* é comumente expresso como:

$$S = \frac{T_{serial}}{T_{paralelo}} \qquad (2.1.2)$$

ou seja, pelas equações *(2.1.1)* e *(2.1.2)* tem-se que $S = p$, que se sabe ser não aplicável em termos práticos. Mediante os apontamentos supracitados, espera-se que o crescimento de p demande um decréscimo de S. Outra forma de abordagem é dizer que S/p irá se tornar cada vez menor com o aumento de p.

Além do número de processadores ou *threads* p, o tempo $T_{paralelo}$ também é influenciado pelo tamanho do problema. Levando em consideração o tempo de *overhead* encontrado na vida real, pode se dizer que:

$$T_{paralelo} = \frac{T_{serial}}{p} + T_{overhead} \qquad (2.3.4)$$

A mediada em que se aumenta o tamanho do problema também se aumenta o tempo de *overhead* do programa paralelo. Todavia, se adotada uma boa técnica de paralelização, este cresce mais lentamente que o tempo serial, justificando o uso da abordagem paralela.

2.1.9 Trabalhos Correlatos Utilizando a Placa *Parallella*

A plataforma *Parallella* tem como ponto central um coprocessador *Epiphany multicore*, apresentado em 2010 pelos autores do projeto em [13], onde são descritos os princípios de sua arquitetura (versão com 16 *cores*) e realizados *benchmarks* para levantamento de desempenho computacional, energético e de velocidade de comunicação entre os 16 *cores* via NoC (*eMesh*). Em 2011 os autores de projeto apresentaram um segundo trabalho em que reforçam a ideia seminal de expansão do número de *cores* internos ao coprocessador *Epiphany*, apresentando uma proposta de implementação de até 1024 *cores* [14].

Desde de a fabricação da primeira versão da placa no ano de 2013, contando com Zynq-7020 dual-core ARM A9 CPU e versões do coprocessador Epiphany com 16 e 64 *cores* com 32KB/*core* de memória interna, muitos trabalhos têm surgido com o intuito de explorar essa arquitetura.

Motivados pelo fato de se tratar de uma plataforma de *hardware* recente, e consequentemente ainda pouco explorada, muitos dos trabalhos desenvolvidos tratam de um estudo inicial do

funcionamento da placa e de suas principais características, como apresentado em [16], [17], [65] e [66], mostrando uma visão geral da arquitetura e das suas potencialidades.

Outra consequência de se tratar de uma plataforma recente diz respeito às ferramentas de desenvolvimento. Apesar de o projeto ter como um dos seus pilares a utilização de ferramentas *OpenSource* e fornecer uma SDK ao usuário, muitos pesquisadores têm concentrado seus esforços no desenvolvimento e aprimoramento de ferramentas de *software* e *frameworks*. Dentre estes últimos, as áreas de aplicação variam entre: (a) visão computacional [12], (b) processamento de imagens [13], (c) processamento de sinais [14], (d) *Deep Learning* [15], (e) Redes Neurais Artificiais [16], (f) algoritmos bio-inspirados [17], (g) UAN's (*Unmanned Aerial Vehicle*) [18], (h) IoT (do inglês, *Internet of Things*) [19], (i) segurança da informação [20], dentre outros.

Tendo em conta os trabalhos correlatos da plataforma Parallella na área de engenharia, assim como sua escassa utilização em aplicações de controle, pode se salientar a relevância do foco desta dissertação, no contexto de testar as particularidades dos recursos desta plataforma na área de controle.

Em se tratando do uso da plataforma Parallella para a área de controle, são encontradas poucas referências na literatura, indicando que este ainda é um ramo pouco explorado para esta arquitetura em questão. Por exemplo, trabalhos como [21] utilizam a placa Parallella na implementação do controle central de um robô humanoide, enquanto que em [22] tal plataforma é utilizada para a proposição de uma implementação de controle baseado em regras fuzzy.

2.2 Controle Preditivo Baseado em Modelo (MPC)

O Controle Preditivo Baseado em Modelo (MPC ou MBPC, do inglês, *Model Based Predictive Control*) se originou em meados da década de setenta, sendo tema recorrente de pesquisas desde então [39]. Essa abordagem de controle apresenta como conceito básico a utilização de um modelo dinâmico para prever o comportamento do sistema e produzir a melhor decisão (ação de controle no instante atual), ganhando aceitação principalmente em aplicações industriais, permitindo lidar com uma enorme variedade de problemas de controle, múltiplas variáveis e a restrições sobre os sinais de controle, estados internos e saídas [32].

Tendo em conta que o MPC é utilizado (nesta dissertação) como um estudo de caso, nesta seção serão apresentados os conceitos básicos relacionados com o MPC (de maneira resumida). Para um estudo mais detalhado sobre esta técnica podem ser consultadas referências específicas, tais como [32] e [40].

A maioria das leis de controle, como PID (do inglês, *Proportional–Integral–Derivative*), não consideram explicitamente as implicações futuras decorrentes das ações de controle atuais, diferente do MPC, que por outro lado, calcula explicitamente o comportamento previsto para um horizonte pré-estabelecido, denominando horizonte de predição. Essa última abordagem permite, portanto, restringir a escolha das entradas de controle (no instante corrente) àquelas que não conduzam a um comportamento indesejado do sistema no futuro, como uma ação de controle inviabilizada pela limitação dos atuadores [40].

Assim, o MPC está baseado nas seguintes ações:

a) Uso explícito do modelo para a predição das saídas do processo em instantes futuros, denominado horizonte de predição.
b) Cálculo de uma sequência de controle que minimize uma função custo.
c) Aplicação do primeiro sinal de controle calculado, recalculando todo o horizonte a cada novo instante

2.2.1 Estratégia do Controle Preditivo Baseado em Modelo

A metodologia utilizada por controladores que faze uso das técnicas MPC é caracterizada pela seguinte estratégia:

- No instante de decisão k, medir o estado do sistema;
- Calcular a sequência de ações futuras de controle que minimizem uma a função custo pré-definida;
- Aplicar a primeira ação de controle da sequência calculada durante o instante $[k, k+1]$;
- No próximo instante de tempo $k + 1$, medir o estado do sistema e refaz os passos supracitados.

Um modelo em diagrama de blocos da estratégia descrita pode ser verificado na Figura 2.2.1.

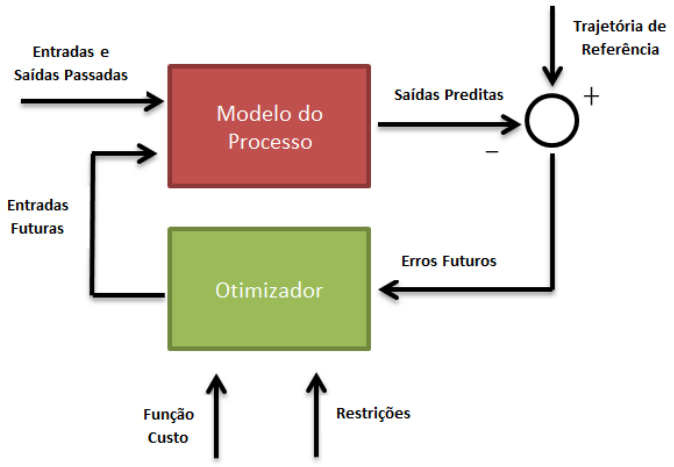

Figura 2.2. 1 Modelo em diagrama de blocos da estratégia MPC [39].

Na Figura 2.2.1, o modelo é utilizado na predição das saídas futuras baseando-se no estado atual do sistema (medido pelos valores passados e correntes), assim como na sequencia ótima de ações de controle. Tal sequência é calculada pelo bloco "otimizador", valendo-se de uma função custo e das restrições do sistema.

A seguir será apresentada uma breve formalização matemática dos elementos gerais do MPC, tendo como base o desenvolvimento apresentado em [38].

a) Predição

Em sistemas de engenharia a predição do comportamento futuro do sistema pode ser realizada através do seu modelo dinâmico discreto e representada por:

$$x(k+1) = f\big(x(k), u(k)\big), \qquad (2.2.1)$$

em que:

$x \in \mathbb{R}^n$: Representa o vetor de estados, ou seja, um vetor cujos componentes descrevem o estado do sistema.

$u \in \mathbb{R}^{n_u}$: Representa o vetor de entradas de controle.

$f: \mathbb{R}^n \, x \, \mathbb{R}^{n_u} \rightarrow \mathbb{R}^n$: Representa o modelo do sistema, permitindo calcular o próximo estado do sistema mediante ao estado atual e a entrada de controle.

$k \in \mathbb{N}$: Representa o instante amostrado;

Uma ação de controle para N estados futuros ($N = horizonte\ de\ predição$) no instante k, pode ser representada por $\tilde{u}(k)$, definido como:

$$\tilde{u}(k) = \begin{pmatrix} u(k) \\ u(k+1) \\ \vdots \\ u(k+N-1) \end{pmatrix}. \qquad (2.2.2)$$

A representação dada por (2.2.2) pode ser estendida ao vetor de estados do sistema para um dado instante k:

$$\tilde{x}(k) = \begin{pmatrix} x(k+1) \\ x(k+2) \\ \vdots \\ x(k+N) \end{pmatrix}. \qquad (2.2.3)$$

Estendendo o conceito apresentado pela equação (2.2.1) para N instantes futuros (horizonte de predição) obtém-se a seguinte equação:

$$\tilde{x}(k|\tilde{u}) = X(x(k), \tilde{u}, \tau, N), \qquad (2.2.4)$$

em que τ é o período de amostragem e $\tilde{x}(k|\tilde{u})$ é a trajetória de estados em um instante k, dado uma sequência ótima de futuras ações $\tilde{u}(k)$ dentro de um horizonte de predição N, sendo representada pela função de mapeamento X.

De modo a facilitar a representação matemática, adota-se a seguinte notação para a representação de dos estados do sistema em um certo instante de tempo $(k + i)$:

$$x(k+i) = \prod_{i}^{(n,N)} \tilde{x}(k); \quad i \in \{1, \dots, N\}, \qquad (2.2.5)$$

em que a matriz de seleção $\prod_i^{(n,N)}$ é esquematizada na Figura 2.2.2.

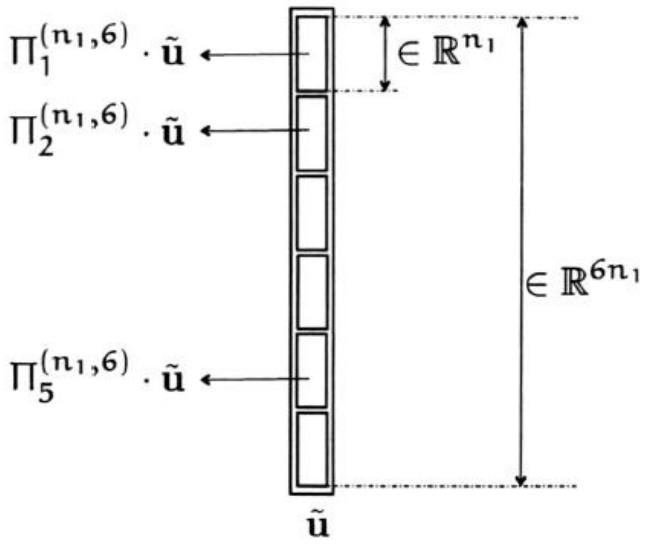

Figura 2.2. 2 A matriz de seleção $\prod_i^{(n_1,N)}$ seleciona o i-ésimo vetor de dimensão n_1 de um vetor composto pela concatenação de N outros vetores . A figura ilustra o caso especial em que N é igual a 6 [38].

b) Função Custo

A estratégia MPC tem como parte de sua implementação o cálculo da sequência de controle que minimize uma função custo para uma determinada trajetória dentro do horizonte de predição pré-estabelecido. Assim, associar as possíveis trajetórias a um escalar facilita a comparação das mesmas e a escolha da sequência de controle a ser escolhida.

$$\{\tilde{x}(k), \tilde{u}(k)\} \rightarrow escalar\ . \qquad (2.2.6)$$

A função custo, representada por uma função genérica J, deve ser definida segundo os interesses de cada aplicação, sendo dependente basicamente de $\tilde{u}(k)$ e da trajetórias de estados no instante k $\tilde{x}(k|\tilde{u})$, definida na equação (2.2.4). Assim:

$$J\big(\tilde{u}(k), \tilde{x}(k|\tilde{u})\big). \qquad (2.2.7)$$

A função custo representada por (2.2.7) pode ser entendida como uma função que fornece um valor escalar dado como argumento o vetor $\tilde{u}$ e o estado atual $\tilde{x}(k)$.

c) Restrições

Os problemas reais vêm sempre acompanhados de restrições, que devem ser incorporadas na escolha do vetor de saída de controle:

$$u(k+i) \in [-u^{max}, +u^{max}]. \qquad (2.2.8)$$

Assim, em termos de restriçoes sobre os sinais de controle, somente as saídas u que satisfazem a expressão (2.2.9) são elegíveis para minimizar a função custo.

Para tanto, define-se para as restrições a função da forma:

$$g\big(\tilde{u}\big|x(k)\big) \leq 0. \qquad (2.2.9).$$

Desse modo, um problema $\mathrm{P}\big(\mathrm{x(k)}\big)$, que será resolvido em um instante de tempo k, deve atender a:

$$P\big(x(k)\big) = \min J\big(\tilde{u}\big|x(k)\big) \; em \; que \;\; g\big(\tilde{u}\big|x(k)\big). \qquad (2.2.10)$$

Assim, a solução ótima $\mathrm{P}\big(\mathrm{x(k)}\big)$ depende do estado atual e consiste em:

$$\tilde{u}^{opt}\big(x(k)\big). \qquad (2.2.11)$$

Existem ainda restrições sobre a variação de tais de controle e sobre variáveis de estado, que serão abordadas na seção 2.2.2.

2.2.2 Fundamentação do Algoritmo MPC Utilizado

Vale salientar que no presente trabalho traz como foco a avaliação da plataforma *Parallella* utilizando aplicações MPC como um estudo de caso. Nesse sentido, não se encontra entre seus

objetivos o desenvolvimento de um algoritmo MPC e sim a utilização de um algoritmo proposto na literatura.

Assim, mediante a necessidade de acesso a todas as operações do algoritmo a ser utilizado, visando sua paralelização, optou-se por adotar a o algoritmo proposto em [38] para aplicações MPC lineares e invariantes no tempo (LTI, do inglês, *Linear Time Invariant*).

Em resumo, as aplicações aqui abordados são àquelas as quais se pode definir um problema de otimização $P(x(k))$ para o qual a função custo é quadrática (e portanto convexa) enquanto que as restrições representadas pela função g, definida pela inequação (2.2.9) na seção 2.2.1, são lineares na variável de decisão $\tilde{u}$ (vetor de controle). Tais problemas são conhecidos como QP (do inglês, *Quadratic Programming*) [38].

A seguir serão desenvolvidos os aspectos fundamentais que dão base para a construção do algoritmo utilizado. Todo o desenvolvimento foi baseado em [38], referência que pode ser consultado para maiores detalhes.

a) Representação do Modelo em Espaço de Estados

A representação do modelo de um sistema em espaço de estados, sobre a qual está concebido o algoritmo MPC utilizado neste trabalho, envolve todo o vetor de estados x, se distinguindo de modelos baseados em função de transferência em que apenas entrada e saída são envolvidas, sendo aquela uma representação mais geral que esta. Valendo-se dessa informação, podem-se definir modelos LTI como àqueles em que a obtenção de um estado em terminado instante $k+1$ pode ser escrita da forma:

$$x(k+1) = Ax(k) + Bu(k), \tag{2.2.13}$$

em que x *e* u estão definidos na seção 2.2.1 e A *e* $B \in \mathbb{R}^{nxn}$ são matrizes constantes de estado e entradas respectivamente.

Expandindo a representação em (2.2.13) para os instantes $(k+2), (k+3), \ldots, (k+i)$ e valendo-se das definições dadas por (2.2.2) e (2.2.3), chega-se a seguinte expressão para um instante de tempo genérico $(k+i)$:

$$x(k+i) = \left[A^i\right]x(k) + \left[A^{i-1}B, \ldots, AB, B\right]\begin{pmatrix} u(k) \\ . \\ . \\ . \\ u(k+i-2) \\ u(k+i-1) \end{pmatrix}. \tag{2.2.14}$$

Como forma de simplificação, define-se as matrizes $\Phi_i \in \mathbb{R}^{nxn}$ e $\psi_i \in \mathbb{R}^{nx(Nn_u)}$ como:

$$\Phi_i := A^i \qquad (2.2.15)$$

e

$$\psi_i := \left[A^{i-1}B, \dots, AB, B\right] \begin{pmatrix} \prod_1^{(n_u,N)} \\ . \\ . \\ . \\ \prod_{i-1}^{(n_u,N)} \\ \prod_i^{(n_u,N)} \end{pmatrix}, \qquad (2.2.16)$$

mostrando que as matrizes Φ_i e ψ_i são constantes e dependem apenas das matrizes A e B. Lembrando-se que n_u é a dimensão do vetor de controle u.

Uma vez que a matriz de seleção $\prod_i^{(n_u,N)}$ foi definida na seção 2.2.1 e que:

$$\begin{pmatrix} u(k) \\ . \\ . \\ . \\ u(k+i-2) \\ u(k+i-1) \end{pmatrix} = \begin{pmatrix} \prod_1^{(n_u,N)} \\ . \\ . \\ . \\ \prod_{i-1}^{(n_u,N)} \\ \prod_i^{(n_u,N)} \end{pmatrix} \tilde{u}(k). \quad (2.2.17)$$

Assim, a nova representação para (2.2.14) pode ser dada por:

$$x(k+i) = \Phi_i x(k) + \psi_i \tilde{u}(k). \qquad (2.2.17)$$

b) A Função Custo

Como apresentado brevemente na seção 2.2.1, a função custo é uma função escalar sobre a variável de decisão $\tilde{u}$ (sequência de futuras ações) que facilita o cálculo, a cada instante de decisão k, da sequência de ações $\tilde{u}^{opt}\big(x(k)\big)$.

De modo a definir uma função custo durante um horizonte de predição $[k, k+N]$, adota-se um vetor de saídas $y_r \in \mathbb{R}^{n_r}$, contendo n_r combinações do vetor de estados. Tal vetor é definido por uma matriz de saídas $C_r \in \mathbb{R}^{n_r xn}$ da seguinte forma:

$$y_r = C_r x \in \mathbb{R}^{n_r} . \tag{2.2.18}$$

Mais precisamente, o objetivo está em manter uma sequência de saídas futuras:

$$\tilde{y}_r(k) = \begin{pmatrix} y_r(k+1) \\ . \\ . \\ . \\ y_r(k+N) \end{pmatrix}, \tag{2.2.19}$$

o mais próximo possível de uma dada referência:

$$\tilde{y}_r^d(k) = \begin{pmatrix} y_r^d(k+1) \\ . \\ . \\ . \\ y_r^d(k+N) \end{pmatrix}. \tag{2.2.20}$$

A saídas futuras $y_r(k+i)$ dependem dos estados futuros $x(k+i)$ de forma que:

$$y_r(k+i) = C_r x(k+i). \tag{2.2.21}$$

Substituindo a equação (2.2.17) na equação (2.2.21), se obtém:

$$y_r(k+i) = \mathrm{C_r}\Phi_i x(k) + C_r \psi_i \tilde{u}(k). \tag{2.2.22}$$

Uma vez que o que se deseja é que as saídas estejam o mais próximas possível de uma certa referência, como já comentado, pode-se definir que a minimização a ser realizada é representada por:

$$\sum_{i=1}^{N} \| y_r(k+i) - y_r^d(k+i) \|_{Q_y}^2 + \sum_{i=1}^{N} \| \prod_{i}^{(n_u,N)} \tilde{u} - u^d \|_{Q_u}^2 , \tag{2.2.23}$$

em que $Q_y \in \mathbb{R}^{n_r \, x \, n_r}$ é uma matriz de pesos quadrada e simétrica, utilizada para definir a penalidade do erro em se seguir a referência. $Q_u \in \mathbb{R}^{n_u \, x \, n_u}$ é uma matriz simétrica positive e u^d é um valor fixo de entrada, podendo ser considerado como o valor de controle do estado. Assim, o segundo somatório da equação (2.2.23) penaliza grandes variações do sinal de controle a fim de forçar as condições necessárias para que a função custo não decresça indefinidamente ou permaneça invariante.

Assim, a função custo definida na seção 2.2.1passa a ser escrita como:

$$J(\tilde{u}|x(k), \tilde{y}_r^d(k)) = \sum_{i=1}^{N} \| C_r\Phi_i x(k) + C_r\psi_i\tilde{u}(k) - y_r^d(k+i) \|_{Q_y}^2 + \sum_{i=1}^{N} \| \prod_i^{(n_u,N)} \tilde{u} - u^d \|_{Q_u}^2 . \quad (2.2.24)$$

Expandindo o somatório em (2.2.24) chega-se a função quadrática sobre a variável de decisão $\tilde{u}$:

$$\frac{1}{2}\tilde{u}^T H\tilde{u} + [F_1 x(k) + F_2\tilde{y}_r^d + F_3 u^d]\tilde{u} + Cste, \quad (2.2.25)$$

em que a matriz quadrada $H \in \mathbb{R}^{Nn_r \, x \, Nn_r}$ é a matriz hessiana dada por:

$$H = 2\sum_{i=1}^{N}[\psi_i^T C_r^T Q_y C_r \psi_i + (\prod_i^{(n_u,N)})^T Q_u \left(\prod_i^{(n_u,N)}\right)], \quad (2.2.26)$$

devendo essa ser positiva, visto que o problema em questão é de minimização.

As matrizes $F_1, F_2 \; e \; F_3$ são definidas por:

$$F_1 = 2\sum_{i=1}^{N}[\psi_i^T C_r^T Q_y C_r \Phi_i], \quad (2.2.27)$$

$$F_2 = -2\sum_{i=1}^{N}[\psi_i^T C_r^T Q_y C_r (\prod_i^{(n_r,N)})] \quad (2.2.28) \text{ e}$$

$$F_3 = 2\sum_{i=1}^{N}[\left(\prod_i^{(n_u,N)}\right)^T Q_u]. \quad (2.2.29)$$

Por fim, a matriz F é definida por:

$$F(k) = F_1 x(k) + F_2\tilde{y}_r^d(k) + F_3 u^d. \quad (2.2.30)$$

c) Restrições

O MPC é uma técnica de controle muito apreciada por permitir a inclusão de restrições ao sistema, fato já discutido anteriormente. Assim, para aplicações reais em geral, há variáveis (saídas) cujas restrições devem ser respeitadas, chamadas de saídas com restrições.

c.1) Restrições nas saídas

Em sistemas LTI, já discutidos, tais saídas são definidas através de uma matriz de restrições $C_c \in \mathbb{R}^{n_c \, x \, n}$.

Neste caso, o vetor de saídas é definido por:

$$y_c = C_c x + D_c u \; , \quad (2.2.31)$$

obedecendo a inequação:

$$y_c^{min} \leq y_c \leq y_c^{max}\ , \qquad (2.2.32)$$

sendo $y_c^{min} \in \mathbb{R}^{n_c}$ e $y_c^{max} \in \mathbb{R}^{n_c}$ os limites para os valores de y_c .

Com base nas restrições expressas acima e na equação (2.2.17), mediante manipulações matemáticas [38], chega-se a:

$$\underbrace{\begin{pmatrix} +Cc\psi_1 + D_c \prod_i^{(n_u,N)} \\ \vdots \\ +Cc\psi_N + D_c \prod_i^{(n_u,N)} \\ -Cc\psi_1 + D_c \prod_i^{(n_u,N)} \\ \vdots \\ -Cc\psi_N + D_c \prod_i^{(n_u,N)} \end{pmatrix}}_{A_{ineq}^{(1)}} \tilde{u} \leq \underbrace{\begin{pmatrix} -Cc\Phi_1 \\ \vdots \\ -Cc\Phi_N \\ +Cc\Phi_1 \\ \vdots \\ +Cc\Phi_N \end{pmatrix}}_{G_1^{(1)}} x(k) + \underbrace{\begin{pmatrix} +y_c^{max} \\ +y_c^{max} \\ \vdots \\ -y_c^{max} \\ -y_c^{max} \end{pmatrix}}_{G_3^{(1)}}. \qquad (2.2.33)$$

c.2) Restrições na variação do sinal de controle

Além das restrições impostas às saídas y_c, existem ainda as restrições por parte da taxa de variação do sinal de controle, devido a limitações dos atuadores. Assim, uma representação conveniente seria:

$$\delta^{min} \leq u(k+i) - u(k+i-1) \leq \delta^{\max}\ . \qquad (2.2.34)$$

A equação (2.2.34) estendida para o vetor contendo o conjunto de sinais de controle pode ser escrita como:

$$\begin{pmatrix} \delta_{min} \\ \delta_{min} \\ \vdots \\ \delta_{min} \end{pmatrix} \leq \begin{pmatrix} +\mathbb{I} & \mathbb{0} & \mathbb{0} & \dots & \mathbb{0} & \mathbb{0} \\ -\mathbb{I} & +\mathbb{I} & \mathbb{0} & \dots & \mathbb{0} & \mathbb{0} \\ & & & \vdots & & \\ \mathbb{0} & \mathbb{0} & \mathbb{0} & \dots & -\mathbb{I} & +\mathbb{I} \end{pmatrix} \tilde{u} + \begin{pmatrix} -\mathbb{I} \\ \mathbb{0} \\ \vdots \\ \mathbb{0} \end{pmatrix} u(k-1) \leq \begin{pmatrix} \delta_{max} \\ \delta_{max} \\ \vdots \\ \delta_{max} \end{pmatrix}. \qquad (2.2.35)$$

Através de algumas manipulações matemáticas, [38], chega-se a:

$$\underbrace{\begin{pmatrix} +\mathbb{I} & \mathbb{0} & \mathbb{0} \dots & \mathbb{0} & \mathbb{0} \\ -\mathbb{I} & +\mathbb{I} & \mathbb{0} \dots & \mathbb{0} & \mathbb{0} \\ & & \vdots & & \\ \mathbb{0} & \mathbb{0} & \mathbb{0} \dots & -\mathbb{I} & +\mathbb{I} \\ +\mathbb{I} & \mathbb{0} & \mathbb{0} \dots & \mathbb{0} & \mathbb{0} \\ -\mathbb{I} & +\mathbb{I} & \mathbb{0} \dots & \mathbb{0} & \mathbb{0} \\ & & \vdots & & \\ \mathbb{0} & \mathbb{0} & \mathbb{0} \dots & -\mathbb{I} & +\mathbb{I} \end{pmatrix}}_{A_{ineq}^{(2)}} \tilde{u} \leq \underbrace{\begin{pmatrix} -\mathbb{I} \\ \mathbb{0} \\ \vdots \\ \mathbb{0} \\ -\mathbb{I} \\ \mathbb{0} \\ \vdots \\ \mathbb{0} \end{pmatrix}}_{G_2^{(2)}} u(k-1) + \underbrace{\begin{pmatrix} +\delta_{min} \\ +\delta_{min} \\ \vdots \\ +\delta_{min} \\ -\delta_{min} \\ -\delta_{min} \\ \vdots \\ -\delta_{min} \end{pmatrix}}_{G_3^{(2)}} . \qquad (2.2.36)$$

c.3) Restrições sobre o sinal de controle

Além das restrições impostas sobre os atuadores, existe ainda a restrição inerente à saturação do sinal de controle, observada por:

$$\tilde{u}^{min} \leq \tilde{u} \leq \tilde{u}^{max} . \qquad (2.2.37)$$

Com base nos desenvolvimentos matemáticos mostrados até aqui, pode-se representar as restrições sobre a variável de decisão $\tilde{u}$ como:

$$A_{ineq}\tilde{u} \leq G_1 x(k) + G_2 u(k-1) + G_3, \qquad (2.2.38)$$

em que $\tilde{u}$ obedece aos limites de máximos e mínimos como representado pela equação (2.2.38) e:

$$A_{ineq} = \begin{pmatrix} A_{ineq}^{(1)} \\ A_{ineq}^{(2)} \end{pmatrix}; \quad G_1 = \begin{pmatrix} G_1^{(1)} \\ \mathbb{0}_{(2Nn_u)n} \end{pmatrix}, \qquad (2.2.39)$$

$$G_2 = \begin{pmatrix} \mathbb{0}_{(2Nn_c)n_u} \\ G_2^{(2)} \end{pmatrix}; \quad G_3 = \begin{pmatrix} G_3^{(1)} \\ G_3^{(2)} \end{pmatrix}. \qquad (2.2.40)$$

As matrizes $A_{ineq}, G_1, G_2, G_3, \tilde{u}^{min}$ e $\tilde{u}^{max}$ são constantes e podem ser calculadas no modo *off-line*.

