UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

DANILLO LEAL BELMONTE

MÉTODO PARA DISTRIBUIÇÃO DA CARGA DE TRABALHO DOS *SOFTWARES* PON EM *MULTICORE*

QUALIFICAÇÃO DE DOUTORADO

CURITIBA
2012

DANILLO LEAL BELMONTE

MÉTODO PARA DISTRIBUIÇÃO DA CARGA DE TRABALHO DOS *SOFTWARES* PON EM *MULTICORE*

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Doutor em Ciências. Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Paulo Cézar Stadzisz
Co-orientador: Prof. Dr. Jean Marcelo Simão

CURITIBA
2012

RESUMO

BELMONTE, D. **Método para distribuição da carga de trabalho dos *softwares* PON em *multicore***. 87 f. Tese (Doutorado em Engenharia Elétrica e Informática Industrial). Pós-Graduação em Engenharia Elétrica e Informática Industrial. Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

Atualmente há uma necessidade por maior capacidade de processamento devido ao aumento da complexidade dos *softwares* comumente desenvolvidos. A indústria de *hardware* vem investindo na produção dos processadores *multicore* e, desde então, tais processadores estão sendo utilizado em larga escala. Para o uso efetivo desses processadores é necessário que o *software* seja executado de forma paralela. Entretanto, exige-se também maior esforço na concepção de *software* paralelo. Ainda, os atuais paradigmas de programação apresentam alto nível de acoplamento de código entre as partes dos objetos. O Paradigma Orientado a Notificações (PON) apresenta baixo nível de acoplamento entre as partes das entidades sendo que elas podem ser separadas “naturalmente” para executarem paralelamente em diferentes núcleos de processamento. Desta forma, esta tese aborda como problema de base o desenvolvimento de *software* que possa fazer uso das infraestruturas de alta capacidade de processamento oferecida pelos processadores *multicore*. Com isso, o objetivo geral desta tese é propor um método para a distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*, contribuindo para o melhor aproveitamento da capacidade de processamento do *hardware* disponível. Percebe-se que existe certa carência nas implementações desenvolvidas atualmente no PON, tanto em relação aos *frameworks*, quanto em relação às aplicações, visto que essas não aproveitam os benefícios da possibilidade de processamento paralelo das entidades que compõem um *software* PON.

Palavras-chave: *Multicore*, *Software* paralelo, Paradigmas de programação, Paradigma Orientado a Notificações, Distribuição da carga de trabalho.

ABSTRACT

Method for distributing the workload of NOP software in multicore

Currently there is a need for greater processing power due to the increased complexity of the software commonly developed. The hardware industry has been investing in the production of multicore processors, and since then, these processors are being used on a large scale. For the effective use of these processors is necessary that the software run in parallel. However, it is also required a greater effort in the conceiving of parallel software. Still, current programming paradigms have a high level of code coupling between the parts of objects. Notification-Oriented Paradigm (NOP) shows low level of coupling between the parts of the entities, they can be separated “naturally” to execute in parallel on different cores. Thus, this thesis discusses as a base problem, software development that can make use of the infrastructure of high processing capability offered by multicore processors. Thus, the principal goal of this thesis is to propose a method for dynamic distribution of the workload of NOP software in multicore, contributing for the better utilization of the processing capacity of the hardware available. It is noticed that there is some deficiency in currently NOP implementations, both in relation to frameworks, and in relation to applications, as these do not use the benefits of parallel processing ability inherent to the entities which compose NOP software.

Keywords: Multicore, Parallel software, Programming paradigms, Notification-Oriented Paradigm, Workload distribution.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS, ACRÔNIMOS E ABREVIATURAS