Por fim, define-se B_{ineq} como:

$$B_{ineq} = G_1 x(k) + G_2 u(k-1) + G_3, \qquad (2.2.41)$$

mediante a restrição:

$$A_{ineq}\tilde{u} \leq B_{ineq}. \qquad (2.2.42)$$

Nota-se que B_{ineq} é atualizado pelos dados dos estados no instante atual k e sinais de controle no instante anterior $k-1$ conhecido e que, com base na equação (2.2.42), pode-se desenvolver uma rotina que atue como "otimizador" calculando a sequencia ótima de controle $\tilde{u}^{opt}$, assunto da próxima subseção.

d) O otimizador – Método Iterativo via Expansão de Gradiente Projetada

Como já mencionado, o problema de otimização (intrínseco ao MPC e analisado em questão) é rotulado como um problema QP (do inglês, *Quadratic Programming*), configurando-se como unimodal, por se tratar de um problema linear. Para se solucionar o mesmo faz-se necessário uma sub-rotina que seja capaz otimizar a função custo. Felizmente, muitas das bibliotecas numéricas trazem tal função, considerada de fácil solução [38]. Um exemplo clássico é a função *QUADPROG* presente no *software* MATLAB, que faz exatamente o descrito acima.

No entanto, para os fins deste trabalho é necessário que se tenha uma função editável para realizar tal otimização, visto que o algoritmo final MPC (que conterá tal função) será paralelizado e embarcado em um *hardware* (*Parallella*) dotado de limitações de espaço de memória e processamento, característica descritiva dos sistemas embarcados.

Assim, mediante ao este fato, optou-se por uma adaptação do método *Iterativo do Gradiente com Expansão*, descrito em [38], tendo em conta sua simplicidade.

Considerando-se uma função custo genérica $J(p)$ atuando sobre uma variável de decisão $p \in \mathbb{R}^{n_p}$ e sendo $G(p) = \frac{\partial J}{\partial p} p$ e $H(p) = \frac{\partial^2 J}{\partial p^2} p$, tem-se que:

$$p^{i+1} = p^i - \alpha G(p^i) \qquad (2.2.43)$$

em que α representa o valor do passo e p^{i+1} é melhor que p^i no sentido de que aquele corresponde a um menor valor da função custo que este. Assim:

$$J(p^{i+1}) < J(p^i) \qquad (2.2.44)$$

Essa é a ideia básica por traz do método de iteração por gradiente. A maior dificuldade está na escolha do valor adequado do passo $\alpha > 0$. Se este for muito pequeno a função convergirá lentamente ao mínimo local. Em contrapartida, se o valor de α for elevado pode levar a um valor da função custo em p^{i+1} maior que p^i, o que não é desejado em um problema de minimização.

Para que o não se tenha o comportamento descrito acima é necessário que se mantenha a iteração no "mesmo lado" da função custo, em direção ao mínimo global [38].

Para solucionar o problema em questão lança-se mão do *teorema do valor central* [38] garantindo que:

para todo $\alpha > 0$ existe um $\alpha_0 \leq \alpha$ em que $\frac{d\bar{J}}{d\alpha}(\alpha) = \frac{d\bar{J}}{d\alpha}(0) + \left[\frac{d^2\bar{J}}{d\alpha^2}(\alpha_0)\right]\alpha$.

A consequência desse teorema é que se conhecido um valor limite h_{max} que satisfaça:

$$\left\|\frac{\partial J}{\partial p^2}(p)\right\|_2 \leq h_{max} \qquad (2.2.45)$$

pode-se afirmar que:

$$\frac{d\bar{J}}{d\alpha}(\alpha) = \left\|G\left(p^i\right)\right\|_2^2[-1 + h_{max}\alpha] \qquad (2.2.46)$$

O que significa que a derivada de $\bar{J}(\alpha)$ não mudará de sinal enquanto se estiver um α que obedeça a relação:

$$\alpha \leq \frac{1}{h_{max}} \qquad (2.2.47)$$

Todavia, o desenvolvimento acima faz com que a convergência de um algoritmo (que faz uso do método do gradiente com passo α) tenha uma convergência bastante lenta após algumas iterações iniciais, tomando passos cada vez menores com a evolução do algoritmo[38].

Uma alternativa a tal problema seria a escolha de passos mais largos tais que:

$$p_\gamma^{i+1} \leftarrow p^i - \frac{\gamma}{h_{max}}G\left(p^i\right);\ \gamma > 1 \qquad (2.2.48)$$

Assim, através da comparação dos valores das funções custos $J\left(p_\gamma^{i+1}\right)$ e $J\left(p_1^{i+1}\right)$ e para $\gamma > 1$, pode-se ter as seguintes escolhas:

Se $J\left(p_\gamma^{i+1}\right) > J\left(p_1^{i+1}\right)$, indica que a expansão adotada foi além do esperado e γ é corrigido de modo que:

$$\gamma \leftarrow max\{\gamma_{min}, \beta^-\gamma\};\ 0 < \beta^- < 1;\ \gamma_{min} > 1 \qquad (2.2.49)$$

e o valor de p é dado por p_1^{i+1}.

Se $J\left(p_\gamma^{i+1}\right) < J\left(p_1^{i+1}\right)$, indica que a expansão pode ser adotada e o parâmetro γ pode ser incrementado de forma que:

$$\gamma \leftarrow \beta^+\gamma;\ \beta^+ > 1 \tag{2.2.50}$$

e o valor de p é dado por p_γ^{i+1}.

Do ponto de vista de restrições, uma maneira de se inseri-las é definir a função custo como:

$$J(p|\rho) \coloneqq J_0(p) + \sum_{i=1}^{n_g} \rho_i \left[\max\left(0, g_i(p)\right)\right]^2 \tag{2.2.51}$$

em que J_0 é a função custo original e ρ é o vetor de penalizações dado por:

$$\rho \coloneqq \begin{pmatrix} \rho_1 \\ \vdots \\ \rho_{n_g} \end{pmatrix} \tag{2.2.52}$$

A equação (2.2.51) indica que a violação das restrições induz um custo adicional excluindo tal violação do conjunto de soluções otimizadas pela função custo.

Utilizando um operador de projeção P_r: $\mathbb{R}^{n_p} \rightarrow [p^{min}, p^{max}]$ definido por:

$$\{p^r \coloneqq P_r(p)\} \Leftrightarrow \{\forall i\ p_i^r = \min(P_i^{max}, \max(p_i^{min}, p_i))\} \tag{2.2.53}$$

Mais precisamente, o método de projeção de gradiente se divide em dois passos:

1. Dado a iteração atual $p^{(i)} \in [p^{min}, p^{max}]$ e o valor atual do vetor de penalidades ρ, computa-se o valor de p_{cand}^{i+1} sem restrições e utilizando a iteração de gradiente, podendo-se utilizar o recurso de expansão já apresentado.

2. Obtém o valor de p^{i+1} por meio da projeção da solução sem restrição p_{cand}^{i+1} no domínio admissível $p^{i+1} \leftarrow \mathrm{P_r}(p_{cand}^{i+1})$.

O algoritmo até aqui descrito é conhecido como *PGE-Algorithm* e os detalhes de seu funcionamento podem ser vistos em [38].

2.2.3 Trabalhos Correlatos na Área de MPC Implementados em Sistemas Embarcados

Essa subseção traz como objetivo uma breve revisão dos trabalhos utilizando a técnica de controle preditivo baseado em modelo aplicada a sistemas embarcados. Sabe-se que essa técnica de controle ganhou aceitação principalmente em aplicações industriais, permitindo lidar com uma enorme variedade de problemas de controle, com múltiplas variáveis e sujeitos a restrições sobre os sinais de controle, estados internos e saídas, fato já mencionada no Capítulo 1.

Apesar de sua popularidade na indústria, a demanda computacional intensa imposta pelo MPC se mostrou um fator crítico para aplicações com exigência de um rápido tempo de resposta ou que devem ser executadas em plataformas embarcadas de recursos limitados, baixo custo e consumo energético [34].

Como resultado da afirmação acima, muitas pesquisas têm se concentrado na busca por algoritmos eficazes do ponto de vista de minimização do esforço computacional para problemas não lineares e lineares do tipo QP ([35], [42] e [58], sendo este último implementado em GPU's) e do consumo energético [43]. Tais fatores são importantes, visto que as restrições mais comuns aos sistemas embarcados dizem respeito à limitação de processamento, memória, e da necessidade de baixo consumo, devido à restrição física para suporte de baterias.

Há ainda trabalhos que se utilizam de recursos de otimização via algoritmos bio-inspirados, como por exemplo: (a) PSO (do inglês, *Particle Swarm Optimization*) aplicado ao MPC não linear, [44] e [45], (b) desenvolvendo comparações entre KPSO (do inglês, *Knowledge Particle Swarm Optimization*), RPSO (do inglês, *Random Particle Swarm Optimization*) e GA (do inglês, *Genetic Algorithm*) [46], (c) uso do algoritmo DE (do inglês, *Diferenttial Evolution*) para o MPC não linear aplicado ao controle de um braço robótico [47], (d) a união desse último com uma rede neural artificial [48], e (e) uma abordagem do MPC como um problema multi-objetivo embarcado em uma FPGA [49].

Assim, essa quantidade de pesquisas com foco em algoritmos eficientes que possam ser embarcados (como mostrado acima), juntamente com a evolução das plataformas de *hardware*, tem feito com que o MPC aplicado a sistemas embarcado se tornasse uma promissora área de pesquisa nos últimos anos. A maioria desses trabalhos utilizam-se de plataformas FPGA, [34] e [51] (implementação de arquitetura de ponto fixo), alcançando frequências na casa dos Mega-hertz.

As pesquisas envolvendo tais plataformas variam desde a proposição de *toolbox OpenSource* para desenvolvimento sobre FPGA's [52], até problemas com aplicações de tempo real [53], controle de aeronaves em uma abordagem do tipo QP baseado em SoC FPGA [54], implementações (em FPGA) de algoritmos baseados no método do gradiente com ponto fixo [55], problemas não lineares [56] e problemas unindo FPGAs e GPUs [57].

Muitos dos trabalhos supracitados acabam por explorar recursos de paralelismo. Em geral, essas pesquisas utilizam plataformas de *hardware* baseadas em GPUs, [58] e [59], havendo ainda a utilização de processadores mais sofisticados como o *Intel Xeon* E5410 [60].

Dentro do exposto nesta seção, percebe-se que há um grande interesse cientifico na viabilização e aperfeiçoamento das técnicas de controle preditivo aplicadas a sistemas embarcados. Apesar da quantidade de publicações existentes nos últimos anos, muitas delas estão focadas no desenvolvimento de algoritmos eficientes, como já citado, validando-os em uma abordagem de aceleração por *hardware* (principalmente FPGAs), mediante o mapeamento dos algoritmos diretamente em *hardwar*e. Tal abordagem tem gerado excelentes resultados do ponto de vista de desempenho computacional e energético, justificando sua ampla utilização.

No entanto, com o avanço das plataformas *multicore* e sob a promessa de rápido desenvolvimento via aceleração por *software* das aplicações (utilizando se dos recursos de paralelismo), novas pesquisas têm surgido com o intuito de se explorar essa abordagem alternativa para aplicações MPC embarcadas, sendo esse o caso do presente trabalho.

2.3 Conclusões do Capítulo

Os conceitos teóricos apresentados neste capítulo são importantes para o entendimento e desenvolvimento da presente pesquisa. Objetivando-se embarcar aplicações MPC lineares, faz-se fundamental o entendimento dos princípios que norteiam a técnica de controle preditivo, bem como da arquitetura sobre a qual o algoritmo será embarcado.

Uma vez analisados as características da plataforma *Parallella* utilizada, pode-se concluir que a mesma, devido a sua flexibilidade, permite uma implementação que varia desde o paralelismo em nível de bits (variando o tamanho da palavra escrita via DMA nos *eCores*, demonstrado em *benchmarks* no Capítulo 4) até um paralelismo em nível de *pipelining*, múltiplas unidades funcionais (FPU do ARM) e, evidentemente, o paralelismo em nível de processos e *threads*.

É importante salientar que os recursos da FPGA proporcionados pelo SoC Zynq não foram utilizados para mapeamento dos algoritmos do MPC, uma vez que a FPGA vem pré-configurada para viabilizar a comunicação entre o ARM e o MPSoC *Epiphany*.

Em se tratando de memória, pode-se dizer que a plataforma *Parallella* oferece um modelo de memória compartilhada não só entre o processador ARM e o coprocessador *Epiphany* como também entre os *eCores* deste ultimo (uma vez que cada *eCore* tem acesso a toda a memória dividida pelos nós da rede a que o mesmo faz parte). No entanto, o desenvolvedor pode estabelecer como regra de projeto que os *eCores* só acessaram suas respectivas áreas de memória, estabelecendo o conceito de memória local privada dos *eCores* e se aproximando de um modelo de memória distribuída.

Por fim, em termos de desempenho será fundamental levar em conta os pontos de serialização presentes no algoritmo MPC a ser paralelizado neste trabalho. Como será mostrado no Capítulo 3, tal algoritmo possui forte dependência de dados, o que, afeta o desempenho da solução paralela.

Toda a base teórica discutida até aqui servirá como referência para se traçar uma estratégia metodológica de caráter experimental, a fim de alcançar os objetivos almejados, dando embasamento para as escolhas a serem tomadas e fornecendo artifícios para mensurar os resultados alcançados.

Finalmente, pode ser observado uma carência de trabalhos usando a plataforma *Parallella* em aplicações do tipo MPC, sendo esta uma parte da justificativa deste trabalho.

Capítulo 3

Avaliação das Características da Arquitetura *Epiphany* via *Benchmarks*

O objetivo deste capítulo é trazer uma série de testes que explorem as características da plataforma de *hardware* utilizada, dando embasamento para propor uma estratégia de paralelização que seja eficiente do ponto de vista de tempo computacional. Em outras palavras, estão descritos a seguir os passos efetuados para se cumprir as etapas 1 e 2 da metodologia apresentada no Capítulo 1.

Como forma de organização, o capítulo foi dividido em subseções que trazem primeiramente uma revisão dos dados de desempenho da plataforma *Parallella* do ponto de vista teórico, mediante documentação do fabricante, e segundo os *benchmarks* realizados por trabalhos correlatos. Em seguida são apresentados os *benchmarks* propostos pelo presente trabalho, focando-se nas características da plataforma que sejam de interesse das aplicações propostas, assim como os resultados obtidos. Este último passo fornece informações sobre as quais serão tomadas as decisões acerca da concepção de uma estratégia de paralelização que melhor atenda aos propósitos desta pesquisa.

3.1 Cenário de Desenvolvimento

Todo o trabalho foi concebido sobre a arquitetura da placa *Parallella* "*Desktop*" composta por (a) um SoC *Xilinx Zynq Dual-core* ARM A9/FPGA XC7Z010 e (b) um coprocessador *Epiphany* 16 core CPU E16G301 (o qual se caracteriza como um sistema do tipo MPSOC).

O sistema operacional embarcado no processador ARM A9 da placa *Parallella* é uma distribuição *Linux* fornecida pelo próprio fabricante, sendo que neste trabalho utilizou-se a versão do *Ubuntu* 15.04. Tal imagem já se encontra dotada do compilador GCC e eSDK 2016.03, contando com ferramentas de desenvolvimento que incluem principalmente um compilador de linguagem C otimizado e um simulador e *debbuger*.

A forma de *boot* adotada para o desenvolvimento foi via cartão *micro-SD* Desse modo, preparou-se tal mídia realizando os processos de formatação e de gravação da imagem no mesmo, fazendo com que o *boot* da placa fosse efetuado diretamente do cartão.

Apesar da possibilidade de acesso gráfico da placa via HDMI (do inglês, *High-Definition Multimidia Interface*) ou mesmo de algum programa VNC (do inglês, *Virtual Network Computing*), o presente trabalho foi concebido em sua totalidade acessando o *hardware* via terminal e protocolo SSH (do inglês, *Secure Shell*). A Figura 3.1.1 ilustra a configuração física final de desenvolvimento.

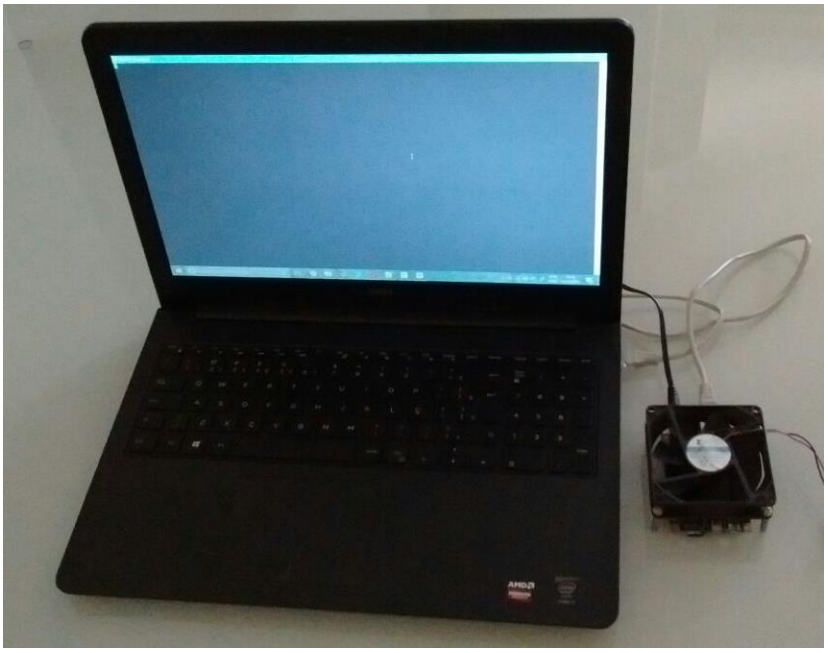

Figura 3.1. 1 Cenário final de desenvolvimento da presente pesquisa.

Para desenvolvimento dos programas em linguagem *C/C++* foi utilizado o *software Code::Blocks* como IDE (do inglês, *Integrated Development Environment*) instalado no próprio sistema operacional do ARM. A visualização gráfica foi viabilizada pela implementação do *X Window System*.

Os projetos dos *benchmarks* e aplicações criadas no *Code::Blocks* foram configurados de modo a ser possível a compilação de códigos tendo como alvo (*target*) ambas as arquiteturas envolvidas, a saber, ARM e *Epiphany*. Assim, criou-se um *target* virtual que englobasse um compilador para a arquitetura ARM e outro para a arquitetura *Epiphany*, sendo este último configurado com a *toolchain* relativa à mesma.

Mediante ao exposto acima, percebe-se que a estrutura do ambiente de desenvolvimento se configura com o ARM fazendo o papel de *host* e realizando a compilação cruzada dos códigos destinados aos *eCores Epiphany*, que neste cenário se enquadra como *device*.

3.2 Estrutura da Aplicação Paralela

Do ponto de vista de desenvolvimento e para que se tenha melhor proveito em termos de desempenho das aplicações a serem embarcadas, devem-se levar em conta alguns fatores

fundamentais da arquitetura, que influenciaram na tomada de decisões e estratégias de desenvolvimento das aplicações.

Para que se tenha uma aplicação embarcada na placa *Parallella* faz-se necessário no mínimo dois programas distintos: (a) um para o lado do SoC *Zynq* (ARM) que irá funcionar como *host* e (b) outro para ser executado nos *eCores* do coprocessador *Epiphany*, funcionando como *device*.

A concepção de tais programas deve levar em conta o *overhead* de comunicação existente entre o ARM e os *eCores*, tomando estratégias que minimizem esse custo, que está diretamente ligado ao fluxo de dados trocados entre as partes e consequentemente a natureza da aplicação.

Em termos funcionais, o programa a ser executado do lado do *host* será o responsável por: (a) definição do *workgroup* (conjunto de *eCores* ativos), (b) área de memória a ser compartilhada com o coprocessador *Epiphany* e (c) chamada da função que irá carregar na memória interna dos *eCores*, o programa a ser executado no mesmo.

Neste cenário, pode-se carregar todo o *workgroup* de uma só vez com o mesmo programa ou ainda, dependendo da estratégia e da implementação realizada, carregar cada *eCore* em separado, com códigos distintos entre si.

Junto com a chamada de função que grava os códigos a serem executados nos *eCores* é passado um parâmetro que autoriza o início imediato do processamento dos mesmos ou os força a esperar por um novo comando especifico enviado pelo *host*.

Após a realização das etapas supracitadas, faz-se necessário a utilização de um mecanismo de sincronização tanto entre os *eCores* quanto entre o *host* e o *device*, típicos de arquiteturas paralelas. Tais mecanismos pode ser implementado de inúmeras formas, sendo que para o presente trabalho utilizou-se "*flags*" sinalizadoras em campos fixos da memória compartilhada.

A arquitetura *Epiphany* possui flexibilidade quanto ao modo de programação sendo compatível com os métodos de programação paralela mais populares, incluindo SIMD (do inglês, *Single Instruction Multiple Data*), SPMD (do inglês, *Single Program Multiple Data*), MIMD (do inglês, *Multiple Instruction Multiple Data*), programação mestre escravo, comunicação por memória compartilhada *mult-thread*, *message-passing* e CSP (do inglês, *Communicating Sequential Processes*), dentre outros [62].

A Adapteva provê a IDE de desenvolvimento em *C/C++ Eclipse*, todavia neste trabalho foi utilizado o *Code::Blocks* como IDE de desenvolvimento. Essa escolha foi tomada devido ao *feedback* relativo a dificuldades na utilização do *Eclipse (*reportados no fórum da plataforma), além de uma nota no presente no manual da *eSDK* [2], alertando o não funcionamento de algumas das funcionalidades desta IDE, tais como o recurso de *debugging* . Há também trabalhos de suporte a *OpenCL* oferecidos pela *Brown Deer* , no entanto, esse último se utiliza

da compilação da SDK *Adapteva* em *background* pouco acrescentando em benefícios de desempenho[65].

Em termos da arquitetura do *hardware* propriamente dita, do lado do coprocessador *Epiphany* a maior limitação existente se dá pelo pequeno espaço de memória RAM local por *eCore* (32KBytes divididos em quatro bancos de 8KBytes cada) que devem ser divididos entre o código do programa (definido como o primeiro banco 0 – 8KB) e os dados.

Em se tratando da precisão, operações em ponto flutuantes de precisão simples se mostram suficientes para muitas aplicações, como pode ser visto em [67] e [68]. Além disso, o *SoC Zynq* provê uma unidade de ponto flutuante de alto desempenho VFPU (do inglês, *Vector Floating Point Units*). O desempenho máximo para operações com ponto flutuante é alcançado por meio de instruções FMADD (do inglês, *Fused-Multiply-Add*) com 64 bits de leitura ou escrita de operações a cada ciclo.

Por fim, dentre os fatores relativos à concepção das aplicações deve-se levar em conta as opções de compilação existentes que ajudam na otimização do código, conferindo-lhe melhor desempenho, tanto para os programas do lado do ARM quanto do lado do *Epiphany*.

3.3 Revisão de Desempenho da Plataforma Parallella

Nesta seção são analisados os desempenhos de algumas das características da plataforma *Parallella*, em especial da arquitetura *Epiphany*, que estão diretamente relacionados com as aplicações MPC de interesse. Estes dados de desempenho foram reportados na literatura, como mostrado nesta seção.

Sabe-se que o desempenho de uma aplicação está ligado a fatores como seu particionamento, escalonamento, requisitos de memória, dependência de dados e eficiência de compilação [79]. É comum encontrar aplicações que não aproveitam de todo o poder computacional teórico esperado da arquitetura de *hardware*. Em [69], por exemplo, a transformada rápida de Fourier em duas dimensões 2D-FFT (do inglês, *Fast Fourier Transform*) atingiu um valor máximo de apenas 13% do pico de desempenho teórico.

3.3.1 Dados Teóricos e *Benchmarks* da Literatura Relacionada

Nesta subseção é apresentada uma breve análise do desempenho relativo ao tempo de *overhead* proveniente da comunicação entre os elementos que compõem a arquitetura da placa *Parallella*. Em se tratando de arquiteturas *multicore*, é fundamental que se tenha um bom conhecimento das taxas de comunicação entre os núcleos de processamento e dos mesmos com as memórias do sistema. Via de regra, esse é um dos maiores gargalos do sistema como um todo e, por consequência, um fator preponderante que pode determinar a viabilidade ou não de uma dada aplicação sobre a arquitetura analisada.

a) Dados de Desempenho Teóricos

Os dados apresentados na Tabela 3.3.1 são referentes à plataforma *Parallella* E16G301e apresentados em [65]. Pode ser observado que o SoC Zynq trabalha a uma frequência de *667 MHz* enquanto que o MPSoC *Epiphany* (E16G301) trabalha a frequência de 600 MHz.

Tabela 3.3. 1 Dados Teóricos da placa Parallella ([65] Adaptado).

Especificação	Valor
Frequência Zynq	667 MHz
Frequência E16G301	600 MHz
Pico de desempenho E16G301 com ponto flutuante	19.2 GFLOPS
Pico da banda Zynq-E16G301	600MB/s up, 600MB/s down
Memória DDR3	1GiB

***b) Benchmarks* de Trabalhos Correlatos**

Dentro da literatura encontram-se algumas pesquisas se propõe a levantarem dados de desempenho em termos de custo computacional e gasto energético da arquitetura *Epiphany*.

Dentro dos interesses do presente trabalho, fatores como o custo de comunicação entre os *eCores* e de acesso a memória compartilhada são de extrema importância, devido à natureza da aplicação MPC utilizada. Em [18] fora realizado um *micro benchmark* em que se explorou a comunicação interna entre os *eCores* de dois modos diferentes: (a) através do uso de DMA e (b) da comunicação ponto a ponto. O gráfico da Figura 3.3.1 ilustra os resultados obtidos para ambos os métodos comparando-os em termos do tamanho da mensagem transmitida e da banda alcançada.