ANSI	*American National Standards Institute*
API	*Application Programming Interface*
Applet	É um pequeno aplicativo que executa uma tarefa específica, ora em execução no contexto de um programa maior (e.g. um navegador *Web*), ora como um *plug-in*. No entanto, o termo normalmente refere-se à *applets* Java (i.e. programas escritos em linguagem de programação Java que rodam em uma página *Web*).
CPU	*Central Processing Unit*
DLL	*Dynamic-link library*
IEEE	O *Institute of Electrical and Electronics Engineers* (IEEE) é uma associação profissional sem fins lucrativos que é dedicada a promover a inovação tecnológica de excelência. Tem aproximadamente 400.000 membros em mais de 160 países. Seus sócios são engenheiros elétricos, eletrônicos, da computação, entre outros e sua meta é promover conhecimento nesses campos de atuação. Um de seus papéis mais importantes é o estabelecimento de padrões para formatos de dispositivos elétricos, eletrônicos e computadores.
MAC	*Media Access Control* ou MAC *address* é um endereço, um identificador único atribuído às interfaces de rede para comunicações no segmento de rede física. Os endereços MAC são utilizados para inúmeras tecnologias de rede.
Mutex	*Mutex* é acrônimo de *mutual exclusion* que é uma técnica usada em programação concorrente para evitar que dois processos ou *threads* tenham acesso simultaneamente a um recurso compartilhado. Esse acesso é denominado seção crítica.
NOP	*Notification Oriented Paradigm*
NVIDIA	Empresa que fabrica peças de computador e é popularmente conhecida por suas unidades de processamento gráfico, mais especificamente por sua série de placas de vídeo *GeForce*.
Plug-in	É um programa usado para adicionar funções a outros programas maiores, provendo alguma funcionalidade especial ou muito específica. Geralmente pequeno e leve, é usado sob demanda.
QEMU	*Quick EMUlator*, ou seja, emulador rápido é um software livre que implementa um emulador de processador, permitindo uma virtualização completa de um PC dentro de outro. É semelhante a outros emuladores de PC, tais como VMware e PearPC, mas possui características adicionais a esses, incluindo aumento de velocidade em x86 (por meio de um acelerador) e suporte para diferentes arquiteturas. Usando tradução binária dinâmica, atinge uma velocidade razoável, não sendo difícil convertê-lo para novos processadores.
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UNIX	É um sistema operacional que foi desenvolvido em 1969; é portável, multitarefa e multiusuário. Dele originou-se o Linux, sistema operacional comumente encontrado nos dias de hoje.
URL	*Uniform Resource Locator*
Web	A *World Wide Web* (também conhecida como *Web* e www) é um sistema de documentos em hipermídia que são interligados e executados na Internet. O usuário pode então seguir os *hiperlinks* nas páginas para outros documentos ou mesmo enviar informações de volta para o servidor para interagir com ele.

SUMÁRIO

1. INTRODUÇÃO

Este capítulo apresenta o contexto desta qualificação de tese, bem como o problema tratado e os objetivos pretendidos. São também expostos os fatores motivadores que justificam este trabalho e o método de pesquisa utilizado para seu desenvolvimento. Por fim, a organização dos capítulos que compõem esta tese é descrita.

1.1. CONTEXTO DA PESQUISA

Na maioria das aplicações atuais da computação há uma demanda crescente por maior capacidade de processamento em razão do aumento da complexidade e sofisticação dos *softwares* utilizados. Até algum tempo esta demanda por capacidade de processamento foi atendida pelo aumento na integração e frequência de operação (*clock*) dos processadores. Nos últimos anos outra estratégia tem sido largamente empregada que consiste em multiplicar o número de unidades de processamento sem aumento da frequência de operação. Neste âmbito, foram desenvolvidos processadores com múltiplos núcleos (*multicore*), os quais estão sendo utilizado em larga escala [1].

A tecnologia *multicore* consiste na integração de dois ou mais núcleos de processamento (*cores*) em um único processador (*chip*). O processador *dualcore*, por exemplo, contém dois núcleos de processamento e o *quadcore* quatro. Outro exemplo de utilização são os processadores *multicore* empregados para executar jogos como o Xenon *threecore* do Xbox 360.

Os sistemas operacionais (SOs) convencionais (e.g. Windows 7 e distribuições do Linux mais recentes) fazem uso dos processadores *multicore* e realizam o escalonamento de tarefas (ou, em inglês, *thread scheduling*) [2] [3], dividindo a carga de processamento das aplicações. Entretanto, a verdade é que tal particionamento entre os núcleos não é satisfatório.

Para o desenvolvimento de *software* para ambiente *multicore* podem-se utilizar recursos de programação, tais como a API *PThreads*, a linguagem de programação de propósito geral *Cilk*, a biblioteca *Threading Building Blocks*, além de *pragmas* de compilador, tais como *OpenMP* e CUDA. Esses recursos amenizam o

trabalho do desenvolvedor, pois a programação paralela exige maior esforço na sua concepção.