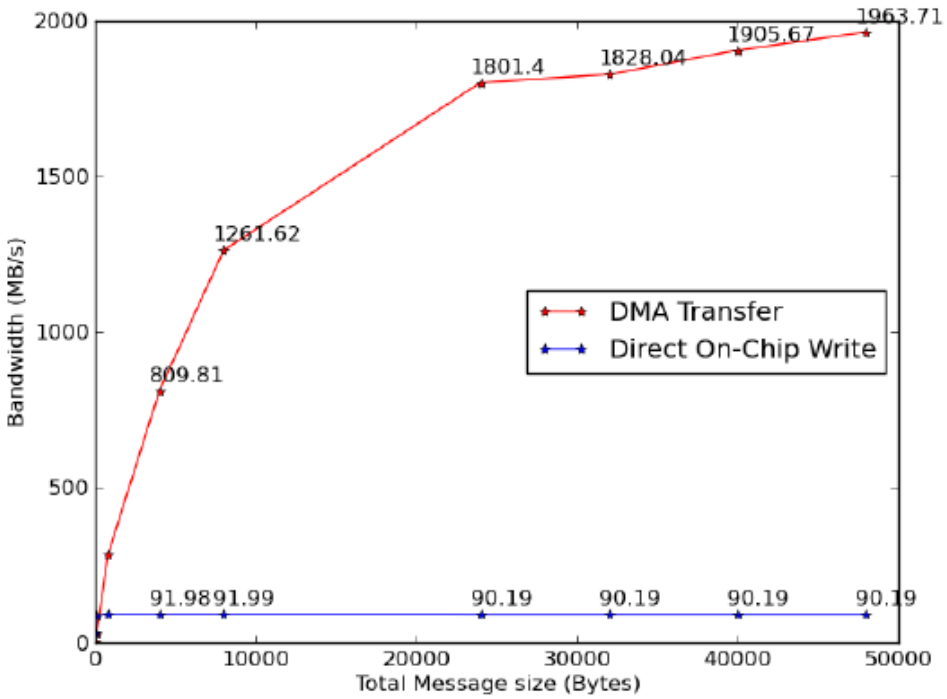

Figura 3.3. 1 Comparação entre comunicação dos eCores via DMA e escrita direta [18].

A conclusão obtida por [18], mediante a análise dos dados contidos no gráfico da Figura 3.3.1 e à análise de comunicação, foi que para dados de até 500 bytes é vantajoso utilizar a escrita direta na memória de um *eCore* adjacente frente ao uso de DMA.

Em [70], realizou-se um *benchmark* a fim de se medir não só o custo da comunicação entre os *eCores*, como também entre o os mesmos e a memória compartilhada DRAM, e também com o ARM. Os resultados são ilustrados na Tabela 3.3.2.

Tabela 3.3. 2 Custo de comunicação entre os elementos da placa Parallella (Adaptado [70]).

Origem	Destino	Operação	Largura de Banda(MB/s)
ARM Host	eCore(0,0)	Leitura	7,4
ARM	eCore(0,0)	Escrita	50.38
ARM	DRAM	Cópia	437.97
ARM	DRAM	Leitura	127.73
ARM	DRAM	Escrita	85.01
eCore(0,0)	DRAM	Leitura (DMA)	161.10
eCore(0,0)	DRAM	Escrita(DMA)	238.54
eCore(0,0)	eCore(0,1)	Leitura(DMA)	333.72
eCore(0,0)	eCore(0,1)	Escrita(DMA)	1271.7

Pela Tabela 3.3.2 observa-se que a leitura e escrita à memória compartilhada tem um custo significativo em termos de tempo de processamento, fato que influencia muito na

escolha de estratégia de paralelização para aplicações com grande volume de dados e forte interdependência entre os mesmos, características da aplicação MPC proposta. A operação entre o processador ARM e a memória compartilhada DRAM descrita como "Cópia" consiste na cópia de dados entre dois *buffers* estaticamente alocados na memória do ARM (via função *memcpy*).

Existem outros trabalhos que também contribuíram com a análise de desempenho da arquitetura Epiphany, como em [71] e [72], em que tal arquitetura é comparada com um processador Intel-I7 e [69] e [73], que exploram o uso MPI (do inglês, *Message Passing Interface*).

3.4 *Benchmarks* Desenvolvidos neste Trabalho para Avaliação de Desempenho

Com o objetivo de explorar características da placa *Parallella* que são de interesse direto da aplicação MPC, foram desenvolvidos programas cujos intuitos são extrair do *hardware* a resposta relativa ao tempo de computação do mesmo mediante uma ação ou operação específica, uma vez que o que se deseja otimizar nas aplicações MPC é o tempo de execução, fator crítico e de impedimento para que tais aplicações sejam executadas em sistemas embarcados. Muitos dos testes realizados aqui, como a multiplicação de matrizes e o acesso a memória compartilhada, são variações de testes realizados por trabalhos correlatos.

3.4.1 Multiplicação de Matrizes

A estratégia de controle MPC (cujo algoritmo será embarcado neste trabalho) faz o uso do modelo de controle em espaço de estados. Assim, a maior parte dos cálculos necessários para se obter os sinais de controle estão relacionados a operações com matrizes, mais especificamente à *multiplicação de matrizes*.

Tendo tais operações como sendo uma das principais características de interesse a se explorar na plataforma de *hardware Parallella*, criou-se um *benchmark* para que se pudessem verificar os seguintes quesitos:

- Eficiência da multiplicação de matrizes com paralelização via coprocessador *Epiphany*, quando comparado ao mesmo cálculo realizado de forma serial no processador ARM.
- Diferença de desempenho ao se variar o tamanho dos dados utilizados nas operações de leitura e escrita via DMA dos *eCores*.

Em [18] fora feito um experimento semelhante em que se obtiveram dados do desempenho da arquitetura *Epiphany* dado à multiplicação de matrizes de variados tamanhos. Todavia, em tais experimentos eram variados não só o tamanho das matrizes como também dos *workgroups*

destinados às operações. Nesse sentido, o *benchmark* desenvolvido nesta subseção se difere do realizado em [18], realizando a comparação entre multiplicações de matrizes (de tamanhos variáveis) de forma serial (realizada no ARM) com a multiplicação paralelizada (utilizando todos os 16 *eCores Epiphany*).

Sabe-se, como será mostrado posteriormente neste trabalho, que existe um custo significante associado ao ato de carregar os programas executáveis nos *eCores*. Dessa maneira, e baseando-se nas características da aplicação MPC, o mais interessante é que se tenha um maior número de *eCores* sendo executados simultaneamente, e que os programas a serem executados nos mesmos sejam carregados o menor número de vezes possível.

De posse de tais informações, no presente *benchmark* será explorada a multiplicação de matrizes utilizando-se os 16 *eCores* (simultaneamente) para todos os tamanhos de matrizes experimentados.

A estratégia de paralelização utilizada está baseada na técnica descrita em [63], em que as matrizes a serem multiplicadas são particionadas e distribuídas entre os *eCores* que por sua vez, ao término das operações sobre os dados que lhes cabem, realizam a rotação vertical e horizontal dos dados computados, até que se tenha processado todo o cálculo da multiplicação da matriz completa.

Um último fato a ser observado acerca do presente *benchmark* diz respeito ao uso de DMA por parte dos *eCores* para transferência de dados da memória compartilhada para as memórias locais dos mesmos, e vice-versa. Tal transferência pode ser feita a três tamanhos diferentes de dados por leitura/escrita, a saber, 16, 32 e 64 bits por operação.

A escolha pelo tamanho dos pacotes de dados utilizados nas operações DMA é configurada na estrutura de descrição da mesma, utilizando-se as macros *E_DMA_HWORD*, *E_DMA_WORD* e *E_DMA_DWORD* para 16, 32 e 64 bits respectivamente. Como nas aplicações MPC são utilizados dados do tipo *float*, formados por 4 bytes cada, optou-se por realizar comparações relativas ao uso de 32(um *float*) e 64(dois *floats*) bits na transferência de dados via DMA, comparando o desempenho entre ambos.

a) Multiplicação de Matrizes utilizando E_DMA_DWORD (64 bits)

A Tabela 3.4.1 e o gráfico da Figura 3.4.1 ilustram os resultados obtidos mediante aos testes de multiplicação de matrizes com tamanho variável. Para efeito de análise, são comparados os tempos gasto por uma multiplicação, sem otimizações, realizada de maneira serial no processador ARM A9 com a multiplicação, também sem otimização que não a própria paralelização, realizada pelos *eCores Epiphany*. Lembrando que este último tempo inclui os custos de comunicação entre os chips *Zynq* e *Epiphany*, bem como de sincronização entre os mesmos e entre os *eCores*.

Tabela 3.4. 1 Tabela da comparação do tempo gasto na multiplicação de matrizes quadradas no processador ARM e no coprocessador Epiphany utilizando transferência via DMA de 64 bits.

Transferência de 64 bits	Tempos (ms)	
Tamanho das Matrizes	ARM A9 Dual Core	Epiphany
8x8	0,019	0,129
16x16	0,092	0,165
32x32	0,774	0,332
64x64	5,991	1,125
128x128	78,078	5,401
256x256	243,6	22,9
512x512	1748,7	136,7

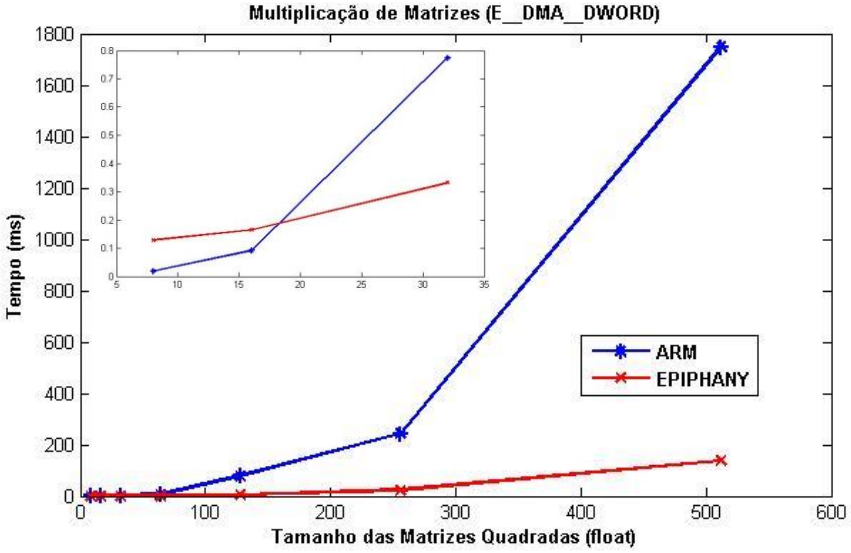

Figura 3.4. 1 Gráfico da comparação do tempo gasto na multiplicação de matrizes quadradas no processador ARM e no coprocessador Epiphany, utilizando transferência via DMA de 64 bits.

Pelos dados expressos pela Tabela 3.4.1 e pelo gráfico da Figura 3.4.1, percebe-se que para tamanhos de matrizes acima de 20x20 (400 elementos) o custo de preparação dos dados para escrita em memória compartilhada e acionamento do *device* (*Epiphany*), por parte do *host* (*Zynq*), bem como o custo de sincronização entre os próprios *eCores* são compensados pela aceleração dos cálculos realizados de forma paralela na arquitetura *Epiphany*. Portanto, quanto maior a demanda de dados, maior a justificativa do uso da arquitetura *Epiphany* para multiplicação de matrizes (tal como é de se esperar). Embora este tenha sido uma variação do *benchmark* realizado em [18], ambos os resultados são concordantes.

b) Multiplicação de Matrizes Utilizando E_DMA_WORD (32 bits)

A exemplo dos testes anteriormente apresentados, realizou-se os mesmos procedimentos, todavia, com 32 bits de dados por operação de leitura/escrita via DMA. Os resultados desta nova configuração são expressos pela Tabela 3.4.2 e pelo gráfico da Figura 3.4.2.

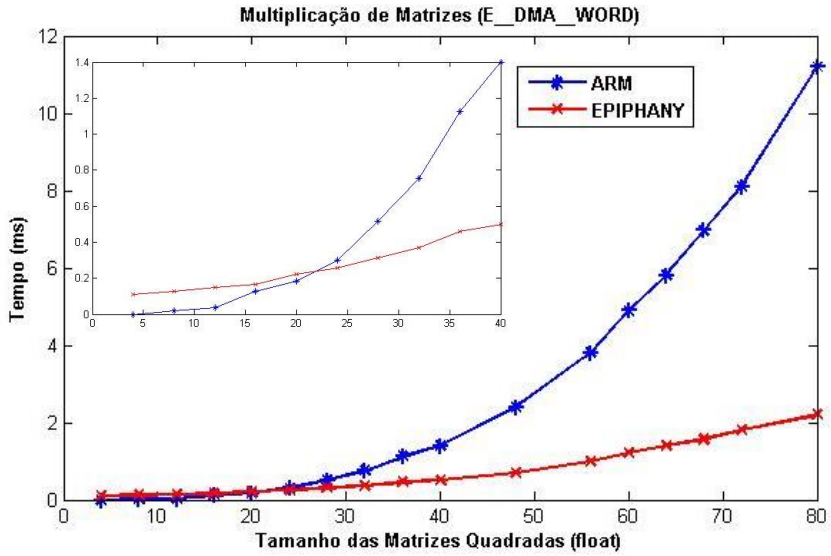

Figura 3.4. 2 Gráfico da comparação do tempo gasto na multiplicação de matrizes quadradas no processador ARM e no coprocessador Epiphany, utilizando transferência via DMA de 32 bits.

Tabela 3.4. 2 Tabela da comparação do tempo gasto na multiplicação de matrizes quadradas no processador ARM e no coprocessador Epiphany utilizando transferência via DMA de 32 bits.

Transferência de 32 bits	Tempos (ms)	
Tamanho das Matrizes	ARM A9 Dual Core	Epiphany
4x4	0	0,111
8x8	0,018	0,129
12x12	0,037	0,147
16x16	0,129	0,166
20x20	0,184	0,222
24x24	0,3	0,257
28x28	0,516	0,313
32x32	0,756	0,368
36x36	1,125	0,461
40x40	1,4	0,5
48x48	2,4	0,7
56x56	3,8	1
60x60	4,89	1,217
64x64	5,806	1,401
68x68	6,968	1,567
72x72	8,1	1,8
80x80	11,2	2,2

Os resultados observados na Tabela 3.4.2 e no gráfico da Figura 3.4.2 retratam a mesma conclusão que se chegara para o caso de anterior, quanto maior o tamanho da matriz mais justificado é o uso da arquitetura *Epiphany* para realização dos cálculos, sendo que tal ganho de desempenho começa a aparecer para matrizes de 24x24 (400 elementos).

c) Comparação Entre o Uso de *E_DMA_DWORD e E_DMA_WORD*

Tendo em conta os resultados experimentais obtidos, por parte da arquitetura *Epiphany* sabe-se que o melhor ganho de desempenho será obtido quando utilizado um maior volume de dados nas transferências via DMA. Tal fato é apresentado na Tabela 3.4.3 e no gráfico ilustrado na Figura 3.4.3.

Tabela 3.4. 3 Tabela de comparação entre transferência de dados via DMA utilizam 64 (E_DMA_DWORD) e 32 (E_DMA_WORD) bits.

Transferência de 64 bits	Tempos (ms)	
Tamanho das Matrizes	ARM A9 Dual Core	Epiphany
8x8	0,019	0,129
16x16	0,092	0,165
32x32	0,774	0,332
64x64	5,991	1,125
128x128	78,078	5,401
256x256	243,6	22,9
512x512	1748,7	136,7

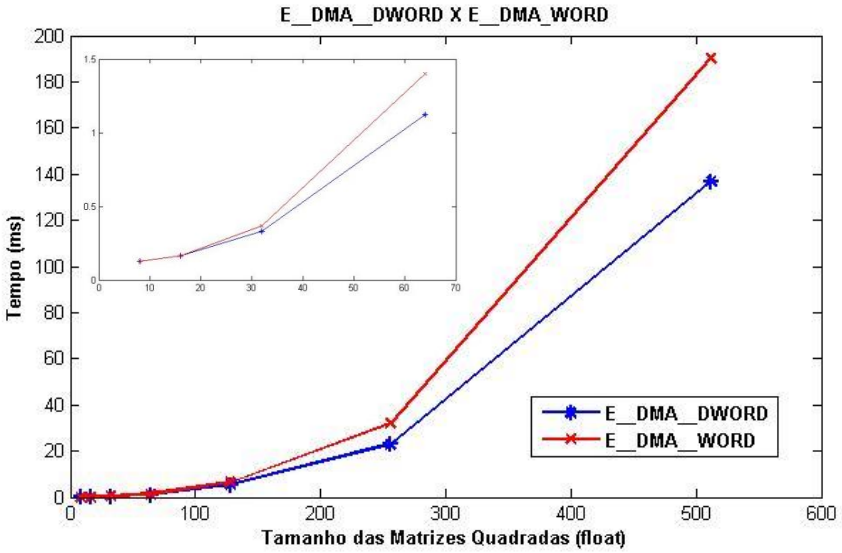

Figura 3.4. 3 Gráfico de comparação entre transferência de dados via DMA utilizando 64 (E_DMA_DWORD) e 32 (E_DMA_WORD) bits.

Mediante aos dados apresentados pela Tabela 3.4.3 e pelo gráfico da Figura 3.4.3, percebe-se que de fato é mais vantajoso (em termos de desempenho e visando tempo de computação), a utilização de 64 bits por transferência DMA. Todavia, para que isso seja possível pode-se fazer necessária uma adaptação das matrizes a serem multiplicadas dependendo do seu tamanho.

Além disso, a melhora só começa a ser significativa para matrizes de tamanhos elevados, acima de 200x200, indicando que tal escolha, entre utilizar *E_DMA_WORD* ou *E_DMA*

DWORD, deve ser tomada segundo a característica de cada aplicação. Este conhecimento experimental é importante para uma aplicação do tipo MPC, onde o tamanho das matrizes a serem calculadas crescem com respeito a características do problema como número de variáveis de estado e horizonte de predição.

3.4.2 Custo de Acesso à Memória Compartilhada

A utilização da arquitetura presente na placa *Parallella* conta com um gargalo em termos de tempo de processamento, inserido pelo acesso a memória compartilhada por parte tanto do Zynq quanto pelo coprocessador *Epiphany*. Desse modo, apesar de se tratar da forma de comunicação entre tais chips para quantidades de dados expressivas, é recomendado que tal recurso seja minimizado tanto quanto possível.

Com o intuito de se avaliar tal característica (de maneira experimental) e em complemento ao *benchmark* realizado em [70], em que foram medidas as larguras de banda (*bandwidth*) existente na comunicação entre os componentes da placa *Parallella*, mediu-se o tempo de acesso do SoC *Zynq* para escrita e leitura na memória compartilhada com tamanhos diferentes de dados. Tais tamanhos foram escolhidos baseando-se nas matrizes presentes na aplicação MPC, e os resultados obtidos podem ser observados na Tabela 3.4.4 e no gráfico da Figura 3.4.4.

Tabela 3.4. 4 Tabela de tempo de acesso à memória compartilhada para operações de leitura e escrita via SoC Zynq.

Acesso à Memória	Tempos (ms)	
Tamanho das Matrizes	leitura	Escrita
8x8	0,006	0,005
16x16	0,01	0,013
32x32	0,032	0,045
64x64	0,104	0,184
128x128	0,438	0,695
256x256	1,659	2,775
512x512	7,552	11,058

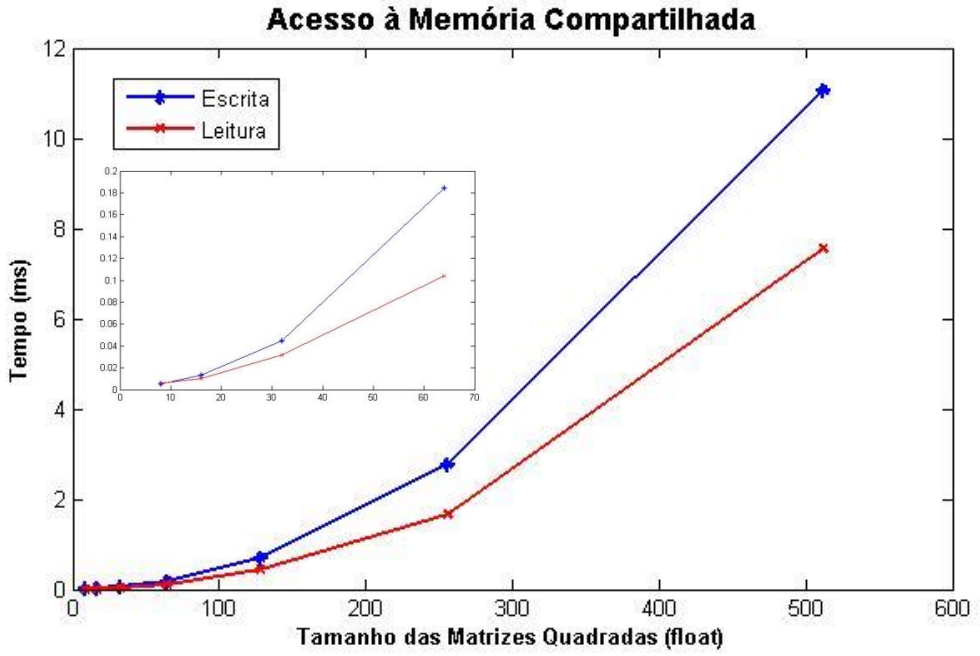

Figura 3.4. 4 Gráfico de tempo de acesso à memória compartilhada para operações de leitura e escrita via SoC Zynq.

Os resultados expressos pela Tabela 3.4.4 e pelo gráfico da Figura 3.4.4 mostram que o custo de acesso à memória cresce de forma exponencial com o aumento do volume de dados a serem escritos/lidos, tendo a escrita um desempenho pior que a leitura. Esse não é um resultado usual, visto que no geral se tem leituras mais rápidas que escrita. No entanto, está em concordância com o apresentado em [70], em que a largura de banda para a leitura é maior que para a escrita.

Tais operações foram realizadas mediante funções específicas (e_read/e_write) da *eSDK* Adapteva, sendo que a documentação técnica da mesma, assim como [70], não abordam o comportamento em questão.

Apesar da ordem de grandeza de tais operações estarem em milissegundos (para leituras e escritas recorrentes), dentro de um *loop* de programa, por exemplo, como é o caso das aplicações MPC, tais custos podem significar a aplicabilidade de se embarcar um algoritmo ou não.

Vale destacar que todos os experimentos apresentados acima têm como propósito específico analisar características da plataforma *Parallella* que são de interesse para o algoritmo MPC a ser paralelizado e embarcado, complementando os *benchmarks* de trabalhos correlatos e propondo uma abordagem experimental do problema. Ressalta-se ainda que os resultados obtidos neste capítulo estão em concordância com os testes realizados pela literatura relacionada.

3.4.3 Custo de se Carregar um Programa Executável na Memória do eCore

Como visto anteriormente neste documento, os *eCores* possuem uma memória local limitada contando com apenas quatro bancos de 8KB que devem ser divididos entre programas e dados.

Assim, evidente que existe um compromisso entre tamanho do programa suportados e tamanho de dados guardados em memória local que deve ser avaliado pelo programador segundo as especificidades de sua aplicação. Isso porque, como já observado, o gargalo no acesso a memória compartilhada deve ser evitado. Desse modo, fazer o chamado *caching* de memória [79], copiando a maior quantidade de dados da memória compartilhada para a memória local, levará a um ganho de desempenho considerável, principalmente para aplicações com grande demanda de dados.

Por outro lado, dependendo da complexidade das operações realizadas e da generalização sobre a qual os programas são desenvolvidos, podem ser necessárias uma grande quantidade de linhas de código, aumentando seu tamanho e levando-o a ocupar um espaço maior na memória local dos *eCores*.

Diante do *trade-off* apresentado acima, uma possível alternativa seria desenvolver programas curtos e otimizados, tanto em termos de tamanho quanto em termos de custo computacional, para executar uma operação específica, mudando o programa na memória dos *eCores* sempre que necessário. Isso implicaria em uma menor utilização da memória local dos *cores* culminando em mais espaço para os dados.

Poder-se-ia ainda trabalhar com diferentes programas rodando em diferentes *cores*, dando maior flexibilidade ao uso do *Epiphany*. No entanto, na prática existe um custo associado ao ato de se carregar um programa, executável na memória dos *eCores*, como mostrado pela Tabela 3.4.5 e pelo gráfico da Figura 3.4.5.

Tabela 3.4. 5 Tabela do custo de inicialização dos programas executáveis nos eCores do coprocessador Epiphany.

Inicialização dos eCores	
Tamanho do Workgroup	Tempo(ms)
1	1,861
2	3,226
3	4,35
4	5,548
6	8,221
8	10,728
9	11,999
12	15,87
16	20,737

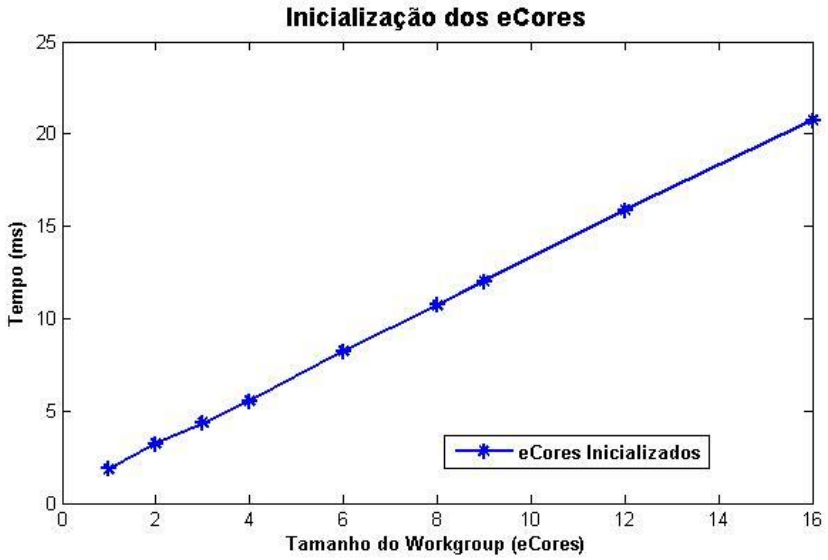

Figura 3.4. 5 Gráfico do custo de inicialização dos programas executáveis nos eCores do coprocessador Epiphany.

Pela Tabela 3.4.5 e pelo gráfico da Figura 3.4.5 apresentados, percebe-se que, a exemplo do acesso a memória, o custo de se carregar programas executáveis aos *eCores* pode ser um fator determinante para a decisão de aplicabilidade de um algoritmo, isso porque se tais operações ocorrerem de maneira demasiada, dentro de *loops* de código, por exemplo, os custos de se carregar os programas nos *eCores* poderiam ser altos o suficiente para neutralizar os ganhos obtidos com o paralelismo das operações que se esteja efetuando.

3.4.4 Análise de Desempenho da Biblioteca *Eigen* para Operação Matricial (avaliação de desempenho dentro do ARM)

As operações com matrizes e vetores são recorrentes em computação, fato que explica a existência de diversas bibliotecas e *softwares* especializados em tais operações. A *Eigen* é uma dentre estas muitas bibliotecas, tendo como característica o fato de ser escrita em linguagem C++ e lidar com *template* de operações de álgebra linear, transformações geométricas, *solvers* numéricos e algoritmos relacionados, além das já citadas operações com matrizes e vetores.