Ao programar para ambiente *multicore*, com a possibilidade de implementação de paralelismo efetivo, a programação paralela tem sido frequentemente adotada para o desenvolvimento de aplicações que demandam alto desempenho. Entretanto, apesar dos recursos de programação disponíveis, para prevalecer-se dos múltiplos núcleos, a implementação de *software* deveria ser concebida de forma a poder distribuir suas partes “naturalmente” em tempo de execução. Mas a realidade é que isso não acontece e, basicamente, devido a dois fatores.

O primeiro fator é, pois, para escrever programas que aproveitem efetivamente os múltiplos núcleos de processamento, é necessário que o desenvolvedor despenda maior esforço na programação paralela [1] [4]; e, o segundo, devido ao alto nível de acoplamento de código entre as partes dos objetos, uma das principais deficiências dos atuais paradigmas de programação [5] [6].

O aspecto central é, portanto, a forma de distribuição da carga de processamento das aplicações entre os núcleos. Isso pressupõe, porém, que a maioria dos *softwares* tenham sido projetados para tirar proveito do paralelismo disponível. De fato, o desempenho obtido por meio da utilização de um processador *multicore* depende do problema a ser resolvido, bem como da sua implementação em *software*, ou da maneira como ele é concebido segundo um dado paradigma [5].

Adicionalmente, o forte acoplamento de código gera dificuldades na partição do *software* em conjuntos independentes de objetos, usando excessivamente mecanismos de sincronização para manter consistência entre os dados [7]. O emprego desses mecanismos de sincronização para prevenir o acesso concorrente aos dados deve ser controlado explicitamente pelo desenvolvedor. Essa prática dificulta a programação paralela, gerando muitas falhas de programação devido ao esforço adicional atribuído ao desenvolvedor.

O Paradigma Orientado a Notificações (PON), desenvolvido por pesquisadores do Laboratório de Sistemas Inteligentes de Produção (LSIP) da UTFPR [8] [9], apresenta alternativas viáveis para a construção de *softwares* paralelos, como desacoplamento implícito entre as partes das entidades, incluindo particularmente as entidades lógico-causais.

O uso do PON motiva pesquisas que, a princípio, proverão facilidades na paralelização de código. Entretanto, a aplicação do PON para tal deve ser ainda melhor estudada a fim de prover, por exemplo, soluções efetivas para tratar de estratégias de balanceamento de carga. Ainda, em termos práticos, os conceitos do PON foram inicialmente concebidos sobre o Paradigma Orientado a Objetos (POO), por meio de um *framework* desenvolvido na linguagem de programação C++, o qual é específico para ambientes monoprocessados.

Portanto, utilizando-se as atuais linguagens e paradigmas de programação, a grande questão é que os SOs não poderão distribuir a carga de processamento de forma adequada entre os núcleos, pois eles não podem antever a variação dinâmica da carga dos programas ao longo do tempo. Cabe ao desenvolvedor de cada *software* construir os algoritmos e a arquitetura que distribua sua carga de processamento. Deste modo, pesquisas sobre distribuição da carga de processamento em plataformas *multicore* são necessárias. Essas pesquisas permitirão desenvolver novas técnicas, métodos e paradigmas para a construção de *software*. A proposta de pesquisa apresentada nesta tese se insere nesse contexto.

1.2. PROBLEMA

A construção de *software* eficiente e de execução paralela é negligenciada frente ao alto crescimento da indústria de *hardware* (em especial, da tecnologia *multicore*), ou seja, a capacidade e produtividade dos desenvolvedores de *software* não acompanha a ascensão da indústria de *hardware*. Há aumento da quantidade de problemas complexos a serem resolvidos computacionalmente. Ainda, as linguagens e os paradigmas de programação existentes, devido ao forte acoplamento de código, dificultam ainda mais a concepção de *software* paralelo [5] [6] [10]. Desta forma, esta tese considera o seguinte problema de base.

Como desenvolver *software* que possa fazer uso das infraestruturas de alta capacidade de processamento oferecida pelos processadores *multicore*?

O PON apresenta baixo nível de acoplamento entre as partes das entidades sendo que elas podem ser separadas para executarem paralelamente em

diferentes núcleos de processamento. A concepção de *software* utilizando o PON torna-o atrativo para sua aplicação no desenvolvimento de *software* paralelo. Porém, hoje, o PON, implementado como um *framework* é específico para ambientes monoprocessados. Com isso, do problema de base apresentado anteriormente, tem-se o seguinte problema decorrente.