Adicionalmente ao apresentado acima, a biblioteca *Eigen* é *open source* e implementada utilizando-se da técnica conhecida como *expression templates metaprogramming*, o que significa que os *templates* são utilizados pelo compilador para gerar códigos temporários que são incorporados, também pelo compilador, ao restante do código fonte e enfim compilados. Por fim, esta biblioteca ainda possui seu próprio método de desdobramento de *loops* (“*loop unrolling*”) e vetorização para conjunto de instruções SSE 2/3/4, ARM NEON (32-bit and 64-bit) e PowerPC AltiVec/VSX (32-bit and 64-bit) [74].

Do ponto de vista de velocidade das operações, a *Eigen* permite o chamado *lazy evaluation*, removendo *buffers* intermediários para resultados temporários, sempre que tal operação for apropriada. Matrizes de tamanho fixo são otimizadas assim: (a) alocação dinâmica é evitada e

(b) os *loops* são “desdobrados”. Para matrizes de grandes dimensões é tomada uma atenção especial com *cache-friendliness*.

Mediante os fatores supracitados e o fato da biblioteca *Eigen* ser de fácil utilização, apresentando expressões intuitivas e semelhantes a pseudocódigos, a faz atrativa no desenvolvimento de programas com operações características como a aplicação MPC do presente trabalho. Nesse sentido, com o intuito de se testar seu desempenho, elaborou-se um *benchmark* comparando (experimentalmente) os tempos gastos na multiplicação de matrizes de diferentes tamanhos realizadas com e sem o uso desta biblioteca. Para tanto, foram utilizadas matrizes de tamanho fixo e os resultados obtidos podem ser verificados na Tabela 3.4.6 e no gráfico da Figura 3.4.6.

Tabela 3.4. 6 Tabela de comparação entre os tempos de multiplicação de matrizes quadradas via biblioteca Eigen e método convencional.

Eigen x Convencional	Tempos (ms)	
Tamanho das Matrizes	Eigen	Convencional
20x20	0,1	0,1
70x70	3,7	2,9
120x120	21,6	19,4
190x190	68	107,7
240x240	127,5	189,4
310x310	297,6	480,5
380x380	492,7	1043
450x450	852,5	1969,7
520x520	1288	3160,6
590x590	1864,5	5253,7
660x660	2789,4	7599,2

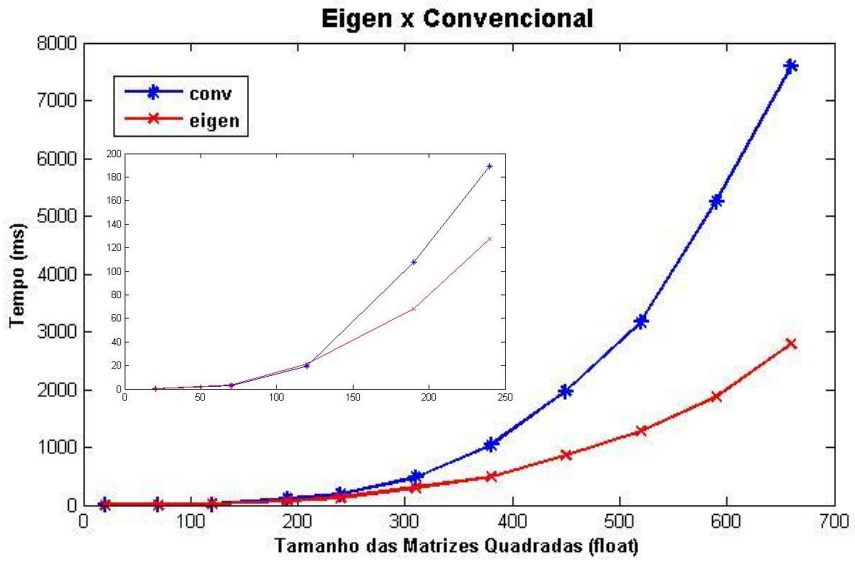

Figura 3.4. 6 Gráfico de comparação entre os tempos de multiplicação de matrizes quadradas via biblioteca Eigen e método convencional.

A Tabela 3.4.6 e o gráfico da Figura 3.4.6 demonstram um desempenho melhor por parte do uso da biblioteca *Eigen* para matrizes de maiores dimensões. Essa melhoria é observada a partir de valores de 130x130 (16900 elementos). Todavia, para matrizes de tamanho menores o tempo demandado é praticamente idêntico entre o uso dos dois métodos aqui discutidos. Assim, mediante a similaridade de tempo para matrizes de tamanhos abaixo de 130x130, deve-se levar em conta na hora da escolha do uso ou não da biblioteca *Eigen* o fato da mesma apresentar uma implementação facilitada e elegante, permitindo uso de expressões matriciais e funções que as envolve, como cálculo de normas, podendo justificar seu uso.

3.5 Conclusão do Capítulo

Os *benchmarks* apresentados nas seções anteriores exploraram características da placa *Parallella* que são fundamentais para a implementação do algoritmo MPC proposto neste trabalho. Os resultados obtidos possibilitam uma análise quantitativa do custo de processamento envolvido em cada operação testada, servindo de base para se mensurar e identificar quais são os potenciais gargalos do sistema.

É evidente que *benchmarks* tratam pontos isolados e, por consequência, costumam apresentar resultados diferentes dos encontrados em aplicações completas. Todavia, isso não invalida sua importância, pois serve como base experimental para a tomada de decisões nos desenvolvimentos futuros.

A fim de se avaliar experimentalmente alguns aspectos do desempenho da plataforma *Parallela* foram feitos os seguintes experimentos: (a) comparação do tempo demandado para o cálculo de operações (multiplicação) matriciais de forma serial e paralela, (b) escrita-leitura do ARM na memória compartilhada, (c) verificação de desempenho da biblioteca Eigen para operações matriciais (quando embarcada para arquitetura ARM) e (d) verificação do custo de carga dos programas a serem executados da memória interna dos *eCores*.

Tais testes foram pensados devido ao fato do algoritmo MPC utilizado neste trabalho ter seus cálculos baseados em operações matriciais e vetoriais, além de uma forte dependência de dados (fator que aumenta a troca dos mesmos entre os elementos de processamento envolvidos) e pela limitação de tamanho da memória interna dos *eCores*.

Assim, como resultado, verificou-se que o MPSoC *Epiphany* apresenta um bom desempenho ao multiplicar matrizes por meio do paralelismo fornecido pelos seus *eCores*, frente a execução serial de tais operações (aqui testadas no ARM Dual Core A9, presente na placa *Parallella*). Além disso, verificou-se que o custo de acesso à memória compartilhada, e consequentemente de comunicação ARM/*Epiphany*, pode representar um gargalo ao sistema, sendo esta uma operação que deve ser minimizada tanto quanto possível.

Por fim, constatou-se que existe um alto custo temporal em se carregar programas executáveis na memória interna dos *eCores*, o que leva a conclusão de que a carga de programas durante a fase de execução *online* do algoritmo MPC (vide Capítulo 4) deve ser evitada. Assim, diante da grande limitação de memória local dos *cores Epiphany*, deve-se estabelecer uma estratégia de paralelização que leve em conta o tamanho dos programas a serem carregados nos *eCores* de modo a maximizar a utilização dos mesmos (visando os ganhos de desempenho obtidos com a paralelização), e ao mesmo tempo, minimizar a troca de dados entre o *host* e o *device*, minimizando assim o tempo de *overhead* proveniente da comunicação dos mesmos.

Contudo, o algoritmo MPC utilizado é intrinsicamente serial (vide Capítulo 2 e Capítulo 4), acrescentando uma dificuldade a mais na escolha da estratégia de paralelismo a ser adotada, como será abordado no Capítulo 4.

Por fim, pode-se concluir que os resultados obtidos estão em concordância com o que se esperava da literatura relacionada ao *Epiphany* e dos conhecimentos teóricos, tanto da plataforma de *hardware* utilizada quanto do conceito de programação paralela, trazendo informações adicionais acerca das funcionalidades testadas.

Capítulo 4

Um Estudo de Caso Utilizando Aplicações de Controle Preditivo Baseado em Modelo

O presente capítulo traz como objetivo a análise da plataforma *Parallella* utilizando aplicações MPC como um estudo de caso, devido seu interesse para o grupo de pesquisa do laboratório LEIA (vide Capítulo 1).

Sabe-se, mediante os trabalhos correlatos e documentações técnicas, [62] e [63], que a plataforma em questão possui sérias restrições de memória, além de elevado tempo de comunicação entre seus elementos de processamento (vide Capítulo 3). Tais características, aliada a alta dependência de dados inerente ao algoritmo MPC proposto, deverão impactar negativamente na avaliação de desempenho aqui realizadas.

Todavia, tendo em conta o baixo custo da plataforma e seu crescimento recente em aplicações diversas de engenharia, acha-se importante realizar uma abordagem inicial de caráter experimental, a fim de se obter informações sobre o potencial de *speedup*, quando aplicada a um problema com alta dependência de dados como é o caso do algoritmo MPC proposto por [38] (vide Capítulo 2), e descritos em detalhes nas seções seguintes.

As informações obtidas com a realização das etapas 1 e 2 deste trabalho (vide Capítulo 1) servirão como referência para as tomadas de decisões necessárias à criação de estratégias de paralelização a serem avaliadas. Tais estratégias serão testadas previamente em uma aplicação MPC simples (integrador triplo [38]), sendo estendida ao problema de controle de atitude de uma plataforma de satélites [75].

Para tanto, as subseções seguintes tratarão dos aspectos relativos às etapas de 3 a 6, apresentadas no Capítulo 1, analisando o algoritmo MPC linear proposto, do ponto de vista de suas características, princípios de funcionamento e mapeamento do custo computacional de suas funções. Por fim, serão exibidas as principais estratégias de paralelização testadas, bem como a configuração de melhor desempenho encontrada.

4.1 Características da Aplicação

Antes de se embarcar as aplicações propostas pelo presente trabalho deverão ser levantados alguns pontos importantes acerca do algoritmo MPC-LTI utilizado (vide Capítulo 2), de modo a facilitar a escolha da estratégia empregada. Este é o objetivo desta subseção.

a) Modos de Execução

O algoritmo conta com dois modos básicos de execução:

- Execução *offline*: Fase inicial em que acontece a preparação das matrizes utilizadas para cálculo da função custo e restrições.

- Execução *online*: Fase final em que são computados os sinais de controle.

A fase *online* é a de maior custo computacional, contando basicamente com dois grandes *loops*, um externo e outro interno. No *loop* externo são atualizados os valores da referência e calculadas as matrizes de custo F e de restrições B_{ineq}, que serão posteriormente utilizadas na resolução da função custo:

$$F = F_1 * lesx^T + F_2 * Y_{ref} \quad (4.1.1)$$

$$B = G_1 * lesx^T + G_2 * G_3 \quad (4.1.2)$$

em que:

$F \rightarrow Matriz\ de\ Custo.$
$B_{ineq} \rightarrow Matriz\ de\ Restrições.$
$lesx \rightarrow vetor\ de\ estados\ do\ sistema$
$Y_{ref} \rightarrow vetor\ do\ sinal\ de\ referência$
$G_1, G_2\ e\ G_3 \rightarrow são\ matrizes\ intermediárias\ utilizadas\ para\ o\ cálculo\ da\ matriz\ de\ restrições\ B\ (vide\ Capítulo\ 1)$

O *loop* interno corresponde, basicamente, à obtenção dos valores de controle ótimo via minimização da função custo (otimizador) e se caracteriza como o trecho de maior custo computacional em todo o algoritmo, resumido pelas seguintes matrizes (vide Capítulo 2).

$$inter = A_{ineq}.p - B_{ineq} \quad (4.1.3)$$

$$G = H * lesp + F + 2.\rho.A^T.\max\_matrix(0, inter) \quad (4.1.4)$$

$$p_1 = lesp - \frac{1}{h_{max} * G} \quad (4.1.5)$$

$$p_2 = lesp - \frac{\gamma}{h_{max} * G} \quad (4.1.6)$$

$$inter_1 = A_{ineq}.p_1 - B_{ineq} \quad (4.1.7)$$

$$inter_2 = A_{ineq}.p_2 - B_{ineq} \quad (4.1.8)$$

$$J_1 = \frac{1}{2}p_1^T.H.p_1 + F^T.p_1 + ||\rho.max\_matrix(0, inter_1)||_2 \quad (4.1.9)$$

$$J_2 = \frac{1}{2}p_2^T.H.p_2 + F^T.p_2 + \quad ||\rho.max\_matrix(0, inter_2)||_2 \quad (4.1.10)$$

em que:

$h_{max} \rightarrow limite\ para\ cálculo\ do\ incremento\ do\ passo\ \alpha\ da\ iteração.$
$lesp \rightarrow vetor\ de\ variáveis\ de\ decisão.$
$p_1, p_2 \rightarrow vetores\ da\ variável\ de\ decisão\ para\ o\ cálculo\ de\ J_\gamma\ e\ J.$
$inter \rightarrow\ vetor\ sobre\ o\ qual\ se\ fará\ o\ cálculo\ prévio\ do\ gradiente.$
$G \quad \rightarrow matriz\ relativa\ ao\ cálculo\ do\ gradiente.$
$inter_1\ e\ inter_2$
$\rightarrow vetores\ relativos\ ao\ cálculo\ do\ gradiente\ com\ e\ sem\ expansão.$
$J_y \rightarrow Valor\ obtido\ pela\ função\ custo\ (Gradiente\ com\ expansão\ do\ passo)$
$J \rightarrow Valor\ obtido\ pela\ função\ custo\ (Gradiente\ sem\ expansão\ do\ passo)$

A Figura 4.1.1 mostra a estrutura básica da parte *online* do algoritmo, em que o otimizador (*loop* interno) aparece destacado em vermelho.

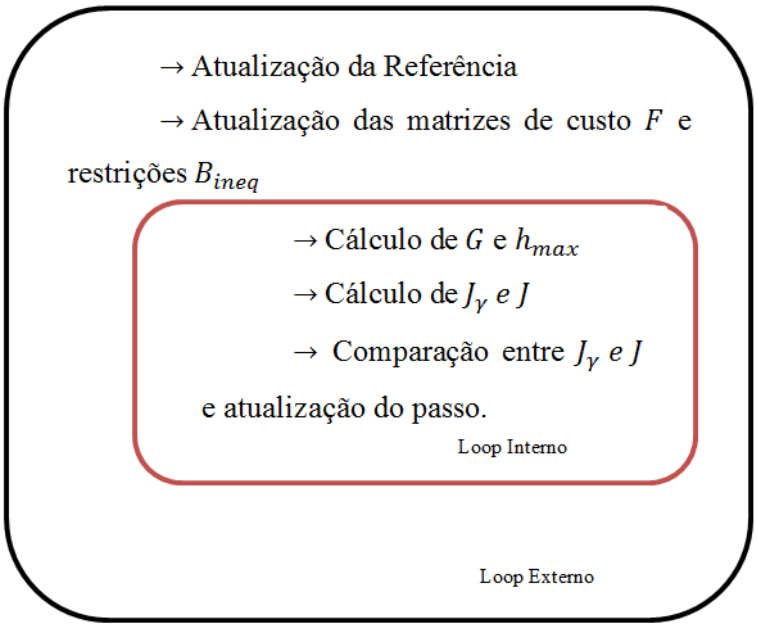

Figura 4.1. 1 Representação da estrutura do algoritmo relativo à aplicação MPC.

A Algoritmo 4.1.1 ilustra o trecho de código em linguagem C correspondente ao *loop* interno (otimizador). Neste caso, a matriz G (relativa ao gradiente, linha 11) é calcula segundo a equação (4.1.4), uma vez já calculado o vetor $inter$ (linha 9 e equação (4.1.3)), com base no vetor das variáveis de decisão ($lesp$) e mediante as matrizes de restrição A_{ineq} e B_{ineq}.

Assim, pode-se calcular os vetores p_1 (linha 13 e equações (4.1.5)) e p_2 (linha 14 e equação (4.1.6)). Esses vetores representam a variável de decisão no instante seguinte, ou seja, após a aplicação do método do gradiente simples (p_1) e com expansão (p_2). Por fim, após a verificação das restrições sobre os mesmos com $inter_1$ e $inter_2$ (linhas 15, 16 e equações (4.1.7) e (4.1.8) respectivamente), são calculados os valores da função custo para ambos os métodos supracitados (gradiente com e sem expansão). Esse último cálculo é representado por J e J_γ (linhas 17, 18 e equações (4.1.9) e (4.1.10) respectivamente), sendo γ o valor a multiplicar o passo do método do gradiente com expansão (atualizado nas linhas 22 ou 26, dependendo do resultado da comparação entre J e J_γ).

Algoritmo 4.1. 1 Trecho de código em linguagem C referente ao otimizador utilizado no algoritmo MPC proposto.

```
(01) for (int i = 0; i < this->Niter-1; ++i) {
(02)         hmax = hmax0 + 2*rho*hmaxg;
(03)
(04)         if (indrho == this->nrho) {
(05)             rho = fmin(this->rho_max, this->beta_plus_rho*rho);
(06)             indrho = 1;
(07)         }
(08)
(09)         inter = this->A*lesp.col(i) - this->B;
(10)
(11)         G = this->H*lesp.col(i) + this->F+2*rho*this->A.transpose()*max_matrix(0, inter);
(12)
(13)         p1 = lesp.col(i)-1/(hmax)*G;
(14)         p2 = lesp.col(i)-gam/(hmax)*G;
(15)         inter1 = this->A*p1-this->B;
(16)         inter2 = this->A*p2-this->B;
(17)         J1 = (0.5*p1.transpose()*this->H*p1*this->F.transpose()*p1)(0,0)+rho*pow(max_matrix(0,inter1).norm(),2);
(18)         J2 = (0.5*p2.transpose()*this->H*p2*this->F.transpose()*p2)(0,0)+rho*pow(max_matrix(0,inter2).norm(),2);
(19)
(20)         if (J1 < J2) {
(21)             lesp.col(i+1) = p1;
(22)             gam = fmax(gam_min, beta_minus*gam);
(23)         }
(24)         else {
(25)             lesp.col(i+1) = p2;
(26)             gam = beta_plus * gam;
(27)         }
(28)         for (int j = 0; j < np; ++j) {
(29)             lesp(j,i+1) = fmin(this->pmax(j,0), fmax(this->pmin(j,0),lesp(j,i+1)));
(30)         }
(31)         indrho = indrho + 1;
(32)}
```

b) Dependência de Dados

Para se paralelizar um algoritmo é fundamental determinar suas partes paralelizáveis. Nesse sentido, existem basicamente duas formas gerais em que se pode paralelizar o algoritmo MPC utilizado neste trabalho:

- *Paralelização de expressões matemáticas complexas (geralmente entre matrizes de elevado tamanho)*: situação em que uma única expressão é paralelizada para que o cálculo final seja segmentado em partes menores calculadas em paralelo.

- *Paralelização de trechos de códigos independentes*: situação em que existam trechos de códigos independentes entre si podendo ser divididos em tarefas independentes a serem executadas em diferentes núcleos de processamento.

Em se tratando da aplicação MPC utilizada nesta pesquisa, verifica-se uma forte dependência de dados inerente ao algoritmo e consequentemente um baixo potencial de paralelismo. Tal fato é observado ao se analisar a própria mecânica de funcionamento do mesmo e no *profile* realizado na seção 4.3.

A cada instante de tempo são calculados N ações de controle, em que N representa o horizonte de predição. Todavia, o cálculo do próximo passo possui dependência com o passo anterior, sendo que isso se estende ao cálculo dos N pontos do horizonte. Assim, verifica-se que o princípio de operação básica do algoritmo MPC, que consiste na predição do sinal de controle a cada instante de tempo, é de caráter serial.

O que se nota é que as expressões (em sua maioria) são dependentes entre si, no sentido de que o próximo cálculo depende de resultados obtidos no cálculo anterior, sendo isso consequência da observação supracitada. Isso dificulta a divisão do algoritmo em tarefas independentes e que possam ser executadas em núcleos de processamento diferentes.

Uma exceção é o cálculo da função custo que é realizado de dois modos distintos e independentes, relativos ao tamanho do passo a ser adotado no cálculo do gradiente, como discutido no Capítulo 2.

Todavia, é sabido que o algoritmo MPC trabalha com o modelo em espaço de estados do sistema a ser controlado. Assim, as operações realizadas são basicamente de natureza matricial. Devido à dependência entre as operações descrita acima, uma alternativa à paralelização seria a utilização dos diferentes núcleos de processamento para calcular partes de uma mesma expressão matricial, ou seja, quanto maior o tamanho das matrizes envolvidas melhor seria o ganho em termos de desempenho computacional, como comprovado nos testes de *benchmark* realizados neste trabalho.

Um detalhe a ser observado é que as operações de maior custo computacional estão descritas dentro do *loop* interno (otimizador), sendo executado um número maior de vezes. Neste sentido, uma boa estratégia seria focar a paralelização neste trecho de maior volume de operações. A fim de se levantar a contribuição de cada uma das funções supracitadas ao custo final do algoritmo, realizou-se um *profile* do algoritmo do integrador triplo, apresentado em detalhes na seção 4.3.

A Figura 4.1.2, fruto do *profile* realizado em 4.3, ilustra as dependências característica do algoritmo MPC utilizado no presente trabalho, enumerando os blocos contendo as expressões a serem calculadas de (1) a (14). Vale ressaltar que a análise foi desenvolvida apenas para o modo *online* de execução do algoritmo, sendo esse o modo de interesse nesta pesquisa.

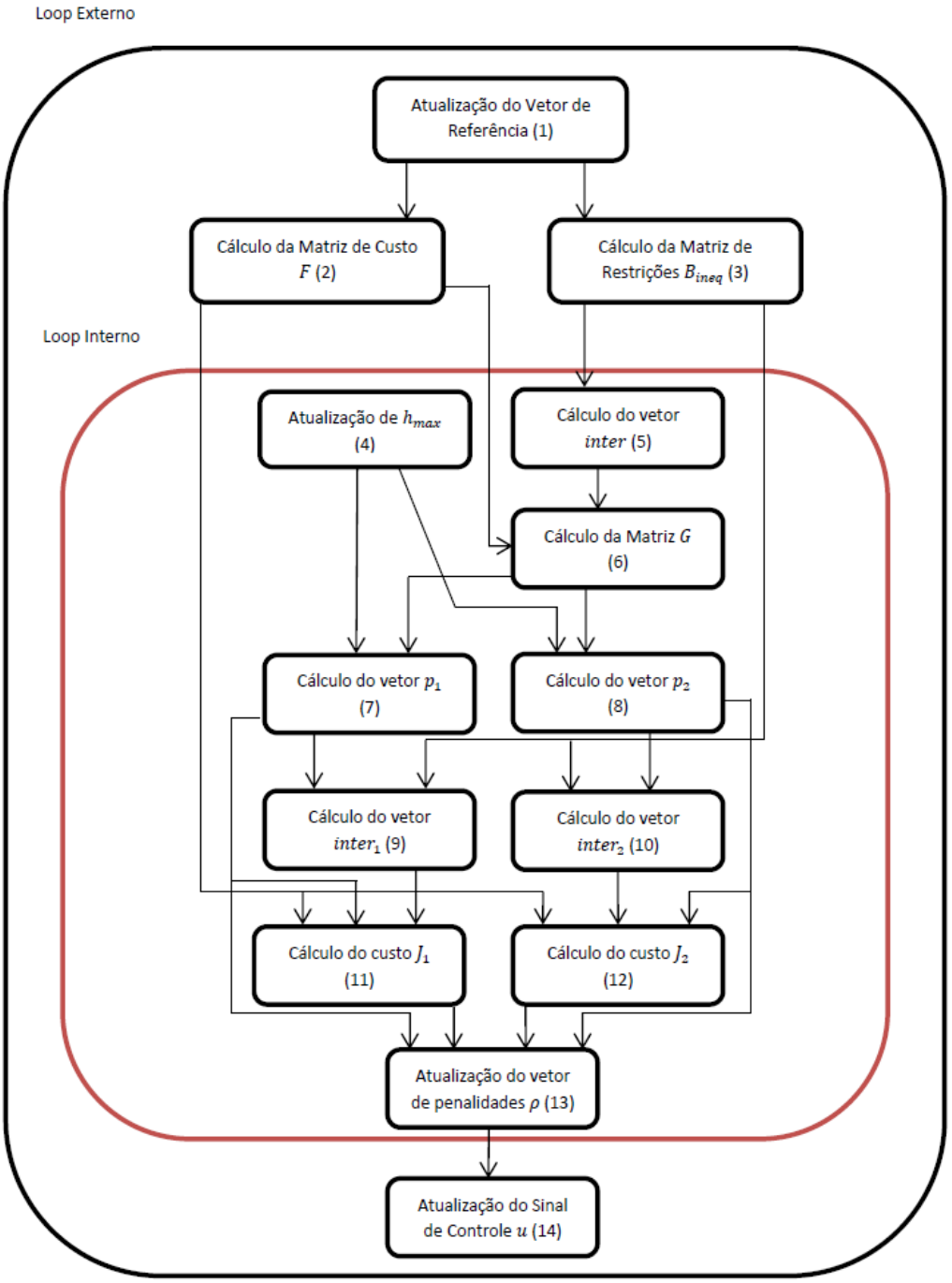

Figura 4.1. 2 Dependência das tarefas no algoritmo relativo à aplicação MPC.

A Figura 4.1.3 traz uma representação simplificada, identificando os blocos da Figura 4.1.2 apenas por seus números e enfatizando a dependência existente entre as tarefas, fator que dificulta sobremaneira o paralelismo do algoritmo.

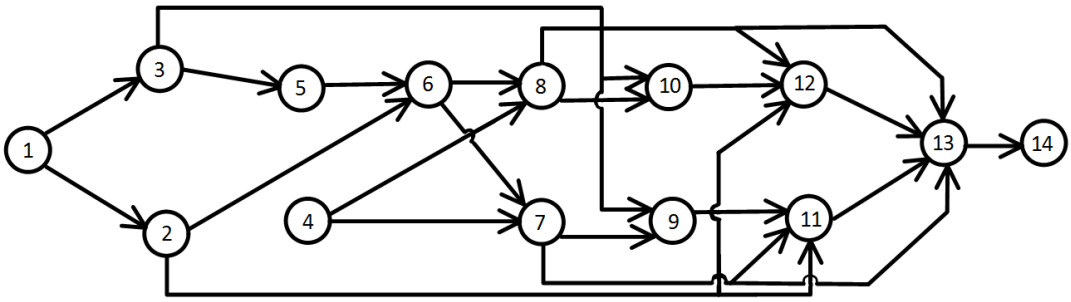

Figura 4.1. 3 Fluxo de execução do algoritmo MPC proposto.

4.2 Aplicações MPC a serem embarcadas

Nesta seção são apresentadas as aplicações MPC utilizadas no presente trabalho como um estudo de caso, a saber, o integrador triplo e o controle de atitude de uma plataforma de satélites.

4.2.1 Sistema Integrador Triplo

A fim de se testar as estratégias de paralelização do algoritmo MPC proposto, utilizou-se a aplicação MPC linear com restrições, referente ao controle de rastreamento de um sistema integrador triplo, baseando-se em [38], com o intuito de explorar técnicas de paralelismo e verificar seus desempenhos. Neste caso, o objetivo é validar tais técnicas em uma aplicação simplificada para então estender a implementação à aplicação do controle de atitude de uma plataforma de satélites.

a) Modelo da Aplicação do Integrador Triplo

Para implementação e testes das estratégias definidas neste trabalho foi utilizado o MPC com objetivo de se seguir um sinal de referência dado para uma onda quadrada de amplitude 1 e período 10 s e ciclo de trabalho de 0.5, aplicado a um sistema integrador triplo baseado em [38], como já mencionado.

O modelo da presente aplicação é composto pelas matrizes:

$$A = \begin{bmatrix} 1.0 & 0.1 & 0.005 \\ 0.0 & 1.0 & 0.1 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (4.2.1) \qquad \rightarrow Matriz\ de\ Estados$$

$$B = \begin{bmatrix} 0.0001666 & 0.005 & 0.1 \end{bmatrix} \quad (4.2.2) \qquad \rightarrow Vetor\ de\ Entradaas$$

$$C_r = \begin{bmatrix} 1.0 & 0.0 & 0.0 \end{bmatrix} \quad (4.2.3) \qquad \rightarrow Vetor\ de\ saídas$$

$$C_c = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix} \quad (4.2.4) \qquad \rightarrow Matriz\ de\ restriçoes$$

As restrições são referentes a um valor máximo e mínimo do sinal de controle u e da saturação inerente à variação mesmo, além das restrições do sinal de saída e de sua derivada, como mostrado a seguir.

$u \in [u^{max}, u^{min}]$ (4.2.5) → *restrição de saturação na atuação de controle*;

$u(k) - u(k-1) \in [\delta^{min}, \delta^{max}]$ (4.2.6) → *restrição de saturação na taxa de variação do sinal de controle*;

$\begin{pmatrix} y_r \\ \dot{y}_r \end{pmatrix} \in [y_c^{min}, y_c^{max}]$ (4.2.7) → *restrições na saída do sistema e em sua derivada*;

Detalhes mais específicos sobre o modelo do problema em questão podem ser verificados em [38].

b) Parâmetros de simulação

A presente aplicação foi implementada tendo em conta os seguintes parâmetros (definidos em [38]):

- *Horizonte de predição* → $N = 20$
- *Tempo total de simulação* → 20 *segundos*
- *Período de Amostragem* → 0.1 *segundos* $(100\ ms)$

Visando diminuir o número de iterações do cálculo da função custo, foi traçada uma curva de convergência do algoritmo utilizado, representada pelo gráfico da Figura 4.2.1.

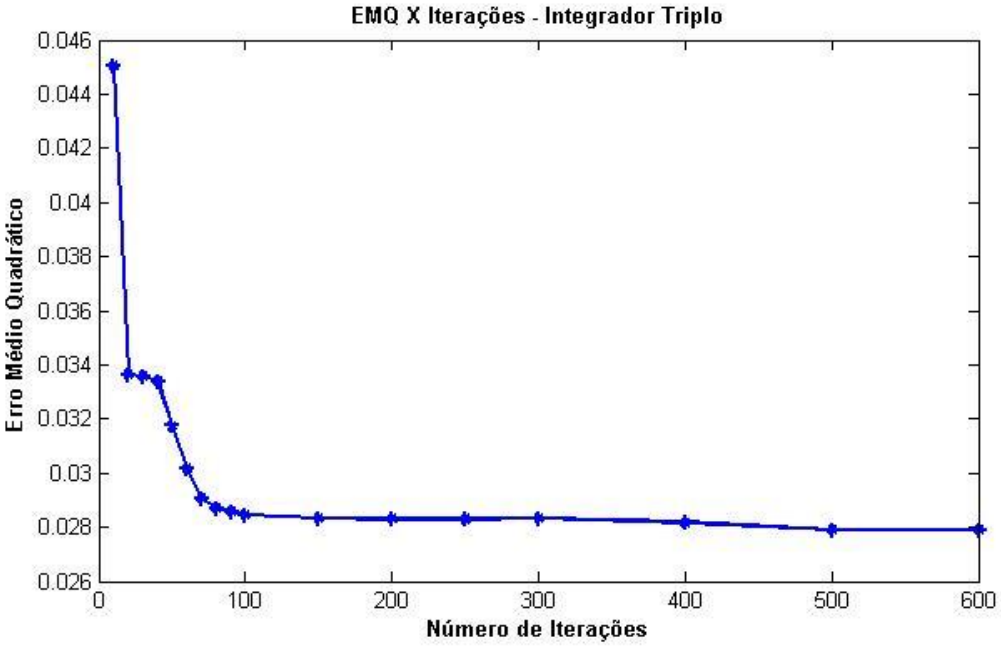

Figura 4.2. 1 Curva de convergência do erro médio quadrático pelo número de iterações do algoritmo de resolução da função custo.

O gráfico da Figura 4.2.1 foi traçado levando-se em conta o erro médio quadrático da saída do controlador com relação ao sinal de referência. Percebe-se que tal erro se estabiliza após um valor de aproximadamente 90 iterações. Assim, tendo em conta este resultado, os testes de implementação da presente aplicação foram realizados mediante um valor fixo de 100 iterações.

Com os dados supracitados obtêm-se uma configuração em que o *loop* externo do algoritmo será executado 200 vezes. Assim, com 100 iterações para se resolver a função custo, as expressões do *loop* interno ocorrerão 20000 vezes, o que explica ser esse o trecho do algoritmo correspondente a 99,86% do custo computacional, como constatado no *profile* do algoritmo apresentado na seção 4.3.

c) Dimensões das Matrizes de Interesse

Consideram-se como matrizes de interesse as matrizes que compõe as expressões de maior custo computacional, listadas na Tabela 4.3.1. da seção seguinte. Desse modo, suas dimensões, levando em conta as configurações supracitadas, são:

- Matriz da variável de decisão p $lesp = 20x100$
- $A_{ineq} = 120x20$
- $B_{ineq} = 120x1$
- $inter, inter_1 e\ inter_2 = 120x1$
- $p_1, p_2\ e\ G = 20x1$

em que $lesp$ é a matriz relativa a variável de decisão.

4.2.2 MPC aplicado ao Sistema de Controle de Atitude de Plataforma de Satélites Artificiais

Juntamente com a aplicação do integrador triplo, utilizou-se o algoritmo MPC paralelo ao controle de atitude de satélites artificiais, desenvolvido por [75]. As seções seguintes trazem os detalhes de tal aplicação.

a) Contextualização

Os satélites artificiais são sistemas robóticos extremamente complexos e caros, empregados em aplicações científicas, militares e de comunicação. O controle de atitude para este sistema é responsável por assegurar que o objeto espacial se encontre na posição, velocidade e trajetória corretas, estabilizando o veículo espacial e o orientando nas direções desejadas durante a missão, independente de perturbações externas nele atuantes.

É importante que o projeto do controlador seja desenvolvido visando atender aos requisitos de funcionamento do satélite, possuindo a capacidade de lidar com as diversas restrições presentes no sistema, caso contrário podem ocorrer falhas ou funcionamento inadequado do mesmo [75].

Assim, a aplicação desenvolvida por [75] refere-se em ao algoritmo de um controlador MPC na sua formulação em espaço de estados, utilizando restrições das variáveis de estado e dos atuadores, assim como da sua taxa de variação para a plataforma de teste de satélites apresentada em [80]. Vale ressaltar que se trata de um controlador MPC linearizado para controlar o modelo não linear da plataforma.

b) Modelo do Sistema de Controle da Plataforma de Satélites

O modelo do sistema refere-se à plataforma de testes de satélite representada na Figura 4.2.2. Como pode ser visto, o simulador é uma mesa giratória de três eixos suportada por um mancal esférico. O uso do mancal esférico é necessário para simular as condições em que é desejado controle de atitude em três eixos de rotação. Em contrapartida, possui a limitação de permitir a rotação completa em apenas um dos eixos.

Figura 4.2. 2 Plataforma de três eixos [75].

Nesta plataforma os elementos responsáveis por manter a orientação desejada são as rodas de reação. A roda de reação é um motor ligado a um volante de alta inércia que é livre para girar ao longo de um eixo fixo da espaçonave [41]. Ela utiliza o princípio da conservação do momento angular, que afirma que um sistema sem a ação de torques externos, a quantidade de momento angular é conservada.

Quando o motor começa a girar, induzindo uma velocidade na roda de reação, surge um torque de mesma intensidade e direção oposta que é aplicado à plataforma. O sistema de referencia inercial $\mathrm{F_i}(\mathrm{i_1}, \mathrm{i_2}, \mathrm{i_3})$ está localizado no centro do mancal esférico, considerado como o centro de rotação do simulador. O sistema de referencia do corpo $\mathrm{F_b}$ é considerado como tendo o mesmo centro, variando-se apenas a orientação do mesmo com relação ao sistema inercial.

Além disso, considera-se também que o sistema de referência do corpo está orientado conforme os eixos principais de inércia da mesa giratória.

Denomina-se *atitude do simulador* a orientação relativa entre o sistema de referência inercial F_i e o sistema de referência do corpo F_b fixo à mesa giratória. Para descrever a orientação de F_b com relação a F_i, utilizam-se os ângulos de Euler na sequência de rotações 3-2-1, ou seja, partindo-se de F_i para se chegar a F_b, o eixo 3 deverá ser rotacionado de um ângulo θ_1, o eixo 2 deverá ser rotacionado de um ângulo θ_2, e o eixo 1 deverá ser rotacionado de um ângulo θ_3.

As equações que governam o comportamento dinâmico da plataforma foram obtidas de [41], onde pode se consultar o desenvolvimento completo. A modelagem do sistema é obtida a partir do teorema de Euler para o momento angular:

$$\dot{\vec{h}} = \vec{g}, \tag{4.2.1}$$

em que $\vec{g}$ é a somatória dos torques externos ao redor do centro de massa e $\dot{\vec{h}}$ é o momento angular ao redor do centro de massa. Utilizando a notação de [81]:

$$\vec{h} = \vec{I}\vec{w} + I_w(\Omega_1 + W_1)\hat{b}_1 + I_w(\Omega_2 + W_2)\hat{b}_2 + I_w(\Omega_3 + W_3)\hat{b}_3\,, \tag{4.2.2}$$

em que $I = diag(I_{11}, I_{22}, I_{33})$ é o tensor de inércia do simulador e I_{11}, I_{22}, I_{33} são os momentos de inércia em torno dos eixos $i_1, i_2\ e\ i_3$, respectivamente. $\vec{w}$ é a velocidade angular de $F_b$com relação a F_i. Utilizando os termos $\vec{I}_w = diag(I_w, I_w, I_w)$ chamado tensor de inércia das rodas de reação, todos iguais a I_w e $\vec{\Omega} = (\Omega_1, \Omega_2, \Omega_3)$, que representa a velocidade angular das rodas de reação nos seus respectivos eixos, a equação pode ser rescrita como:

$$\vec{h} = \vec{I}\vec{w} + \vec{I}_w\left(\vec{\Omega} + \vec{w}\right). \tag{4.2.3}$$

Diferenciando a equação (4.2.3) utilizando a regra de [50] e considerando $\vec{g} = 0$, chega se na expressão:

$$\dot{w} = (I + I_w)^{-1}[-w^x(I + I_w)w - w^x I_w\Omega - I_w\dot{\Omega}], \tag{4.2.4}$$

em que w^x é a chamada matriz *skew-symmetric*, definida por:

$$w^x = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix}, \tag{4.2.5}$$

sendo $w_1, w_2\ e\ w_3$ as velocidades angulares de F_b com relação a F_i. A atitude do simulador como função da velocidade angular utilizando a sequência de rotação de ângulos de Euler 3-2-1 é dada por:

$$\begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix} = \begin{bmatrix} 0 & \dfrac{\sin(\theta_3)}{\cos(\theta_2)} & \dfrac{\cos(\theta_3)}{\cos(\theta_2)} \\ 0 & \cos(\theta_3) & -\sin(\theta_3) \\ 1 & \dfrac{\sin(\theta_3)\sin(\theta_2)}{\cos(\theta_2)} & \dfrac{\cos(\theta_3)\sin(\theta_2)}{\cos(\theta_2)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}. \tag{4.2.6}$$

Neste modelo, as variáveis $\theta_1, \theta_2 \, e \, \theta_3$ são os ângulos que descrevem a atitude do simulador como a orientação relativa entre o referencial inercial F_i e a referência fixada ao corpo F_b. A partir das equações (4.2.4) e (4.2.6) busca-se representar o modelo do sistema em espaço de estados em que o vetor de estados é definido por $x = (\theta_1 \; \theta_2 \; \theta_3 \; w_1 \; w_2 \; w_3)^T$ e o vetor de entrada é definido como $u = \left(\dot{\Omega}_1 \; \dot{\Omega}_2 \; \dot{\Omega}_3\right)^T$, correspondente às acelerações das rodas de reação [75].

A implementação do presente trabalho baseia-se em uma linearização do modelo não linear de controle de atitude da plataforma de satélites em questão. Assim, o ponto operacional escolhido para o cálculo da matriz linear é aquele para pequenas excursões em torno da origem, ou seja, pequenas variações de ângulo, velocidade angular e aceleração.

Desse modo, a matriz de estados é definida por [75]:

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.1 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.1 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \rightarrow Matriz\ de\ Estados$$

Já as matrizes de entrada, saídas e restrições (restringindo-se apenas aos estados $\theta_1, \theta_2 \, e \, \theta_3$), são definidas por [75]:

$$B = \begin{bmatrix} 0.0 & 0.0 & -4.2\ 10^{-6} \\ 0.0 & 77\ 10^{-6} & 0.0 \\ 77\ 10^{-6} & 0.0 & 0.0 \\ 153.6\ 10^{-6} & 0.0 & 0.0 \\ 0.0 & 153.6\ 10^{-6} & 0.0 \\ 0.0 & 0.0 & 84.4\ 10^{-6} \end{bmatrix} \rightarrow Matriz\ de\ Entradas$$

$$C_r = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \rightarrow Matriz\ de\ Saídas$$

$$C_c = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \rightarrow Matriz\ de\ Restrições$$

Todo o desenvolvimento acima foi baseado em [75], onde se podem obter maiores detalhes da modelagem do sistema.

c) Parâmetros de Simulação

A presente aplicação foi implementada mediante os seguintes parâmetros cuja escolha foi baseada em [75] (mantendo-se o horizonte de predição adotado ao caso do integrador triplo):

- $Horizonte\ de\ predição \rightarrow\ N = 20$
- $Tempo\ total\ de\ simulação \rightarrow 20\ segundos$
- $Período\ de\ Amostragem \rightarrow 0.1\ segundos\ (100\ ms)$

Semelhante ao realizado com a aplicação do *Integrador Triplo*, visando diminuir o número de iterações do cálculo da função custo, foi traçada uma curva de convergência do presente algoritmo, observando o mesmo comportamento apresentado pelo gráfico da Figura 4.2.1. Assim, os teste de implementação da presente aplicação foram realizados mediante um valor fixo de 100 iterações.

d) Dimensões das Matrizes de Interesse

Consideram-se como matrizes de interesse as matrizes que compõe as expressões de maior custo computacional, listadas na Tabela 4.3.1. Desse modo, suas dimensões, levando em conta as configurações supracitadas, são:

- $lesp = 60x100$
- $A_{ineq} = 240x60$
- $B_{ineq} = 240x1$
- $inter, inter_1 e\ inter_2 = 240x1$
- $p_1, p_2\ e\ G = 60x1$

4.3 Execução do Profile do Algoritmo MPC serial do Integrador Triplo

Antes de se tomar as decisões acerca da estratégia de paralelização, foi realizado um mapeamento das expressões que compõem o algoritmo, objetivando a obtenção dos pontos de maior custo computacional. Para tanto, foi utilizado a ferramenta *GNU Profile* (*Gprof*), aplicada sobre a versão serial do algoritmo MPC do integrador triplo.

A ferramenta *Gprof* [76] é uma ferramenta do projeto GNU que faz parte do conjunto de ferramentas binárias *GNU Binutils* [77]. Ela permite a análise dinâmica de programas computacionais através da coleta de informações e identificação de trechos de código mais constantemente solicitados (por um programa em execução), além do tempo consumido por cada rotina de um programa.

Este tipo de informação permite analisar a quantidade de métodos (ou funções) existentes no código, bem como o número de chamadas de cada método e a porcentagem de tempo gasto para executar cada método ou função. Esta informação é relevante, pois mostra quais partes do programa estão demandando mais tempo para serem executadas ou quais partes estão sendo executadas mais ou menos vezes do que era esperado [78].

Tais informações permitirão que se tenha uma visão quantitativa do custo de cada função do algoritmo, auxiliando na estratégia de paralelização a ser adotada.

Como resultado do *profile*, verificou-se que 99,86% do custo de processamento do algoritmo correspondiam ao cálculo da função custo (*loop* interno). Assim, a Tabela 4.3.1 traz uma representação das expressões matriciais mais relevantes que compõe o *loop* interno organizadas em ordem crescente de custo computacional.

Tabela 4.3. 1 Expressões matriciais de maior custo computacional no algoritmo MPC.

Expressão	Custo de processamento em relação ao custo total de execução
$J_1 = \frac{1}{2} p_1^T . H . p_1 + F^T . p_1 + \|\|\rho . max\_matrix(0, inter_1)\|\|_2$	6.319%
$J_2 = \frac{1}{2} p_2^T . H . p_2 + F^T . p_2 + \ \|\|\rho . max\_matrix(0, inter_2)\|\|_2$	6.319%
$inter = A_{ineq} . \rho - B_{ineq}$	16,85%
$inter_1 = A_{ineq} . p_1 - B_{ineq}$	15,798%
$inter_2 = A_{ineq} . p_2 - B_{ineq}$	15,798%
$G = H * lesp + F + 2 . \rho . A^T . max\_matrix(0, inter)$	33,24%

Um fator interessante a ser observado é que o custo de computação da matriz G (expressão de maior custo em termos de tempo), é fortemente influenciado por uma função de maximização de matriz max$_matrix()$, sendo esta ultima correspondente a 10,2% do custo total de execução do algoritmo.

4.4 Estratégias Testadas

O problema em questão apresenta um leque de abordagens possíveis. Sendo assim, o caminho até a estratégia de melhor desempenho implementada passou por inúmeras tentativas cujos resultados não foram satisfatórios, mas que contribuíram para que se chegasse a configuração final da implementação paralela.

Esta foi a etapa de desenvolvimento que demandou maior tempo. Devido a o reduzido potencial de paralelismo do algoritmo MPC, inúmeras estratégias foram testadas, sendo apresentadas a seguir as de maior interesse para o trabalho.

4.4.1 Utilização dos eCores em Operação Específica

Como estratégias inicial, foram utilizados os 16 *eCores* do coprocessador *Epiphany* para o cálculo da matriz $inter$ (vide linha 9 do Algoritmo 4.1.1) e de sua função de maximização

(função *max_matrix(0, inter)* da linha 11 do Algortmo 4.1.1), utilizando-se de um mesmo programa para todos os *eCores.*

Todo o restante do código fora realizado no processador ARM, assim, diminuiu-se os dois principais fatores de elevação do tempo de *overhead*, a saber, (a) o custo de se carregar os programas executáveis nas memórias dos *eCores* e (b) o custo da sincronização de dados ARM/Epiphany. A Figura 4.4.1 ilustra o fluxo de execução do algoritmo MPC com destaque (em verde) para as operações a serem executados no Epiphany (operação (5) e parte da operação (6), vide Figura 4.1.2).

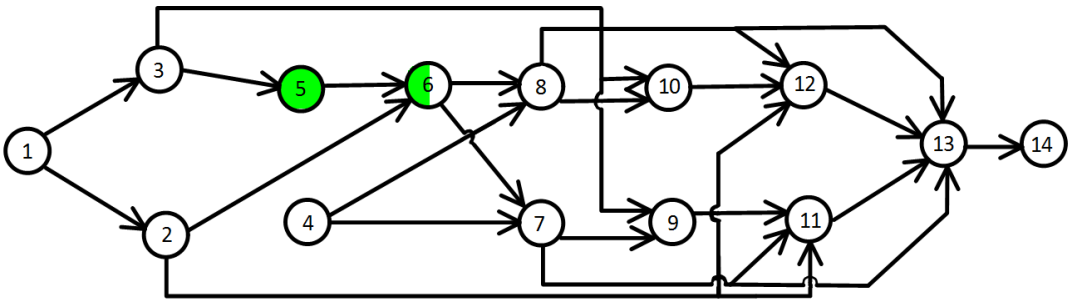

Figura 4.4. 1 Fluxo de execução do algoritmo MPC com destaque para as operações realizadas pelos eCores Epiphany, destacadas em verde.

A comunicação entre o SoC Zynq e o MPSoC *Epiphany* é ilustrada na Figura 4.4.2, ao passo que a configuração dos *eCores* Epiphany estão ilustradas na Figura 4.4.3.

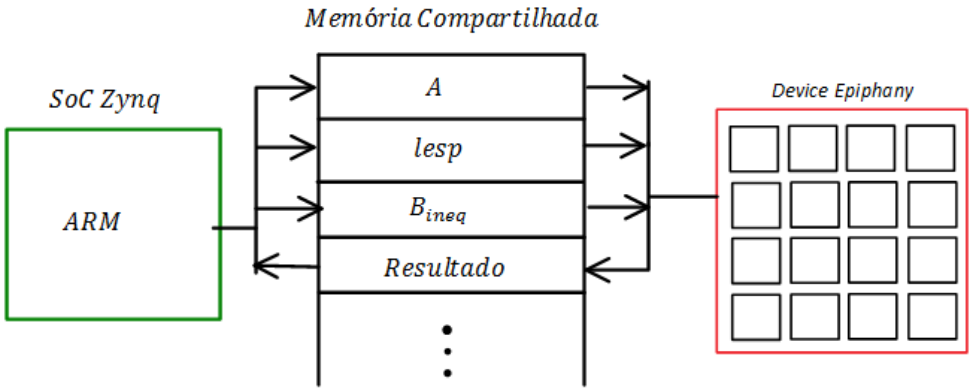

Figura 4.4. 2 Comunicação entre o SoC Zynq e o MPSoC Epiphany.

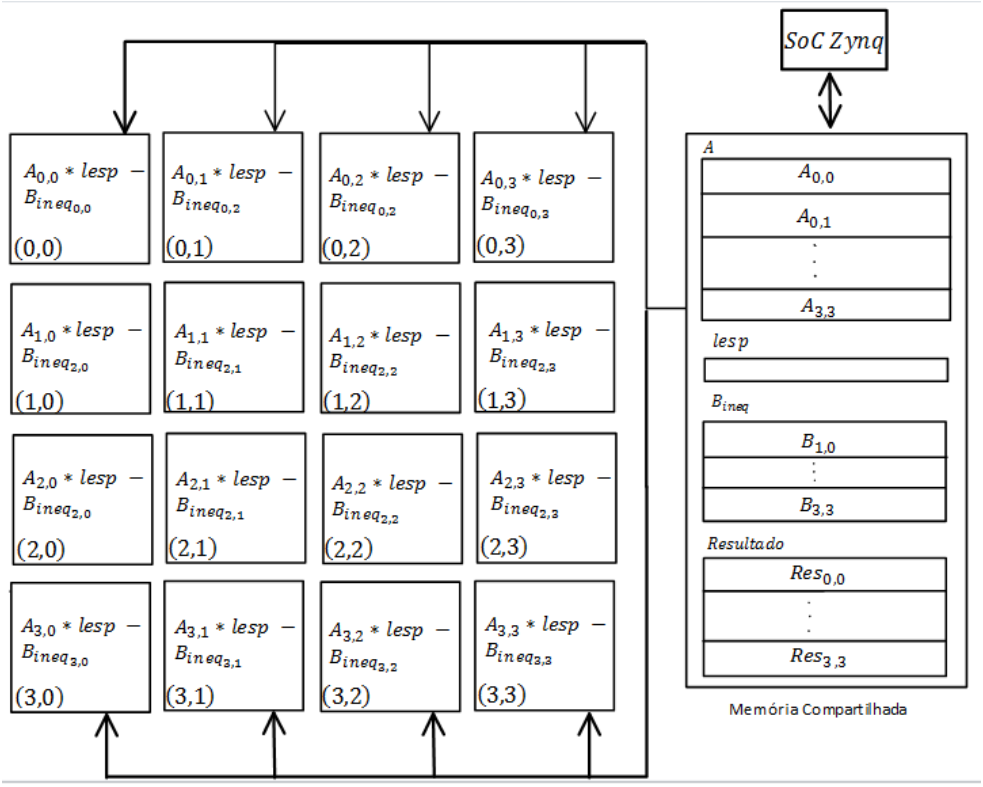

Figura 4.4. 3 Configuração dos eCores Epiphany para cálculo do vetor $inter$.

Dentro da presente estratégia, cada *eCore* copia para sua memória local uma parte específica das matrizes a serem computadas (indicados pelos índices na Figura 4.4.3), realizando os cálculos necessários e escrevendo o resultado na memória compartilhada (também em uma região específica para cada *eCore).*

Essa estratégia apresentou uma pequena melhoria em termos de tempo de execução do programa paralelo (com relação à versão serial), com um *speedup* médio de 1,11. Assim, verificou-se que a pequena melhora de desempenho atrelada a subutilização do coprocessador *Epiphany* não colocava tal estratégia como uma abordagem razoável para os propósitos deste trabalho.

4.4.2 Utilização de PTHREADS

Ainda na linha da estratégia anterior, manteve-se a implementação do lado do coprocessador *Epiphany*, modificando se apenas o lado do algoritmo a ser executado no ARM, explorando o fato de o mesmo ser *dual-core* com a utilização de *threads* via *PTHREADS*.

Foram criadas duas *threads* distintas: (a) uma auxiliar, executando somente as expressões de cálculo de $inter_2, p_2 \, e \, J_2$ (linhas 14, 16 e 18 do Algoritmo 4.1.1) e (b) outra principal, executando o restante do algoritmo. A Figura 4.4.4 representa o fluxo de execução do algoritmo MPC com destaque para as operações realizadas pela *thread* auxiliar (verde escuro) e pelo *Epiphany* (verde claro), sendo as demais operações executadas pela *thread* principal do ARM.

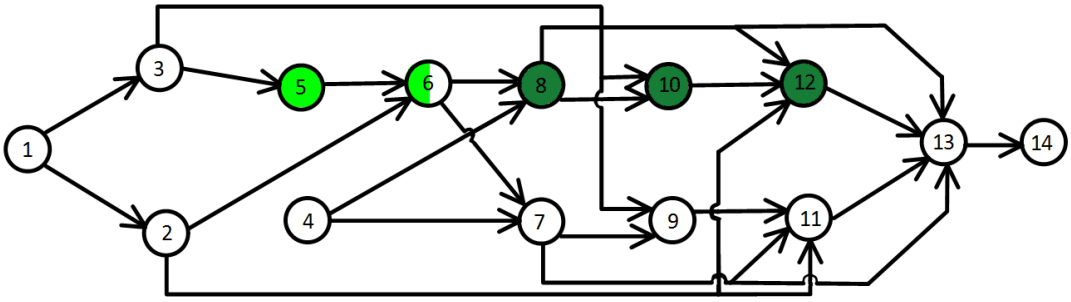

Figura 4.4. 4 Fluxo de execução do algoritmo MPC com destaque para as operações executadas nos eCores Epiphany (verde claro) e pela thread auxiliar do ARM (verde escuro).

Todavia, com a presente técnica foi obtido um tempo de execução paralelo próximo ao tempo de execução serial, levando a um *speedup* bem próximo ao apresentado pela estratégia anterior.

Isso se deve ao custo de sincronização entre as *threads* embalado pela alta dependência de dados, o que levou a alta utilização de mecanismos de controle (como *mutex lock* para se evitar condições de corrida).