> **Não há um modelo nem técnica para tratar explicitamente e usufruir da programação paralela usando o PON**.

Os conceitos do PON foram inicialmente implementados sobre o POO, por meio de um *framework* desenvolvido na linguagem de programação C++. A versão prototipal foi concebida por Simão [9] [11] [12] [13] [14] e a versão original, intitulada *Framework* PON, por Banaszewski [4]. Ainda, recentemente, o *Framework* PON sofreu evoluções por meio dos trabalhos de Ronszcka [15] [16] e Valença [17]. Entretanto, ambas as implementações, versão prototipal e original, são específicas para ambientes monoprocessados. Como consequência, o desenvolvimento de aplicações PON que façam uso de múltiplos processadores é limitada (se usando *threads* [18]) ou impraticável com o *framework* atualmente desenvolvido.

1.3. OBJETIVOS

O PON trás elementos promissores para a construção de *software* paralelo uma vez que ele favorece o desacoplamento dos módulos do *software* [5]. Esta pesquisa explora esse aspecto e tem por objetivo geral propor um método para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*, contribuindo para o melhor aproveitamento da capacidade de processamento do *hardware* disponível. Trata-se de uma nova abordagem a ser empregada para a resolução de problemas desafiadores ligados à programação paralela em ambientes com vários núcleos. Dentro do objetivo geral enunciado, essa pesquisa tem os seguintes objetivos específicos:

- Analisar o PON com vistas ao paralelismo de *software*. O PON apresenta baixo nível de acoplamento de código sendo um dos requisitos principais ao desenvolvimento de *software* paralelo.

- Aprofundar o conhecimento sobre a tecnologia *multicore* visando à concepção de técnicas de paralelismo de *software*. Em computadores com processadores *multicore* as tarefas podem ser processadas de forma realmente paralela.
- Propor um conjunto de técnicas para a distribuição da carga de trabalho dos *softwares* PON em *multicore*. A dissociação entre os componentes de *software* fornecidos naturalmente pelo PON mostra-se como uma alternativa viável para a distribuição de carga de trabalho.
- Avaliar o método proposto por meio de experimentos. O método será desenvolvido como uma extensão do *Framework* PON e como a distribuição da carga de trabalho de *software* PON acontecerá em ambiente *multicore*, o paralelismo em nível de *threads* apresenta-se como o mais adequado.

1.4. MOTIVAÇÃO

A solução do PON é considerada inovadora, não consistindo apenas de um acréscimo aos conceitos vigentes, apesar da sua inspiração nos estilos de programação já existentes. As suas contribuições mostram-se apropriadas. Ainda, o PON vem apresentando resultados favoráveis no âmbito do estado da técnica, com implementações de *frameworks* e aplicações, que norteiam suas evoluções.

Entretanto, percebe-se que existe certa carência nas implementações, visto que essas não aproveitam os benefícios da possibilidade de processamento paralelo das entidades que compõem um *software* PON. Neste sentido, a proposta desse trabalho, além de viabilizar a distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*, fornece um ambiente de execução capaz de operacionalizar esse tipo de aplicação.

1.5. METODO DE PESQUISA ADOTADO

O método de pesquisa adotado nesta tese, ilustrado na Figura 1, é composto pelas seguintes etapas: caracterização da pesquisa, revisão bibliográfica, revisão sistemática, realização da proposta e qualificação da proposta.

Iniciou-se a pesquisa com a definição do tema (i.e. *multicore*) e do contexto (i.e. programação distribuída para ambiente *multicore*). Posteriormente, definiu-se o problema, por meio da formulação de uma pergunta e, em seguida, propôs-se os objetivos a serem alcançados.

Foram selecionadas as literaturas pertinentes (i.e. revisão bibliográfica), buscando referências em relação ao tema, contexto e problema. Nessa etapa as referências selecionadas e analisadas, consistiram basicamente em: SO, gerenciamento de processos, arquiteturas de computadores e paradigmas de programação.

A terceira etapa, revisão sistemática, contemplou avanços na seleção e análise da literatura pertinente, sendo essa composta por: PON, *multicore*, sistemas paralelos e EPOS (acrônimo de *Embedded Parallel Operating System*), que é um sistema operacional paralelo para plataformas embarcadas, definido como infraestrutura para ensaios desta tese.