4.4.3 Heterogeneidade de Programas

Nesta estratégia, 4 dos 16 eCores foram carregados com programas heterogêneos que implementavam de maneira complementar as expressões do *loop* externo do algoritmo MPC:

$$F = F_1 * lesx^T + F_2 * Y_{ref}$$
$$B = G_1 * lesx^T + G_2 * G_3$$

Assim, após o processador ARM escrever os vetores $lesx$ *e* Y_{rev} na memória compartilhada (as matrizes F_1, F_2, G_1 *e* G_2, são escritas na part *off-line* do algoritmo), cada par de *eCores* fica responsável pela execução de uma das 4 multiplicações de matrizes existentes nas expressões acima, sendo reservado um dentre esses mesmos *eCores* para assumir o papel de somar os resultados e gravá-los nas posições da memória compartilhada destinadas as matrizes F *e* B.

A Figura 4.4.5 ilustra essa primeira etapa do processo, em que o *eCore* (0,0) é o mestre, encarregado de somar os resultados das operações dos demais *eCores* envolvidos nesta primeira etapa e de realizar a sincronização com o processador ARM. O último campo da memória interna (*Res*) representa o resultado das operações realizadas.

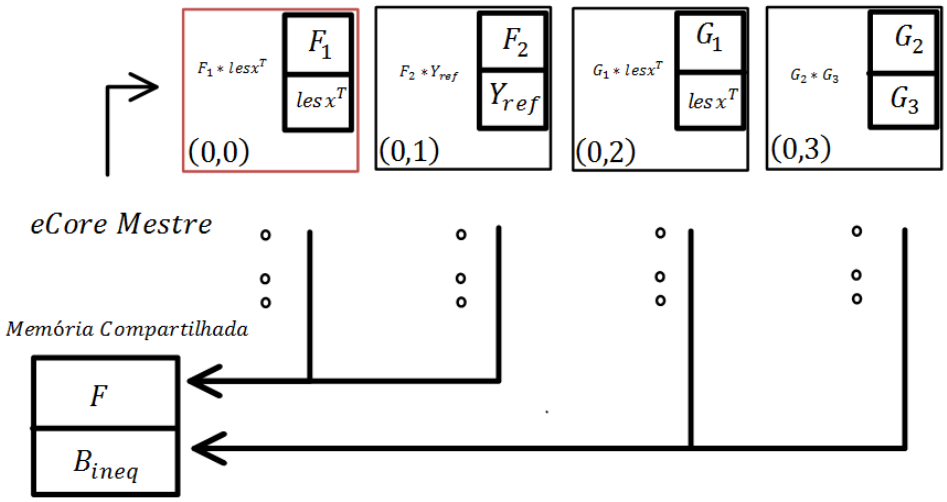

Figura 4.4. 5 Cálculo das matrizes de custo F e restrições B_{ineq}.

Terminado esse processo, os *eCores* enviam um sinal de término de cálculo aos demais nós de processamento da rede e aguardam um sinal dos mesmos para então reiniciarem suas operações. Neste ponto existe uma sincronização global (do tipo *barrier* - Barreira) entre todos os *eCores*.

O processador ARM por sua vez, espera até receber o sinal de finalização por parte do *eCore* mestre para dar início ao *loop* interno, findando a primeira etapa do algoritmo. A Figura 4.4.6 ilustra a comunicação entre ARM e *Epiphany* para esta etapa do algoritmo.

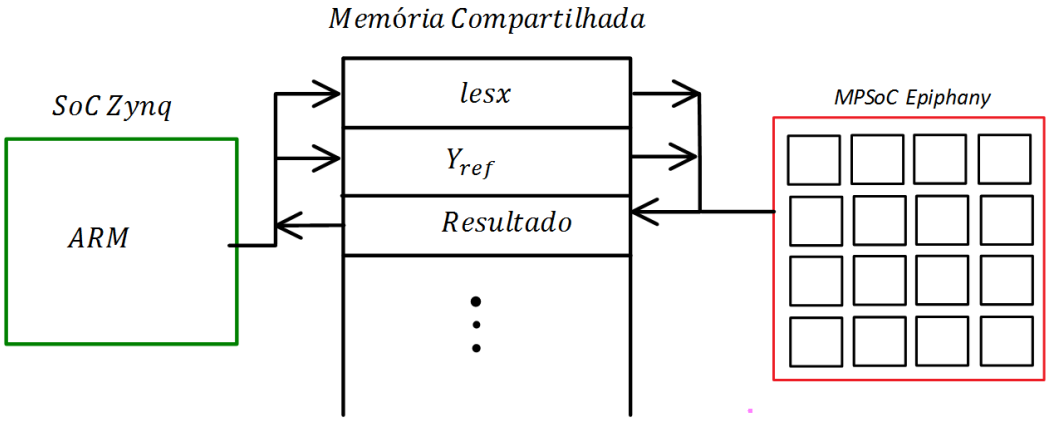

Figura 4.4. 6 Comunicação entre o SoC Zynq e o MPSoC Epiphany para cálculo das matrizes F e B_{ineq}.

Na etapa seguinte, 8 dos 12 *eCores* restantes assumem o cálculo do vetor $inter$ (linha (09) do Algoritmo 4.4.1 e operação 5 da Figura 4.1.2) e de sua função de maximização (parte da linha (11) do Algoritmo 4.1.1), já identificada como uma função de alto custo. Assim, todos os *eCores* utilizam-se do mesmo código, acessando partes distintas das matrizes sobre as quais irão operar. Cada *eCore* realiza parte do cálculo de uma única operação de matrizes, aproximando-se de um modelo SIMD. A Figura 4.4.7 ilustra essa etapa do processo.

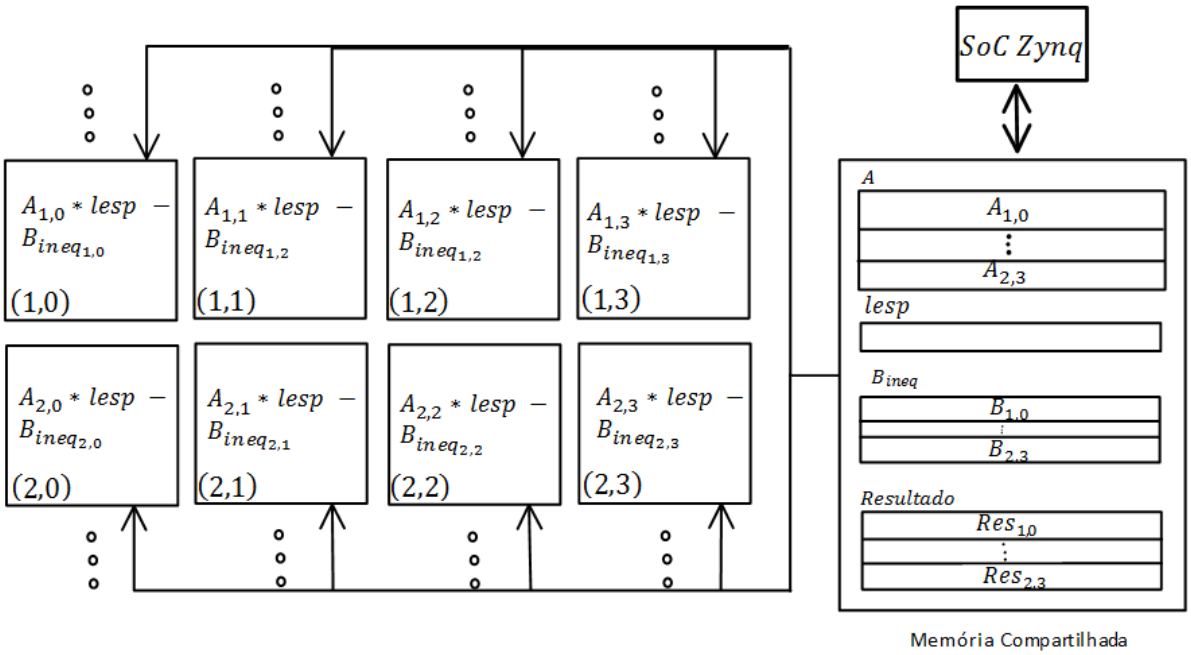

Figura 4.4. 7Configuração dos eCores Epiphany para cálculo do vetor inter e sua maximização.

Enquanto os *eCores* executam a etapa descrita pela Figura 4.4.7, o processador ARM calcula parte da expressão da matriz G, a saber, $G = H * lesp + F$ (linha (11) do Algoritmo 4.1.1). Ao fim dessa etapa ocorre uma nova sincronização global entre os núcleos de processamento.

Por fim, os 4 *eCores* restantes se encarregavam do cálculo dos vetores p_1 *e* p_2 (linhas (13) e (14) do Algoritmo 4.1.1), sendo descritas na Figura 4.4.8 as operações realizadas em cada um dos núcleos.

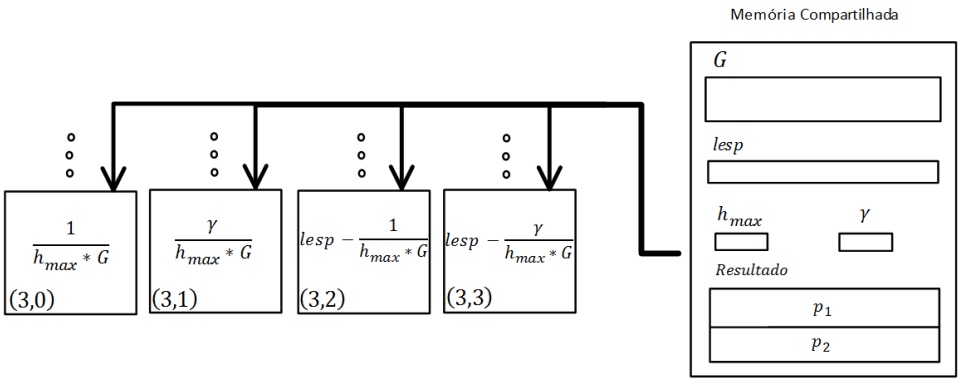

Figura 4.4. 8 Configuração dos eCores Epiphany para cálculo dos vetores p_1 e p_2.

Os *eCores (3,0) e (3,1)* realizam suas operações e escrevem o resultado na memória dos *eCores (3,2) e (3,3)* juntamente com um sinal de controle (*flag* na memória), para que esses últimos possam iniciar suas operações.

Terminado esta última etapa, os *eCores (1,0)* até *(1,3)* são chamados novamente para o cálculo de $inter_1$ (linha (15) do Algoritmo 4.1.1) e os *eCores (2,0)* até *(2,3)* para o cálculo de $inter_2$ (linha (16) do Algoritmo 4.1.1), procedendo de forma análoga ao cálculo de $inter$, citado acima. Em paralelo com essa etapa, o SoC ARM calcula parte da operação de J_1 ($J_1 = \frac{1}{2}p_1^T.H.p_1 + F^T.p_1$) *e* J_2 ($J_2 = \frac{1}{2}p_2^T.H.p_2 + F^T.p_2$) (linhas (17) e (18) do Algoritmo 4.1.1).

Vale lembrar que todo o restante do algoritmo foi executado no SoC ARM. A Figura 4.4.9 ilustra o fluxo de execução do algoritmo com destaque para as operações realizadas no coprocessador *Epiphany*.

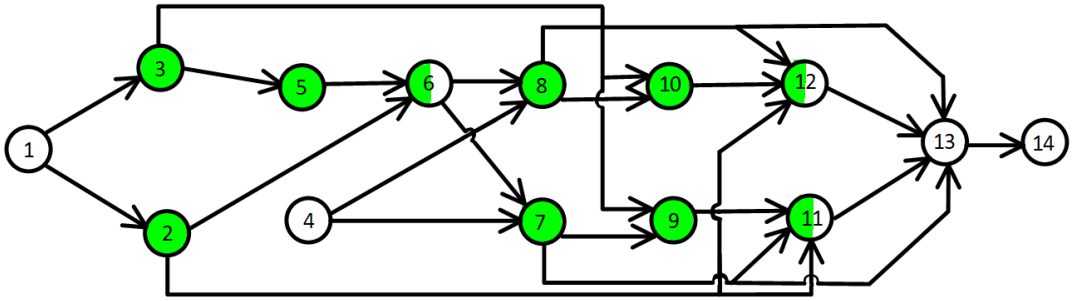

Figura 4.4. 9 Fluxo de execução do algoritmo MPC com destaque (em verde) para as operações executadas pelos eCores Epiphanys.

A estratégia em questão foi pensada na tentativa de diminuir o tamanho dos programas a serem carregados nos *eCores*, devido à limitação de memória local dos mesmos, ao mesmo tempo em que eram realizados cálculos em paralelo da maioria das expressões do algoritmo, ainda que esse paralelismo compreendesse poucos *eCores* simultaneamente.

Todavia, verificou-se como resultado da implementação que o tempo demandado pela execução da solução paralela se mostrou ligeiramente maior que o tempo gasto pela versão serial do programa.

Analisando o custo de cada parte da implementação paralela, notou-se que o ganho que se poderia obter com a paralelização das expressões foi menor que o tempo de *overhead* presente no custo de se carregar programas heterogêneos e na sincronização entre os *eCores* com seus vizinhos e dos mesmos com o processador ARM.

Neste caso, o número de chamadas ao coprocessador *Epiphany* foi contribuinte principal para o elevado *overhead*, justificado pelo custo de escrita e leitura da memória compartilhada para a sincronização de dados entre o processador ARM e o coprocessador *Epiphany*.

4.4.4 Configuração Final do Algoritmo Paralelo

As estratégias até aqui descritas e suas variações (tendo em conta a abordagem experimental adotada) deram base para que se chegasse à configuração final do algoritmo paralelo. Por meio dos resultados das implementações citadas acima se verificou que, devido às características do

algoritmo MPC já apresentadas, os melhores desempenhos eram obtidos quando se focava na paralelização das expressões do *loop* interno do mesmo, como já era de se esperar pela análise de custo realizada (*profiling*).

Observou-se ainda (vide Capítulo 3 e implementações realizadas) que carregar os *eCores* com programas diferentes durante a parte *off-line* do algoritmo, levava a um alto custo adicional. Desse modo, implementações que utilizavam o mesmo programa para todos os *eCores*, sem cargas durante a execução do algoritmo, obtinham maior desempenho.

No entanto, há uma preocupação com o tamanho do programa a ser carregado nos *eCores*, devido à limitação da memória local, o que dificultava fazer códigos generalistas (para ser carregados em todos os núcleos), que fossem capazes de realizar um número elevado de operações, dando flexibilidade de execução por parte do coprocessador Epiphany.

Assim, utilizar todos os 16 eCores em uma mesma operação levava a uma maior velocidade de execução das operações e, consequentemente, em um maior desempenho do algoritmo. Além disso, outro ponto importante a ser explorado diz respeito à utilização do SoC ARM para a realização de partes independentes de operações enquanto espera pela finalização do MPSoC *Epiphany*.

Por fim, também foi observado que o custo de escrita e leitura de dados na memória compartilhada, tanto por parte do ARM quanto por parte do *Epiphany*, apresentava um alto *overhead*, principalmente quando tais operações aconteciam dentro do *loop* interno, devido à quantidade de vezes em que o mesmo era executado. Nesse sentido, uma boa medida a se tomar seria minimizar a quantidade de dados trocados entre as partes supracitadas, diminuindo o custo adicional de comunicação.

Mediante ao exposto acima, adotou-se a estratégia de se focar nas expressões do *loop* interno, de maior custo computacional, paralelizando-as no coprocessador *Epiphany*. A estratégia completa foi dividida em X passos descritos a seguir.

a) Passo 0 (*off-line*):

Na etapa *off-line* do algoritmo são calculadas as matrizes de custo (F) e restrições (A_{ineq} *e* B_{ineq}) utilizadas nos cálculos da etapa *online*, de interesse na paralelização. Uma vez que a matriz A_{ineq} (representada como A na parte *online* do algoritmo) teve seu cálculo realizado neste passo, não sofrendo posterior alteração, o processador ARM a escreve na memória compartilhada antes que se inicie a parte *online* do algoritmo. Além disso, são carregados os programas executáveis nas memórias internas de cada um dos *eCores Epiphany*.

Após receberem o programa a ser executado, os *eCores* iniciam sua configuração previa, antes de entrarem em seus laços de cálculos. Nesta etapa são definidos os tamanhos das sub-matrizes a serem computadas correspondentes a cada *eCore*, os parâmetros de configuração da DMA (utilizado para leitura e escrita de dados entre a memória local dos *eCores* e a memória compartilhada) e os ponteiros que apontam para a memória compartilhada e *mutex lock*.

Como parte da implementação, é definido um *eCore* mestre, encarregado de receber e enviar os sinais de sincronização de operação com o SoC ARM. Assim, o *eCore* definido como mestre (*eCore (0,0)*) envia um sinal de confirmação para o ARM comunicando que o MPSoC *Epiphany* se encontra pronto para inicializar os cálculos. Enquanto isso, o restante dos *eCores* entram em um estado de espera até que o *eCore* mestre receba o sinal do ARM que autorize o inicio da operação.

b) Passo 1:

Uma vez realizadas as operações correspondentes a parte *off-line*, entra-se então na parte *online* de interesse para a paralelização. Nesta etapa o ARM prossegue calculando as matrizes F e B_{ineq} no *loop* externo, copiando esta última para a memória compartilhada, e só então entrando a executar o *loop* interno (parte em que as operações serão executadas tanto pelo ARM quanto pelo *Epiphany*). Isso faz com que o custo de escrita à memória compartilhada (por parte do ARM) e leitura (por parte do Epiphany) sejam minimizados. A Figura 4.4.10 ilustra as operações até aqui descritas.

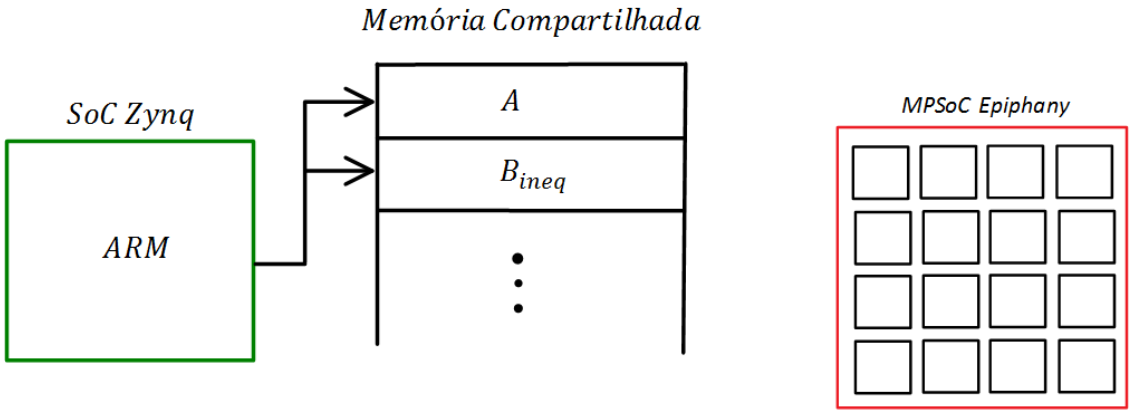

Figura 4.4. 10 Comunicação entre SoC Zynq e MPSoC Epiphany antes do início da parte online do algoritmo.

c) Passo 2:

Já na execução do *loop* interno, a primeira chamada ao MPSoC *Epiphany* é realizada utilizando os 16 *eCores* em conjunto, e carregados com o mesmo programa, a fim de

executar o cálculo do vetor $inter$ (linha (9) do Algoritmo 4.1.1) e parte da operação realizada em G $(2 * rho * A^T * \max\_matrix(inter, 0))$, referente a linha (11) do Algoritmo 4.1.1. A Figura 4.4.11, ilustra essa etapa de execução por parte dos *eCores* do *Epiphany*.

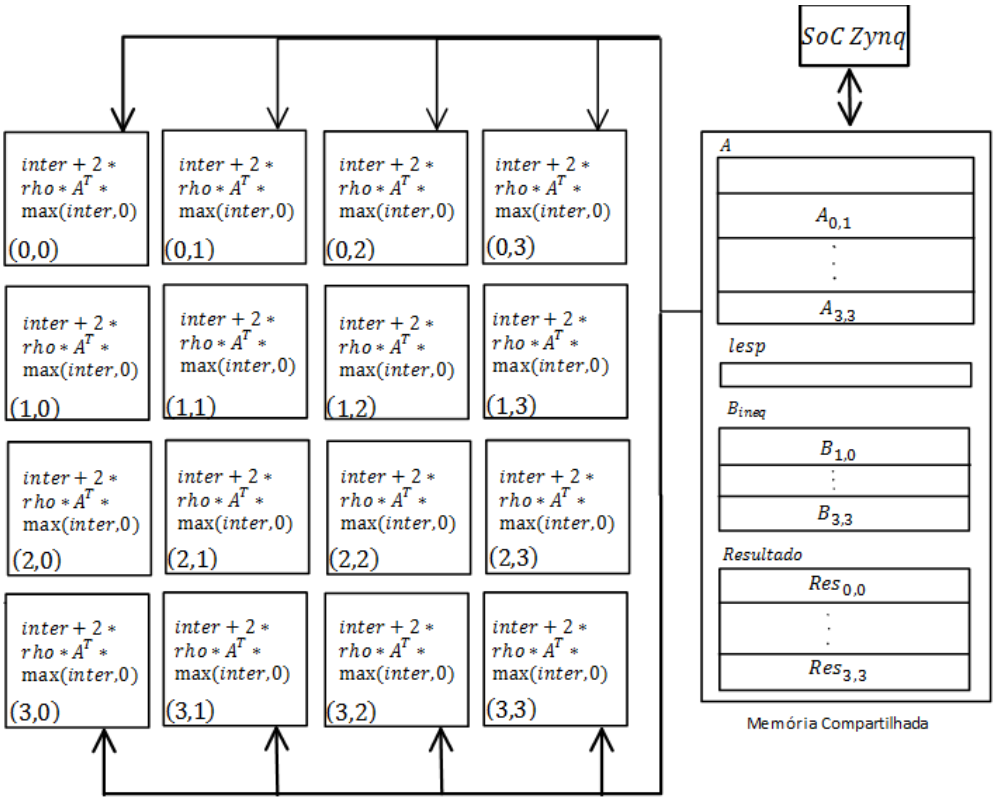

Figura 4.4. 11Configuração dos eCores Epiphany para cálculo do vetor inter e sua maximização.

Para isso, o SoC ARM copia o vetor $lesp$ na memória compartilhada e envia um sinal (*flag* definida na memória compartilhada) para que os *eCores Epiphany*, cujos programas já foram carregados *off-line*, iniciem a sua execução. Um detalhe a ser observado é que o sinal de autorização de início enviado pelo ARM corresponde ao valor atual da variável rho (definida previamente e atualizada na linha (5) do Algoritmo 4.1.1), que representa a penalidade devido às restrições, e que será utilizado para os cálculos a serem realizados pelos *eCores*.

Uma vez autorizado pelo ARM, os *eCores* iniciam sua execução de maneira paralela, acessando cada um uma região especifica da memória compartilhada, seja para ler os trechos que lhes cabem das matrizes sobre as quais se operará ou para escrever seus resultados. Vale ressaltar que cada *eCore* possui um identificador que vai de 0 a 15, utilizando-se do mesmo para realizar os cálculos de incremento dos ponteiros relativos às matrizes na memória compartilhada que serão copiadas para sua memória local.

Após a leitura dos dados que lhes competem, os eCores iniciam a computação dos mesmos escrevendo o resultado final em uma posição específica da memória compartilhada, também calculado sobre seu ID, que levará ao resultado completo esperado pelo ARM.

Por fim, na medida em que cada *eCore* vai terminando suas operações, entra-se em um novo estado de espera (*barrier*) até que todos os *eCores* terminem e um sinal de finalização seja enviado pelo *eCore* mestre, voltando ao inicio do *loop* e esperando uma nova rodada de cálculos.

Enquanto os *eCores* executam a etapa descrita pela Figura 4.4.11, o processador ARM calcula parte da expressão da matriz G, a saber, $G = H * lesp + F$ (linha (11) do Algoritmo 4.1.1). Ao fim dessa etapa ocorre uma sincronização global entre os núcleos de processamento e finalização do cálculo da matriz G, realizado pelo ARM.

As verificações à *flag* de sinalização de início e término de tarefa, utilizada para a sincronização entre ARM e *Epiphany*, são realizadas por acessos recorrentes à mesma (*polling*). A Figura 4.4.12 ilustra a comunicação entre SoC ARM e MPSoC *Epiphany* relativa a este primeira etapa de execução do algoritmo. A numeração nas setas indicam a ordem de em que as ações são executadas.

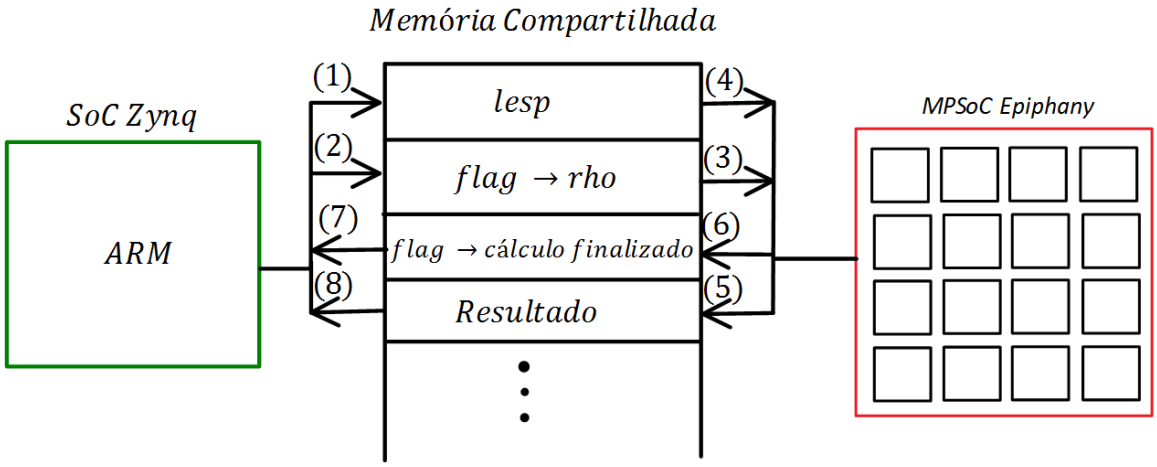

Figura 4.4. 12 Comunicação entre Zynq e Epiphany relativa ao passo 2 da estratégia final de paralelização adota. A numeração nas setas indicam a ordem de em que as ações são executadas.

Assim, o ARM lê da memória compartilhada o resultado do cálculo da expressão $Resultado = 2 * rho * A^T * \max\_matrix(inter, 0)$, realizado pelo *Epiphany* e correspondendo a segunda parte do cálculo de G.

d) Passo 3:

Uma vez que o cálculo dos vetores $inter_1 e\ inter_2$ (linhas (15) e (16) do Algoritmo 4.1.1) e suas maximizações são semelhantes ao cálculo do vetor $inter$, mudando apenas o vetor de variáveis de decisão p, aproveitou-se o mesmo código utilizado para cálculo desta, a fim de se calcular aquelas, a menos de alguns sinais de controle descritos posteriormente.

Desse modo, o ARM efetua todo o cálculo da matriz p_1, escrevendo-a na memória compartilhada e enviando um novo sinal de autorização de inicio para o *Epiphany*, que se

encontrava em estado de espera após a realização dos cálculos descritos no passo 2. Vale ressaltar que nesta etapa o sinal enviado é igual a 1.0, indicando aos *eCores* que se trata do cálculo de p_1.

Semelhantemente ao que ocorre na primeira chamada ao *Epiphany* (Passo 2), o ARM utiliza-se do tempo gasto até que os *eCores* concluam suas operações para calcular o vetor p_2, que será utilizado no passo seguinte (Passo 4). Além disso, neste tempo de espera também é calculado no ARM parte da expressão de J_1, a saber, $J_1 = \frac{1}{2}p_1^T.H.p_1 + F^T.p_1$. A Figura 4.4.13 ilustra a comunicação entre ARM e *Epiphany* referentes ao passo descrito acima. A numeração nas setas indicam a ordem de em que as ações são executadas.

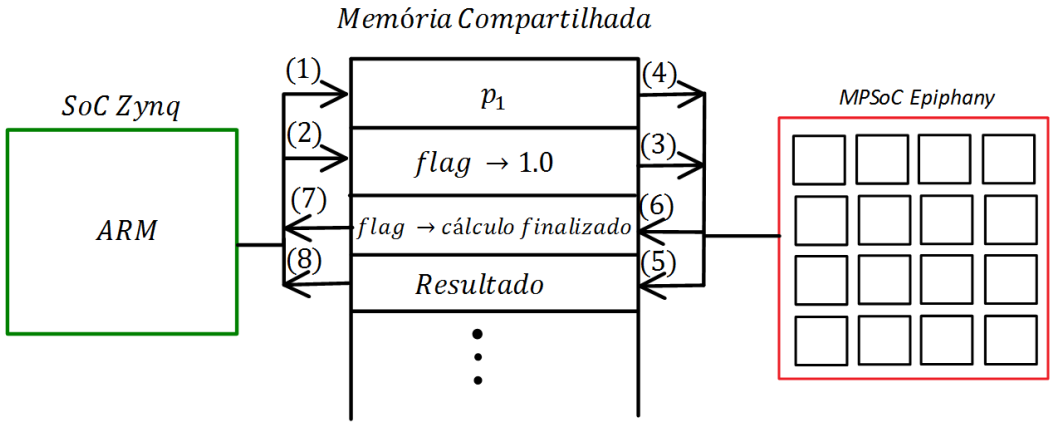

Figura 4.4. 13 Comunicação entre Zynq e Epiphany relativa ao passo 3 da estratégia final de paralelização adota. A numeração nas setas indicam a ordem de em que as ações são executadas.

Terminado o cálculo por parte dos *eCores*, o SoC ARM acessa a memória compartilhada para leitura da matriz $inter_1$ já maximizada.

e) Passo 4:

Neste último passo o SoC ARM realiza a escrita do vetor p_2 na memória compartilhada, calculado no passo anterior, além de um novo sinal de autorização de inicio ao *Epiphany*. Semelhante ao passo anterior, o ARM aproveita-se do tempo de espera de conclusão dos cálculos por parte dos *eCores Epiphany* para calcular a expressão parcial de J_2, a saber, $J_2 = \frac{1}{2}p_2^T.H.p_2 + F^T.p_2$ (linha (18) do Algoritmo 4.1.1), e finalizar o cálculo de J_1 (linha (17) do Algoritmo 4.1.1).

Por fim, após o término do cálculo da matriz $inter_2$ realizado pelos *eCores*, o ARM lê tal resultado da memória compartilhada finalizando o cálculo de J_2, e prosseguindo com a execução do algoritmo. A Figura 4.4.14 lustra o fluxo de execução do algoritmo MPC com destaque em verde para as operações executadas pelos *eCores Epiphany*.

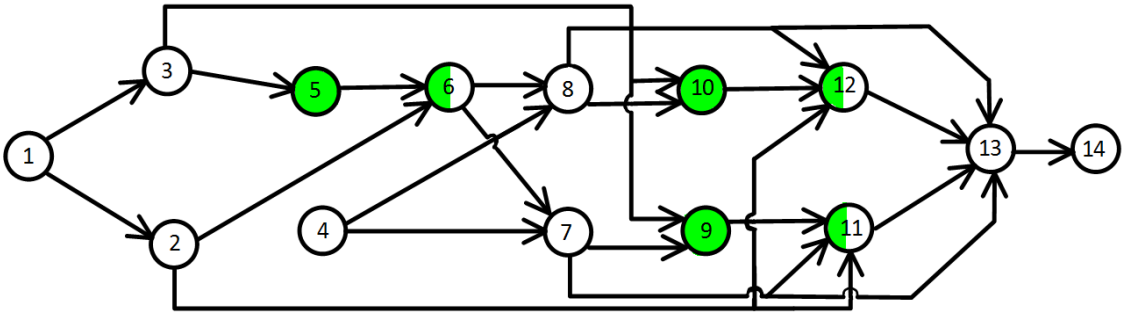

Figura 4.4. 14 Fluxo de execução do algoritmo MPC com destaque para as operações realizadas pelos eCores Epiphany, representadas em verde.

A estratégia descrita acima foi a de melhor desempenho na avaliação realizada com a aplicação do integrador triplo, estendendo-se a aplicação da mesma ao problema do controle de atitude da plataforma de satélites, cujos resultados são ilustrados no Capítulo 5.

4.5 Conclusões do Capítulo

Com base nas subseções anteriores, verifica-se que o caminho até a configuração final do algoritmo MPC paralelo a ser embarcado passou por diversas estratégias intermediárias que, embora não alcançassem o desempenho esperado, contribuíram para que se chegasse a estratégia final utilizada. Percebe-se ainda que muitas outras abordagens poderiam ser exploradas visando o aumento do desempenho obtido com a paralelização, mostrando que não existe uma forma canônica e única de implementação.

Um fator que restringiu e dificultou as decisões tomadas foi a alta dependência entre as operações do algoritmo MPC proposto, levando a muitos pontos de serialização (como acesso a memória compartilhada).

Vale lembrar que toda esta etapa está a cargo do desenvolvedor, assim, cabe ao mesmo pensar em qual o melhor arranjo possível para que se evite *idle time* por parte dos processadores, promovendo um balanceamento de carga, e que ao mesmo tempo minimize a troca de dados entre os mesmos, evitando o alto custo de overhead envolvendo tal operação, demonstrado pelos *benchmarks*.

Para tanto, optou-se por manter o fluxo principal do algoritmo no processador ARM, fazendo chamadas pontuais ao coprocessador *Epiphany*, para se calcular as funções de maior custo computacional. Além disso, enquanto espera a resposta dos *eCores*, o processador ARM continua a execução de outras operações, ou parte delas, diminuindo seu *idle time*.

Devido a dependência de dados já mencionada, observou-se uma dificuldade na divisão do algoritmo em tarefas independentes. Assim, para aumentar o ganho obtido com a paralelização, optou-se por utilizar uma fragmentação de operações matriciais, situação em que uma única

expressão é paralelizada para que o cálculo final seja segmentado em partes menores calculadas em paralelo.

Todos os *eCores* foram carregados com o mesmo algoritmo, ainda no modo *offline*, evitando o custo referente a tal carga e diminuendo o *overhead* de sincronização entre os núcleos de processamentos. Tal algoritmo tem o fluxo de execução direcionado por *flags* provenientes do *SoC Zynq*.

Capítulo 5

Resultados

Até aqui foram apresentados os métodos utilizados para se chegar a uma configuração final de um algoritmo paralelizável a ser embarcado na arquitetura de *hardware* da placa *Parallella.* Assim, o escopo deste trabalho se restringe a uma avaliação de tal placa utilizando-se de aplicações MPC como um estudo de caso, em uma abordagem experimental. As seções seguintes trarão os resultados obtidos bem como uma análise acerca dos mesmos.

5.1 Pontos de Implementação Comuns a Todas as Aplicações

A implementação da estratégia final do algoritmo paralelizável foi concebida conforme discutido nos Capítulo 4 e comparada, em termos de tempo de execução, com a versão serial do algoritmo executando somente no processador ARM A9 *Dual Core* (do SoC).

Vale ressaltar que tanto para a versão serial quanto para a versão paralela do algoritmo MPC foram utilizadas diretivas de otimização suportadas pelo compilador GCC que maximizassem o desempenho em termos de custo computacional. Além disso, foram utilizados dados do tipo *float* e habilitado o uso da unidade de ponto flutuante presente no processador ARM.

Como medida quantitativa da qualidade da resposta do algoritmo, foi utilizado o *Erro Médio Quadrático* (EMQ) das saídas do controlador quando comparadas aos sinais de referência. Por fim, foram calculados o *speedup* de cada uma das implementações; lembrando que para o cálculo desta última foram considerados 17 núcleos de processamento (ARM + 16 eCores), uma vez que os mesmos trabalham em paralelo.

Por fim, em cada iteração são calculadas as mesmas operações. Assim, o algoritmo possui um tempo de execução regular, demandando o mesmo tempo para se calcular cada uma das iterações.

5.2 Resultado da Aplicação do Integrador Triplo

O integrador triplo é uma aplicação simples baseada em [38] e cujas características de seu modelo foram apresentadas no Capítulo 4. Em resumo, tratam-se de três variáveis de estado e

uma saída seguindo um sinal de referência dado por uma onda quadrada de amplitude 1, período 10 s e ciclo de trabalho de 0.5. As restrições adotadas para os valores da variável de estado x_1 e sua derivada $\dot{x_1}$, juntamente com as restrições sobre o sinal de controle u e sua variação δ, foram baseadas em [38] e estão expressas na Tabela 5.2.1.

Tabela 5.2. 1 Restrições impostas sobre o algoritmo de controle do integrador triplo.

Parêmetros	Valores
x_1	[-∞, ∞]
$\dot{x_1}$	[-2,2]
u	[-30,30]
δ	[-50,50]

A versão serial do programa executada no processador ARM levou um tempo de execução de:

$$T_{serial} = 3.88\ segundos$$

Considerando que em cada iteração são calculadas as mesmas operações, e que o algoritmo possui um tempo de execução regular (demandando o mesmo tempo para se calcular cada uma das iterações), conclui-se que foram gastos um tempo de 19,4 ms por período de amostragem.

Já a versão paralelizada levou um tempo de execução por período de amostragem de 8,75 ms e um tempo total de:

$$T_{paralelo} = 1{,}75\ segundos$$

Desse modo, o *speedup* alcançado com a paralelização foi de aproximadamente:

$$speedup = 2{,}2$$

Assim, a paralelização do problema possibilitou um ganho em termos de tempo de execução próximo de 54,9%.

Os gráficos representados nas Figuras 5.2.1 até 5.2.4 ilustram a saída obtida pelo controlador comparado ao sinal de referência, a derivada do sinal de saída x_1, o vetor de sinais de controle u e a variação deste. Neste caso, o EMQ obtido foi de 0,027.

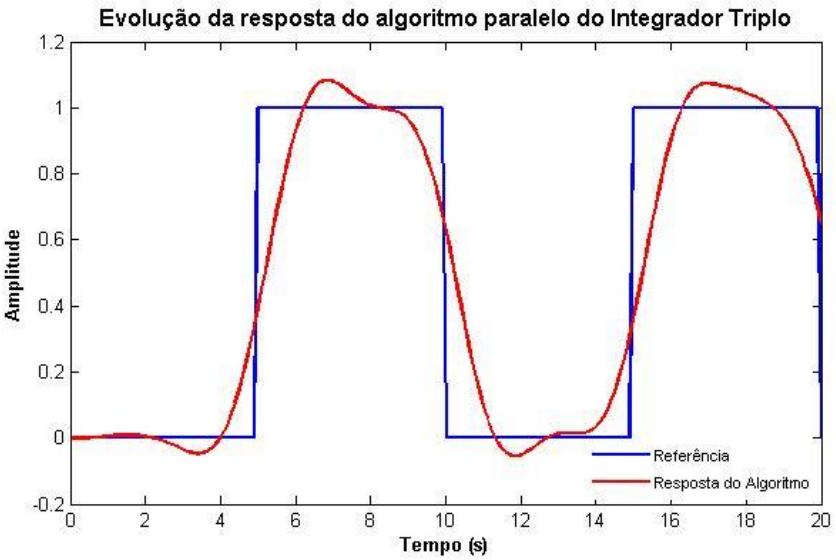

Figura 5.2. 1 Gráfico da saída do controlador MPC comparado com o sinal de referência.

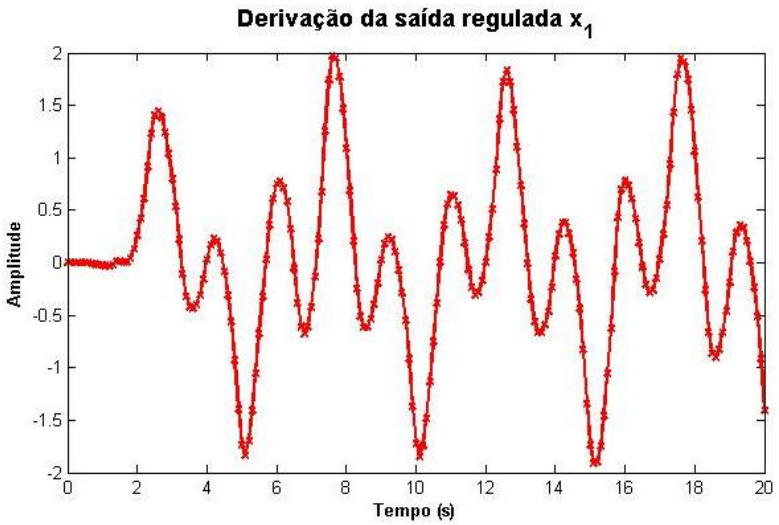

Figura 5.2. 2 Gráfico da saída $\dot{x}_1$da aplicação do integrador triplo.

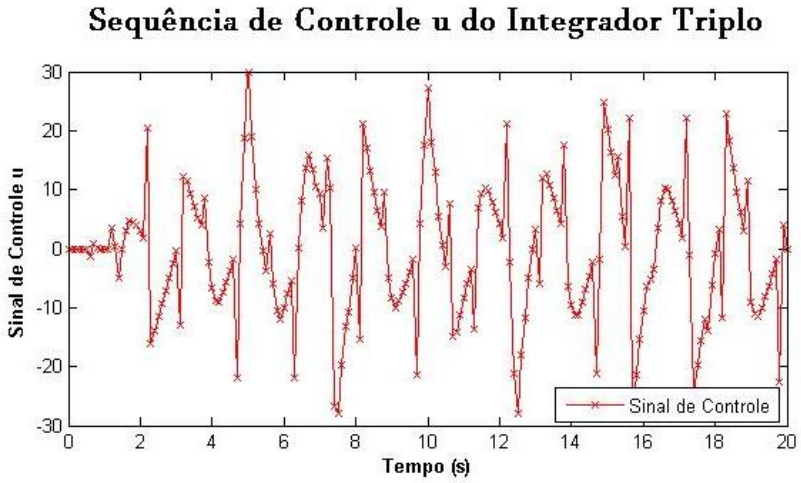

Figura 5.2. 3 Gráfico do sinal de controle u

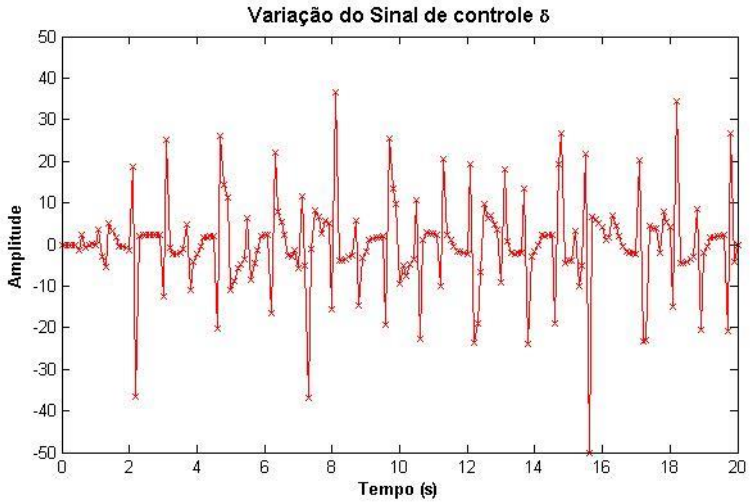

Figura 5.2. 4 Gráfico da variação δ do sinal de controle u.

Para efeitos de comparação entre o desempenho encontrado com a paralelização e as dimensões dos problemas, foram realizados testes adicionais com a aplicação de *benchmark* do integrador triplo variando o horizonte de predição. Os resultados obtidos estão expressos na Tabela 5.2.2.

- Para um horizonte de predição igual a $N = 40$
  - $lesp = 40x100$
  - $A_{ineq} = 240x40$
  - $B_{ineq} = 240x1$
  - $inter, inter_1 e\ inter_2 = 240x1$
  - $p_1, p_2\ e\ G = 40x1$

- Para um horizonte de predição igual a $N = 60$

- $lesp = 60x100$
- $A_{ineq} = 360x60$
- $B_{ineq} = 360x1$
- $inter, inter_1 e\ inter_2 = 360x1$
- $p_1, p_2\ e\ G = 60x1$

➢ Para um horizonte de predição igual a $N = 80$

- $lesp = 80x100$
- $A_{ineq} = 480x80$
- $B_{ineq} = 480x1$
- $inter, inter_1 e\ inter_2 = 480x1$
- $p_1, p_2\ e\ G = 80x1$

Tabela 5.2. 2 Variação do horizonte de predição na aplicação do integrador triplo.

Horizonte de Predição	T_{serial} (s)	$T_{paralelo}$(s)	*Speedup*
20	3,88	1,75	2,2
40	12,05	4,07	2,96
60	35,77	9,8	3,65
80	103,22	26	3,97

Como pode ser observado pelos dados acima apresentados e pelo gráfico da Figura 5.2.3, a relação entre *speedup* e as dimensões do problema não obedece a um comportamento linear.

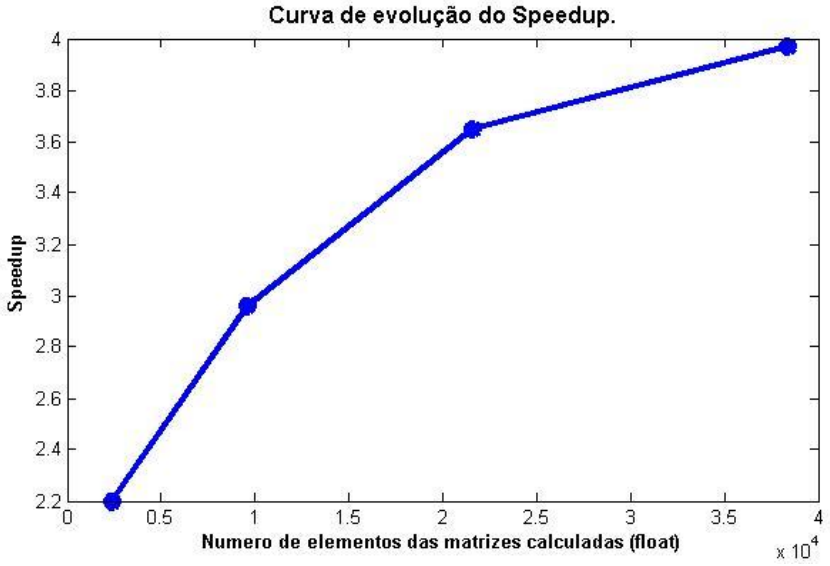

Figura 5.2. 5 Curva de evolução do Speedup em relação as dimensões das matrizes calculadas no problema.

5.3 Resultado do Sistema de Controle de Atitude de Satélites Artificiais

O controle de atitude de satélites artificiais é uma aplicação real desenvolvida por [75] e cujas características de seu modelo são apresentadas no Capítulo 5. Em resumo, trata-se de seis variáveis de estado e três saídas de interesse, conforme mostrado no gráfico da Figura 5.3.1. O ponto operacional escolhido para o cálculo da matriz linear corresponde a pequenas variações de ângulo, velocidade angular e aceleração. Assim, as restrições adotadas para os valores de θ_1, θ_2 e θ_3, juntamente com as restrições sobre o sinal de controle u e sua variação δ estão expressas na Tabela 5.3.1.

Tabela 5.3. 1 Restrições impostas sobre o algoritmo de controle de atitude de satélites artificiais.

Parâmetro	Valor
$\theta_1(rad)$	$[-\frac{10\pi}{180}, \frac{10\pi}{180}]$
$\theta_2(rad)$	$[-\frac{10\pi}{180}, \frac{10\pi}{180}]$
$\theta_3(rad)$	$[-\frac{10\pi}{180}, \frac{10\pi}{180}]$
$u(\frac{rad}{s^2})$	[-1.5,1.5]
$\delta(\frac{rad}{s^3})$	[-50,50]

A versão serial do programa executada no processador ARM levou um tempo de execução de:

$$T_{serial} = 19.75\ segundos$$

Levando em conta as mesmas considerações feitas para o caso do integrador triplo (seção anterior), conclui-se que o algoritmo demandou um tempo de $98{,}75\,ms$ por período de amostragem. Já a versão paralelizada levou um tempo de execução por período de amostragem de $29{,}55\,ms$ e um tempo total de:

$$T_{paralelo} = 5{,}91\,segundos$$

Desse modo, o *speedup* alcançado com a paralelização foi de aproximadamente:

$$speedup = 3{,}34$$

Assim, a paralelização do problema possibilitou um ganho em termos de tempo de execução próximo de 70,08%.

O gráfico representado na Figura 5.3.1 ilustra as respostas dos ângulos de Euler obtidas mediante as entradas iguais a $\theta_1 = 5°, \theta_2 = 2°\ e\ \theta_3 = -3°$. Vale ressaltar que as entradas em questão correspondem a entradas degrau filtradas, como discutido em [75]. Para este caso, o EMQ obtido foi de 0,02974.

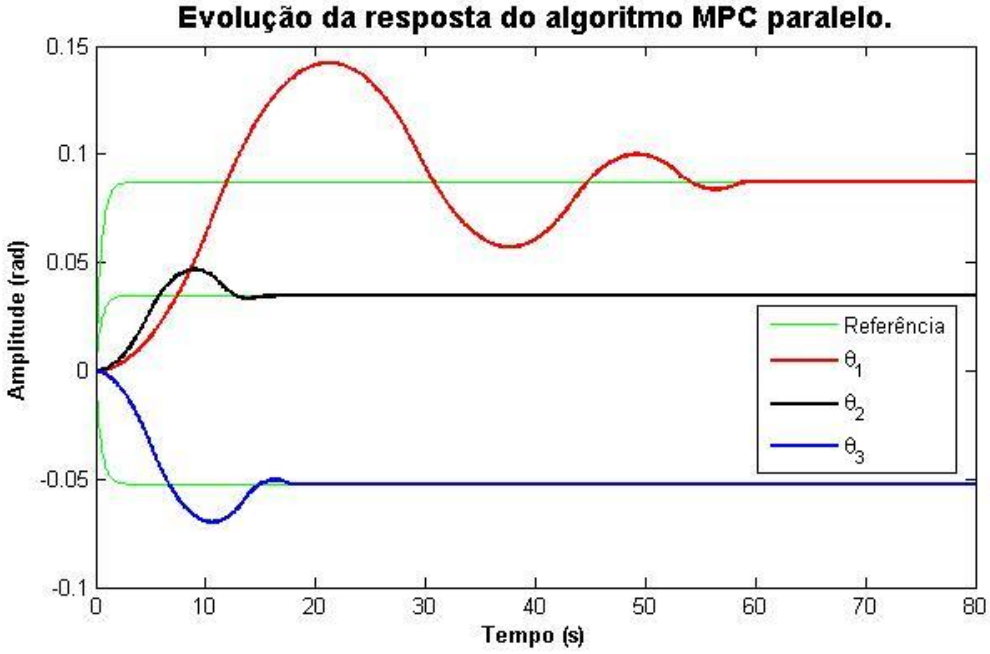

Figura 5.3. 1 Gráfico da resposta do controlador MPC de controle de atitude de satélites artificiais.

Pelo gráfico representado pela Figura 5.3.1 percebe-se que o tempo de simulação de 20 segundos não é o suficiente para que a resposta referente a θ_1 atinja o tempo de assentamento. Notou-se que o mesmo ocorria próximo a 60 s, ao passo que nas simulações realizadas em [75] apontam este último como sendo de 36 segundos.

A evolução dos sinais de controle das acelerações Ω_1, Ω_2 *e* Ω_3 podem ser vistas no gráfico da Figura 5.3.3.

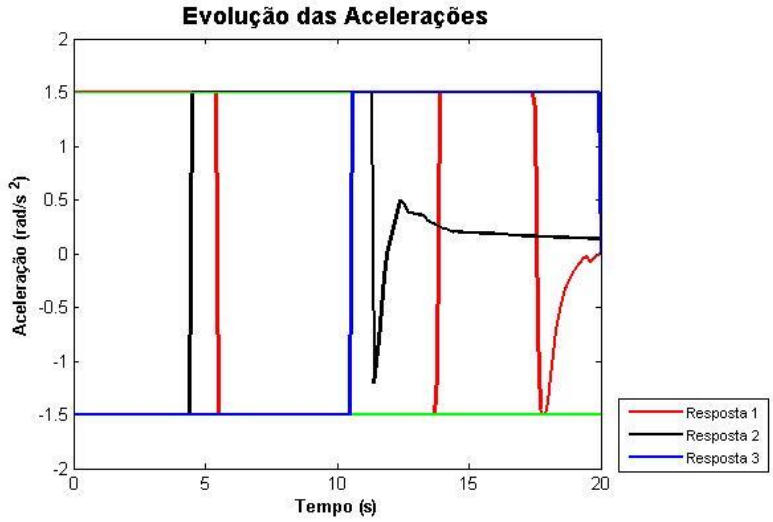

Figura 5.3. 2 Evolução das acelerações Ω_1(Resposta 1), Ω_2(Resposta 2) e Ω_3 (Resposta 3) pelo tempo.

5.4 Discussão dos Resultados

Nas seções anteriores foram apresentados os resultados referentes ao algoritmo paralelo MPC embarcado na plataforma *Parallella,* tendo em conta uma abordagem experimental. Dessa forma, a estratégia de paralelismo desenvolvida sobre a aplicação do integrador triplo foi validada para o problema de controle de atitude da plataforma de satélites.

O primeiro ponto a se observar é que quanto menor forem as dimensões do problema menor será o ganho de desempenho obtido com a paralelização do mesmo. Isso porque a redução de tempo de processamento obtida com a versão paralela do algoritmo é compensada em boa parte pelo tempo de *overhead* devido às comunicações e sincronias entre os núcleos de processamento. O gráfico da Figura 5.2.3 ilustra a afirmação sobredita, assim como o aumento do *speedup* obtido com a aplicação do controle da plataforma de satélites, de maior dimensão.

Em ambas as aplicações utilizadas como um estudo de caso, observou-se uma melhora significativa com a versão paralela do algoritmo frente a sua versão serial, o que poderia permitir restrições ainda maiores de *hardware*, acarretando em uma redução de custos de projeto. Além disso, uma vez que se tenha certa folga de execução do algoritmo (com relação ao tempo de simulação), pode-se pensar ainda em uma implementação adicional de modo a buscar melhor desempenho e robustez do sistema embaraçado.