Na próxima etapa, realização da proposta, definiu-se o método e redigiu-se o texto para a qualificação, além de artigos que foram escritos (autoria e coautoria) e submetidos para periódicos e congressos, logrando êxito em alguns desses, sendo um em especial em relação a esta pesquisa.

Por fim, a última etapa, constituiu da qualificação da proposta desta tese. Desta forma, os passos utilizados para a realização dessa pesquisa são os ilustrados na Figura 1.

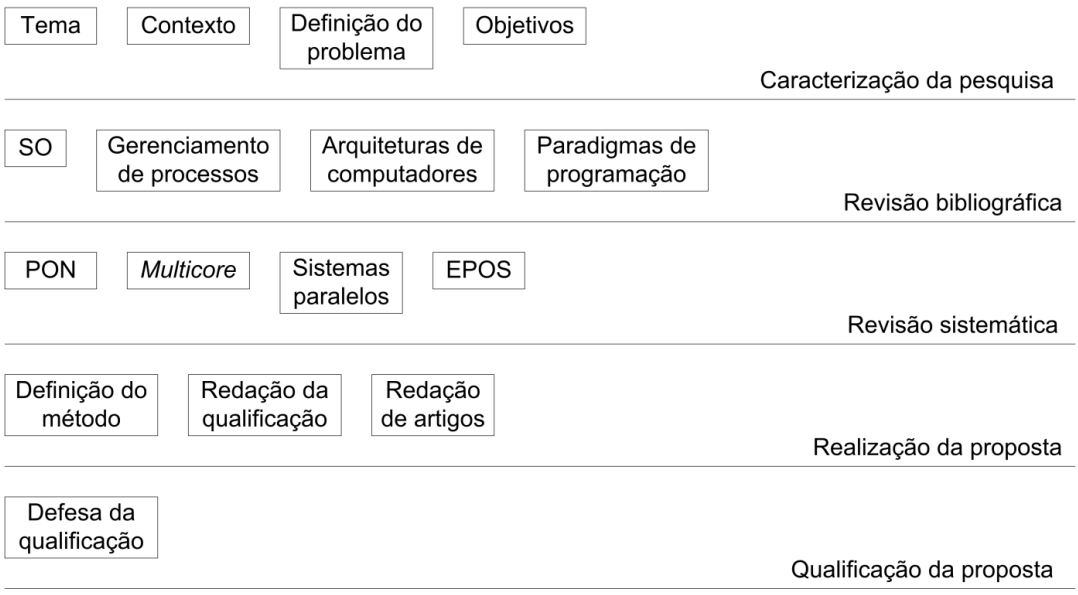

Figura 1. Método de pesquisa adotado

1.6. ORGANIZAÇÃO DA TESE

Esta tese é organizada em seis capítulos. No Capítulo 2, apresenta-se um embasamento teórico dos principais conteúdos ao seu entendimento, que são o Paradigma Orientado a Notificações, o escalonamento de tarefas, a tecnologia *multicore* e o *Embedded Parallel Operating System*. No Capítulo 3, apresenta-se o método proposto para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*. No Capítulo 4, apresentam-se ensaios por meio de experimentos. Para as experimentações foi desenvolvimento um caso de estudo que é um simulador para o controle de operação de um avião. No Capítulo 5, apresentam-se as considerações finais e perspectivas para possibilidades de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo fornece uma contextualização sobre a essência do PON e os principais conceitos sobre paralelismo e distribuição. Escalonamento de tarefas é explicado na sequência. São enfatizados, também, os conteúdos relativos às arquiteturas paralelas e técnicas para obtenção de desempenho relativa à tecnologia *multicore*. Ainda, recursos de programação para ambiente *multicore* são abordados. Posteriormente, expõe-se o *Embedded Parallel Operating System*, que é um SO baseado em componentes para a geração de ambientes dedicados.

2.1. PARADIGMA ORIENTADO A NOTIFICAÇÕES

O Paradigma Orientado a Notificações (PON) foi concebido por Simão (por meio dos seus trabalhos de dissertação [8] e tese [9]), ambos sob a orientação de Stadzisz, no Laboratório de Sistemas Inteligentes de Produção (LSIP) da UTFPR. Ainda, o PON foi aprimorado por Banaszewski, Ronszcka e Valença (por meio dos seus trabalhos de dissertação [4] [15] [17]) sob a orientação de Simão e co-orientação de Stadzisz.