Em alguns casos, como na aplicação do integrador triplo para um horizonte de predição de $N = 60$, verificou-se que o tempo de execução da versão serial do algoritmo se mostrou maior

que o tempo de simulação (20 s), indicando sua não aplicabilidade. No entanto, o mesmo não ocorre quando se aplica a estratégia de paralelização.

No Capítulo 4 foi discutido que é comum encontrar aplicações que não aproveitam de todo o poder computacional teórico esperado da arquitetura de *hardware*. Como exemplo de pesquisas desenvolvidas com a plataforma *Parallella*, foi citado o trabalho descrito em [69], em que a transformada rápida de Fourier em duas dimensões 2D-FFT (do inglês, *Fast Fourier Transform*) atingiu um valor máximo de apenas 13% do pico de desempenho teórico.

Em outros trabalhos que também fazem uso da *Parallella*, como em [84] (em que se implementou um acelerador em *software* para unidades de controle inteligentes de conversores de frequência utilizados em *smart grids)* e em [22] (com a implementação de um algoritmo de detecção de faces), foram alcançados *speedups* de 1,78 e 7,8 respectivamente. Tais valores, assim como o *speedup* alcançado no presente trabalho, ainda estão distantes dos valores encontrados com aceleração via *hardware* (principalmente com FPGA's). No entanto, é importante salientar que o presente trabalho e as pesquisas supracitadas tratam de uma nova abordagem, testando uma aceleração por *software*, mais prática e de menor tempo de desenvolvimento.

Neste contexto, FPGAs podem alcançar *speedups* maiores pelo mapeamento dos algoritmos diretamente em *hardware*, eliminando assim as restrições inerentes ao gargalo de *von Neumann* [85]. Entretanto, FPGAs perdem no contexto da programabilidade, tendo em conta que as mesmas requerem engenheiros altamente especializados no projeto de *hardware*, o que dificulta que cientistas da computação ou engenheiros, sem uma formação especializada, possam usufruir das suas vantagens. Este problema é referente uma lacuna entre a formação de cientistas da computação e engenheiros de *hardware*, mais conhecido como *education-wall* [85].

Por fim, os ganhos de desempenho observados nos resultados acima ficam ainda mais interessantes quando se pensa na forte restrição de memória presente na plataforma utilizada, aliado ao baixo grau de paralelismo do algoritmo MPC embarcado.

Capítulo 6

Conclusões e Trabalhos Futuros

Na tentativa de manter o desempenho exponencial dos dispositivos de processamento e com inviabilidade de continuar a melhorar efetivamente o desempenho dos microprocessadores por encolhimento dos transistores (relacionado com a lei de Moore e o *power-wall*) e mais encapsulamento destes em chips com um único núcleo, têm surgido cada vez mais plataformas com múltiplos núcleos (*multicore*). Esta nova forma de se pensar em arquiteturas de *hardware* vem ganhando popularidade no meio de desenvolvimento e pesquisa com plataformas compactas, flexíveis e com propostas de eficiência tanto do ponto de vista de consumo quanto de tempo de processamento.

Valendo-se desse avanço tecnológico das plataformas de *hardware* e do uso da paralelização de algoritmos, aplicações que outrora se mostravam inviáveis de serem implementadas em sistemas embarcados, devido às restrições inerentes aos mesmos, podem vir a se tornar viáveis, permitindo o desenvolvimento de sistemas com algoritmos complexos e de alto custo computacional, como o caso do MPC.

Nas seções seguintes é apresentada uma visão geral da pesquisa desenvolvida sobre a plataforma *multicore Parallella* e a conclusão sobre o funcionamento da mesma quando utilizada para embarcar algoritmos de controle preditivo para aplicações diversas de engenharia. Por fim, são apresentados pontos em potencial para futuras pesquisas na área.

6.1 Discussões Gerais

Neste trabalho propôs-se uma metodologia experimental para se testar a plataforma *multicore Parallella* quando aplicada a problemas MPC lineares como um estudo de caso. Essa plataforma é altamente flexível e traz como vantagem a facilidade e a grande variedade de formas possíveis de implementação, além do baixo consumo energético e do custo reduzido. Utilizando-se de linguagens de programação populares como C/C++ e da aceleração via *software* por paralelismo, pode-se levantar uma gama enorme de estratégias para explorar o SoC Zynq e o coprocessador Epiphany com 16 *eCores* presentes na arquitetura.

A metodologia em questão passou por diversas etapas até que chegasse a implementação das aplicações MPC. Primeiramente, foi realizado um estudo acerca da placa *Parallella* e suas características, mais especificamente da arquitetura *Epiphany*, entendendo como a mesma funciona. Foram dados uma atenção especial a fatores como comunicação entre os *eCores* e do

processador ARM com os mesmos, além da capacidade de memória presente em cada núcleo de processamento. Essa etapa foi importante para que se entendesse o processo de implementação dos algoritmos na placa em questão e para que se pensasse em *benchmarks* que explorassem características de interesse das aplicações MPC a serem embarcadas.

Os *benchmarks* (vide Capítulo 3), exploraram característica como o tempo de processamento de operações matriciais (crucial para as aplicações MPC), além do custo inerente à inicialização dos *eCores*, notando que para alguns casos o tempo demandado para a comunicação e sincronização entre os processadores acabam sendo mais elevados que o ganho com o paralelismo das operações.

Os resultados adquiridos com os *benchmarks* (vide Capítulo 3), não só comprovaram o que já se esperava pela teoria e pela literatura relacionada como também forneceram parâmetros quantitativos adicionais e fundamentais para a tomada de decisão quanto à estratégia de paralelização do algoritmo MPC. Em outras palavras, os *benchmarks* possibilitaram um entendimento prático da plataforma *Parallella*, levantando métricas de desempenho de algumas características da arquitetura de *hardware* que teriam impacto direto no algoritmo MPC utilizado nas aplicações propostas.

Posteriormente a etapa de *benchmarks*, foram implementadas estratégias de paralelização do algoritmo MPC utilizado, mediante uma abordagem experimental. Esta é a tarefa central da pesquisa, uma vez que os resultados finais, desempenho das aplicações MPC embarcadas, estão diretamente ligados à estratégia de paralelização escolhida.

É importante ressaltar que ao se embarcar um algoritmo, é necessário a avaliação de inúmeros fatores como o *hardware* que esteja sendo utilizado e as características do próprio problema em questão. Dentro deste último, deve-se levar em conta fatores como a interdependência entre tarefas e o volume de dados trabalhados. Desse modo, dentro da abordagem experimental adotada, foram desenvolvidas inúmeras tentativas de paralelização até se chegar à configuração final do algoritmo paralelo.

A estratégia de paralelização escolhida levou em conta alguns gargalos identificados na plataforma *Parallella*, como o custo temporal da comunicação entre o processador ARM e o coprocessador *Epiphany*, que impactam tanto na troca de dados e sinais de controle quanto na inicialização dos *eCores*, além da forte limitação de memória por parte destes.

O desenvolvimento de tais estratégias foi realizado sobre uma aplicação simples MPC, (integrador triplo) e posteriormente validado em uma aplicação do controle de atitude de uma plataforma de satélites, sendo verificado que aplicações de maiores dimensões alcançaram maiores valores de *speedup*.

6.2 Conclusões

Apesar de se tratar de uma plataforma recente, a *Parallella* tem surgido como uma alternativa promissora à aplicações embarcadas, principalmente para àquelas que possuem entre suas características o paralelismo de tarefas. Esse fato pode ser observado pelo aumento de pesquisas em diversas áreas que tem surgido nos últimos anos fazendo uso dessa plataforma de *hardware* (vide subseção 2.1.8, Capítulo 2). No entanto, fatores como a limitação da memória interna dos *eCores* e o custo envolvido na comunicação entre o SoC Zynq e o MPSoC *Eiphany*, têm limitado o desempenho de aplicações embarcadas nessa arquitetura.

De uma forma geral, mediante os resultados obtidos e as discussões apresentada na seção anterior, conclui-se que a plataforma *Parallella* oferece grande potencial para o desenvolvimento de aplicações embarcadas via uso do paralelismo de tarefas em uma abordagem de aceleração via *software*. Isso porque, a arquitetura em questão se mostra extremamente flexível, possibilitando a implementação de inúmeras estratégias que se adaptem às características do problema que esteja sendo embarcado. Além disso, existe a facilidade de implementação encontrada pelo uso de ferramentas *OpenSource* e linguagens de programação de mais alto nível que linguagens de *hardware e assembly*, levando a um menor tempo de desenvolvimento.

No entanto, o *hardware* em questão conta com alguns pontos negativos que devem ser levados em conta pelo desenvolvedor. Os pontos mais críticos dizem respeito à limitação de memória interna dos *eCores* e o custo de comunicação Zynq/*Epiphany*, já comentado. Além disso, por se tratar de uma arquitetura nova, muitas das ferramentas de desenvolvimento disponíveis, como eSDK fornecida pelo fabricante, ainda estão em processo de evolução e consolidação. Prova disso é o grande número de trabalhos com foco na portabilidade de ferramentas já existentes para a plataforma em questão, como mostrado na subseção 2.1.8 do Capítulo 1.

Apesar de crescente, a comunidade de desenvolvedores da placa *Parallella* ainda é pequena, quando comparada ao encontrado em arquiteturas como *Raspbarry Pi*, fator esse que dificulta o suporte a problemas que por hora são encontrados pelos desenvolvedores.

Assim, além da influência dos fatores supracitados, o algoritmo MPC embarcado não apresentou melhor eficiência devida suas próprias características, com inúmeros pontos de serialização e alta dependência de dados, como já mencionado. Mesmo assim, para aplicações de maiores dimensões foi verificado um *speedup* crescente, como o caso do integrador triplo com horizonte de predição igual a 60, em que se obteve um algoritmo paralelo aproximadamente quatro vezes mais rápido que sua versão serial.

Os valores de *speedup* alcançados neste trabalho ainda estão aquém dos valores obtidos por uma abordagem de aceleração via *hardware* (principalmente com a utilização de FPGA's). Todavia, deve-se lembrar que o objetivo do trabalho é propor uma abordagem experimental

alternativa, com a aceleração via *software*, em geral mais fácil e de menor tempo de desenvolvimento. Além do mais, o trabalho trata de uma aplicação de natureza serial, devido ao interesse da mesma, vide Capítulo 1.

Desse modo, como discutido na seção 5.4 do Capítulo 5, os resultados obtidos passam a se tornar interessantes quando são levadas em conta as restrições de *hardware* já discutidas, juntamente com a característica não paralelizável do algoritmo MPC.

Por fim, o que se conclui é que a plataforma *Parallella* se apresenta como uma arquitetura em potencial, podendo ser vista como uma alternativa à aceleração de algoritmos em hardware. Melhorias futuras como a expansão do número de núcleos do MPSoC *Epiphany* e da memória local dos mesmos, como previsto pelos fundadores do projeto (vide Capítulo 1), poderão alavancar ainda mais o uso de tal arquitetura em aplicações embarcadas.

6.3 Trabalhos Futuros

Sabe-se que um fator predominante para o sucesso de implementações paralelas está na própria característica do algoritmo a ser paralelizado. O que se notou é que o algoritmo MPC utilizado possui um baixo fator de paralelização, como já discutido. Assim, uma alternativa para futuros trabalhos seria utilizar um algoritmo diferente para o cálculo da sequência de sinais de controle u, que corresponde a parte *online* do algoritmo.

Existem algumas alternativas já exploradas pela literatura que poderiam obter melhores desempenhos com a placa *Parallella*. Outra abordagem seria utilizar recursos que reduzissem a quantidade de cálculos realizados a cada iteração, como a técnica de parametrização descrita em [38].

São encontrados muitos trabalhos na literatura que tratam de aplicações NMPC (do inglês, *Nonlinear Model Predictive Controll*) acelerados em *hardware*, assim, estender a abordagem aqui apresentada para esta nova classe de problemas seria uma tarefa promissora, pois tratam-se de problemas mais complexos e potencialmente mais difíceis de se alcançar restrições exigidas, principalmente com respeito ao tempo de computação do algoritmo. Pode-se ainda utilizar os recursos da computação natural, como algoritmos bio-inspirados, para resolução da função custo não linear, uma vez que muitos deles apresentam facilidade de paralelização.

Por fim, pensando do ponto de vista do *hardware*, seria possível explorar outras plataformas como a própria *Parallella* de 64 núcleos, aumentando o poder de paralelização e consequentemente o ganho de desempenho da aplicação.

Referências

[1] Embedded Systems in Industrial Applications - Trends and Challenges. IEEE Industrial Electronics, Society SIES 2007. Disponível em <http://www.uninova.pt/sies2007/SIES2007 _RZ_plenary_Lisbon_v1.pdf>. Acesso em: 06 mar. 2017

[2] Epiphany SDK Reference – REV 5.13.09.10. Disponível em: < http://www.adapteva .com/docs/epiphany_sdk_ref.pdf> Acessado em: Maio, 2016.

[3] Ju¨rgen Teich. "Hardware/Software Codesign:The Past, the Present, and Predicting the Future", Março 2012, IEEE.

[4] A. Goyal. "Multi-Core Processors and Systems: State-of-the-Art and Study of Performance Increase", San Jose State University, 2014.

[5] S. Pasricha, N. Dutt. " On-Chip Communication Architectures - System on Chip Interconnect". Elsevier, 2008. P 541

[6] L. S. Indrusiak, P. Dziurzanski and A. K. Singh. "Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing", River Publishers, 2016. 178p.

[7] P. Hsiung, Y. Chen, C. Lin. "Multicore Hardware-Software Design and Verification Techniques", Bentham Science, 2011. 95p.

[8] Família de produtos Intel® Xeon Phi™. Disponível em < http://www.intel.com.br/content /www/br/pt/processors/xeon/xeon-phi-detail.html>. Acessado em: Março, 2017.

[9] Recore Systems Xentium DSP central to many-core demo at date 2011. Disponível em: < http://www.recoresystems.com/news/press-releases/article/recore-systems-xentiumR-dsp-central-to-many-core-demo-at-date2011/> Acessadoem : Março, 2017

[10] TILE64 - Processor: A 64-Core SoC with Mesh Interconnect – Disponível em: < http://ieeexplore.ieee.org/document/4523070/> Acessado em: Março, 2017.

[11] Parallella: A Supercomputer For Everyone. Disponível em < https://www.kickstarter. com/projects/adapteva/parallella-a-supercomputer-for-everyone > Acessado em: Março, 2017.

[12] Zynq-7000 All Programmable SoC Product Advantages. Disponível em: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Acessado em: Março, 2017

[13] A. Olofsson, R. Trogan, O. Raikhman, "A 25 GFLOPS/Watt Software Programmable Floating Point Accelerator", High Performance Embedded Computing Computing Conference 2010

[14] A. Olofsson, R. Trogan, O. Raikhman. "A 1024-core 70 GFLOP/W Floating Point Manycore Microprocessor", High Performance Embedded Computing Conference, Sept 2011

[15] Parallella – Research. Disponível em: https://www.parallella.org/publications/ Acessado em: Março, 2017.

[16] G. Rey, Eliecer. “Configuración y uso de Parallella-16, University de Malaga”, Technical Report 2017
[17] Matthews, SJ. “Teaching with Parallella: A First Look in an Undergraduate Parallel Computing Course“. Journal of Computing Sciences in Colleges, 31(3), pp. 18-27. 2016.
[18] A. Varghese, B. Edwards, G. Mitra, A. P. Rendell, “Programming the Adapteva Epiphany 64-core NoC Coprocessor” 14 pages, submitted to IJHPCA Journal special edition.
[19] J. Pallister, S. J. Hollis, and J. Bennett, “Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms“, The Computer Journal, 2013.
[20] Z. Ul-Abdin , M. Yang, “A Radar Signal Processing Case Study for Dataflow Programming of Manycores“, Journal of Signal Processing Systems, Nov 27. 2015
[21] A. Kurth, et al. “Mobile Ultrasound Imaging on Heterogeneous Multi-Core Platforms“, Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia 2016
[22] F. Gao, et al, “Optimized Parallel Implementation of Face Detection Based on Embedded Heterogeneous Many-Core Architecture“, International Journal of Pattern Recognition and Artificial Intelligence 2017
[23] M. Reichenbach, et al, ”Fast Heterogeneous Computing Architectures for Smart Antennas“, Journal of Systems Architecture, Nov 10, 2016
[24] G. Hegde Siddhartha, N. Ramasamy, V. Buddha, N.Kapre, “Evaluating Embedded FPGA Accelerators for Deep Learning Applications“, IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)2016.
[25] F. Gao, Z. Huang, S. Wang , X. Ji, “A Manycore Processor Based Multilayer Perceptron Feedforward Acceleration Framework for Embedded System“, Information Science and Control Engineering (ICISCE), 2016 3rd International Conference.
[26] D.R. Mendat, S. Chin, S. Furber, A. G. Andreou, “Neuromorphic sampling on the SpiNNaker and Parallella chip multiprocessors” 2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS).
[27] Olesen, et al, “GNSS Software Receiver for UAVs“, European Journal of Navigation 2016.
[28] Tortonesi, et al,, “Leveraging Internet of Things within the military network environment — Challenges and solutions“, Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum.
[29] S.J. Matthews, R. W. Blaine, A.F. Brantly, “Evaluating single board computer clusters for cyber operations“, Cyber Conflict (CyCon U.S.), International Conference.
[30] B.Pierce and G. Cheng, “Herbert: design and realization of a full-sized anthropomorphically correct humanoid robot“, Frontiers in Robotics and AI.
[31] Bartok R,”A fuzzy rule interpolation base algorithm implementation on different platforms”, Carpathian Control Conference (ICCC), 2015 16th International
[32] D. Mayne and J. Rawlings, “Model Predictive Control: Theory and Design Madison”, WI, USA: Nob Hill, 2009, p 711.
[33] A. A. Jerraya, W. Wolf. “ Multiprocessor Systems-on-Chips”. Elsevier Science, 2004,p 608.

[34] J. L. Jerez, P. J. Goulart , S. Richter. “Embedded Online Optimization for Model Predictive Control at Megahertz Rates”, IEEE Transactions on Automatic Control (Volume: 59, Issue: 12, Dec. 2014).
[35] P. Patrinos, A. Bemporad. “An Accelerated Dual Gradient-Projection Algorithm for Embedded Linear Model Predictive Control”, IEEE Transactions on Automatic Control (Volume: 59, Issue: 1, Jan. 2014).
[36] A. Domahidi. “Survey of Industrial Applications of Embedded Model Predictive Control”, European Control Conference Aalborg, Denmark, Junho 29, 2016.
[37] H. Ayala, R. Sampaio, D. Munz, C. Llanos, L. Coelho and R. Jacobi, “Nonlinear Constrained Model Predictive Control Hardware Implementation with Custom-precision Floating Point Operations,” 24th Mediterranean Conference on Control and Automation 2016.
[38] Alamir, M. “A Pragmatic Story of Model Predictive Control: A Self-Contained Algorithms and Case-Studies”. CNRS - Univesity of Grenoble, 2013.
[39] E. F. Camacho e C. Bordons. “Model Predictive Control”. Springer, 1999, p 294.
[40] L. Wang. “Model Predictive Control System Design and Implementation Using MATLAB”. Springer, 2009, p 403.

[41] Gonzales, R. G., 2009. Utilização dos Métodos SDRE e Filtro de Kalman para o Controle de Atitude de Simuladores de Satélites. Msc Thesis, Instituto de Pesquisa Espaciais (INPE).
[42] P. Patrinos, A. Bemporad. “Simple and Certifiable Quadratic Programming Algorithms for Embedded Linear Model Predictive Control”. Noordwijkerhout, The Netherlands, August 23, 2012.
[43] A. Suardi, S. Longo, E. C. Kerrigan. “Energy-aware MPC co-design for DC-DC converters”. Control Conference (ECC), 2013 European.
[44] J. Mercieca and S. G. Fabri. “Particle Swarm Optimization for Nonlinear Model Predictive Control”, ADVCOMP 2011 : The Fifth International Conference on Advanced Engineering Computing and Applications in Sciences.
[45] J. Mercieca and S. G. Fabri. “A Metaheuristic Particle Swarm Optimization Approach to Nonlinear Model Predictive Control”. International Journal on Advances in Intelligent Systems, vol 5 no 3 & 4, 2012.
[46] S.Sivananaithaperumal, et al. “Particle Swarm Optimization Algorithm based Nonlinear Model Predictive Control”. Proceedings of the International Conference on Advances in Control and Optimizationof Dynamical Systems (ACODS2007), 2007.
[47] G. H. Negri, et al. “Differential Evolution Optimization Applied in Multivariable Nonlinear Model-Based Predictive Control”. Computational Intelligence (LA-CCI), 2015 Latin America Congress on.
[48] A.M. An, et al. “Non-linear model predictive control based on neural network model with modified differential evolution adapting weights”, International Journal of Advanced Mechatronic Systems, 2010.
[49] B. Khusainov, E. C. Kerrigan and G. A. Constantinides. “Multi-objective Co-design for Model Predictive Control with an FPGA”. Control Conference (ECC), 2016 European.
[50] Hughes, P., C., 1980. Spacecraft attitude dynamics. John Wiley & Sons, London.

[51] J. L. Jerez, E. C. Kerrigan and G. A. Constantinides. “A Low Complexity Scaling Method for the Lanczos Kernel in Fixed-Point Arithmetic”. IEEE Transactions on Computers (Volume: 64, Issue: 2, Feb. 2015).
[52] A. Suardi and E. C. Kerrigan and G. A. Constantinides. “Fast FPGA prototyping toolbox for embedded optimization”. Control Conference (ECC), 2015 European.
[53] L. G. Bleris .“A Co-Processor FPGA Platform for the Implementation of Real-Time Model Predictive Control”. Proceedings of the 2006 American Control Conference Minneapolis, Minnesota, USA, June 14-16, 2006.
[54] E. N. Hartley, et al. “Predictive Control Using an FPGA With Application to Aircraft Control”, IEEE Transactions on Control Systems Technology, 2011.
[55] J. L. Jerez, et al. “Embedded Predictive Control on an FPGA using the Fast Gradient Method”, Control Conference (ECC), 2013 European.
[56] B. Kapernick, et al.“A Synthesis Strategy for Nonlinear Model Predictive Controller on FPGA”, Control (CONTROL), 2014 UKACC International Conference on.
[57] S. Longo, et al.“Parallel Move Blocking Model Predictive Control”, Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on.

[58] G. Nielsen. “Interior Point Methods on GPU with application to Model Predictive Control”. Technical University of Denmark (DTU Compute PHD-2014, p 162).

[59] L. Yu, A. Goldsmith and S. Di Cairano. “Efficient Convex Optimization on GPUs for Embedded Model Predictive Control”, Symposium on Principles and Practice of Parallel Programming, 2017.

[60] A. Kelman ; F. Borrelli. “Parallel Nonlinear Predictive Control”, Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on.

[61] The Parallella Board. Disponível em: <https://www.parallella.org/board/>. Acessado em: Maio, 2016.

[62] Adapteva, “Parallella-1.x Reference Manual” REV REV 14.09.09, 2014 - p 50. Disponível em: < http://www.parallella.org/docs/parallella_manual.pdf> Acessado em: Março, 2017
[63] Adapteva, “Epiphany archtecture reference” REV 14.03.11, p 158, 2014. Disponível em: < http://www.adapteva.com/docs/epiphany_arch_ref.pdf>. Acessado em: Março, 2017.
[64] Adapteva, “EPIPHANY SDK 5 RELEASED!”. Disponível em: < http://www.adapteva.com /announcements/the-new-epiphany-sdk-5-release/>. Acessado em: Maio, 2016
[65] T. Vocke. “An evaluation of the Adapteva Epiphany Many-core architecture“, Technical Report, Master’s Thesis, University of Twente/ Thales, 2016
[66] N. Kocher. “Parallella computing: Another distributed system story”. Technical Report, University of Applied Sciences of Western Switzerland , 2016.

[67] J. Dongarra, “The impact of multicore on math software and exploiting single precision computing to obtain double precision results,” in Parallel and Distributed Processing and Applications, ser. Lecture Notes in Computer Science, M. Guo, L. Yang, B. Di Martino, H. Zima, J. Dongarra, and F. Tang, Eds. Springer Berlin Heidelberg, 2006, vol. 4330, pp. 2–2

[68] A. BUTTARI, J. DONGARRA, J. K. J. LANGOU, J. L. P. LUSZCZEK, and S. TOMOV, “Exploiting mixed precision floating point hardware in scientific computations”, University of Tennessee Knoxville ,2007.

[69] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, “Parallel Programming Model for the Epiphany Many-Core Coprocessor Using Threaded MPI,” CoRR, vol. abs/1506.05442, Jun. 2015.

[70] A. Olofsson. “Epiphany Bandwidth Test,” May-2014. [Online]. Disponível em: < https:// github.com/adapteva/epiphany-examples/tree/2015.1/apps/e-bandwidth-test>. Acessado em: Março, 2017.

[71] Zain-ul-Abdin, A. Ahlander, and B. Svensson, “Energy-Efficient Synthetic-Aperture Radar Processing on a Manycore Architecture,” in ICPP, Lyon, 2013, pp. 330–338.

[72] Zain-ul-Abdin, A. Ahlander, and B. Svensson, “Real-time Radar Signal Processing on Massively Parallel Processor Arrays,” in Asilomar 2013, Pacific Grove, CA, 2013, pp. 1810–1814

[73] D. Richie, J. Ross, S. Park, and D. Shires, “Threaded {MPI} programming model for the Epiphany {RISC} array processor,” J. Comput. Sci., vol. 9, no. 0, pp. 94 – 100, 2015.

[74] Eigen C++ template library. Disponível em: <http://eigen.tuxfamily.org/index.php?title= Main_Page>. Acessado em: Abril, 2016.

[75] Rodrigues R. S., Murilo A. and Souza L. C. G. “Projeto de Controlador Preditivo Baseado em Modelo para Sistemas de Controle de Atitude de Satélites Artificiais”. CILAMCE 2016.

[76] GNU Gprof. Disponível em: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html Acessado em: Junho, 2016.

[77] GNU Binutils. Disponível em: <http://www.gnu.org/software/binutils/> . Acessado em: Junho, 2016.

[78] Introdução ao GPROF. Disponível em: < http://www.ibm.com/developerworks /br/local/linux/gprof_introduction/. Acessado em: Junho, 2016.

[79] P. S. Pacheco. “An Introduction to Parallel Programming”, Morgan Kaufmann, 2011, p 370.

[80] Votel, R., Sinclair, D., 2012. Comparison of Control Moment Gyros and Reaction Wheels for Small Earth-Observing Satellites. Proceedings of the AIAA/USU Conference on Small Satellites, Advanced Technologies III, SSC12-X-1.

[81] k. Raj. “Embedded Systems: Architecture, Programming and Design”, 2nd Edition Paperback – March 9, 2009. 704p

[82] F. Xiaocong. “Real-Time Embedded Systems: Design Principles and Engineering Practices” 1st Edition, Kindle Edition, 2015. 686p.

[83] N. Nandan. “From big to LITTLE: A look at the latest Cortex-A processors”, ARM Tech Symposia India – Dezembro 2016.

[84] V. Steffen, et al. “Embedded Parallel Computing Accelerators for Smart Control Units of Frequency Converters”, 29th International Conference on Architecture of Computing Systems, 2016.
[85] R. Hartenstein, “The von Neumann Syndrome”, The Future of Computing, essays in memory of Stamatis Vassiliadis , 2007.
[86] D. E. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann Pub. Inc., 1999, 1025 pages.