O PON proporciona uma solução advinda em partes do Paradigma Imperativo (PI) e do Paradigma Declarativo (PD). É um paradigma que se guia e evolui de dois sub-paradigmas do PI e PD. O PON se inspira no POO e no Paradigma Lógico (PL), enfatizando conceitos dos Sistemas Baseados em Regras (SBR), que oferecem modelos de programação com proximidade à cognição humana [4]. Dessa forma, reaproveita os principais conceitos do POO, como a abstração em forma de classes e objetos e a reatividade da programação dirigida a eventos. Também reaproveita conceitos próprios dos SBR, como a representação do conhecimento como regras e as facilidades da programação declarativa. A Figura 2 ilustra a relação do PON com os outros paradigmas.

Na Figura 2, graças ao reaproveitamento dos principais conceitos do POO e dos SBR, o PON constrói-se a partir da intersecção deles. Com isso, tal representação mostra que o PON apresenta a possibilidade de uso (de partes) de ambos os estilos de programação. Dessa forma, o programador tem a liberdade de empregar princípios do PD para se beneficiar das facilidades de programação e usar os princípios do PI para manter a flexibilidade na programação [4].

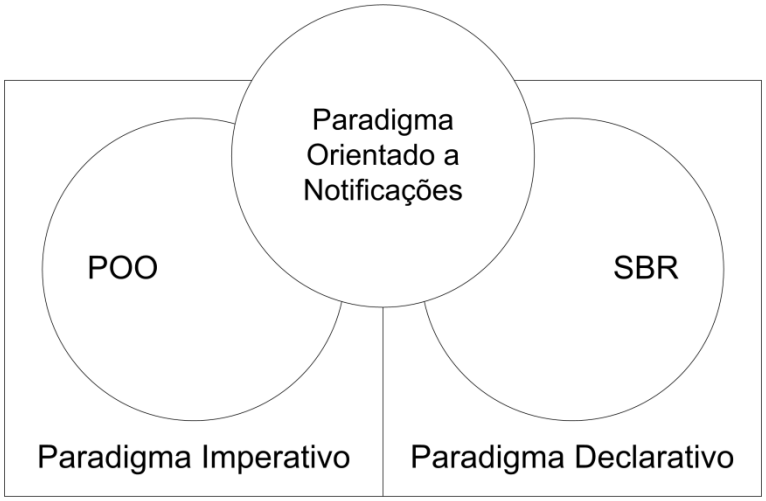

Figura 2. Relação entre o PON, o PI e o PD

2.1.1. A origem do PON

O objetivo original dos trabalhos de dissertação e tese de Simão [8] [9] foi propor um novo mecanismo de controle que suprisse as necessidades relacionadas aos sistemas modernos de produção, tais como o tratamento das variações de produção e a produção em massa [9].

Simão [9] propôs uma abordagem de controle que permite, de forma intuitiva, organizar as colaborações entre entidades de manufatura (e.g. recursos ou equipamentos) a fim de alcançar agilidade na produção. Tal abordagem refere-se a um meta-modelo de controle discreto que foi aplicado à simulação de sistemas de manufatura ditos holônicos. Essa simulação ocorreu na ferramenta de projeto e simulação de sistemas de manufatura ANALYTICE II[1].

As entidades de manufatura desses sistemas “inteligentes” são integradas a sistemas computacionais “comuns” (e.g. *software* de controle) por meio de “recursos virtuais” (i.e. *drivers* avançados), que permitem o acesso a dados e serviços pelos sistemas computacionais por meio de uma rede de comunicação de dados.

Para exemplificar, suponha a entidade de manufatura Kuka386 (i.e. robô de transporte de peças) integrada com um componente de um sistema computacional de controle por meio do Kuka386-virtual (i.e. um *smart-driver*). Tal

[1] Esse projeto foi desenvolvido por gerações de pesquisadores do LSIP da UTFPR [55] [3].

integração se dá por meio da rede de comunicação de dados, onde os *feedbacks* (e.g. comandos) entre o sistema computacional e o equipamento real são intermediados pelo recurso virtual ou *driver* desse equipamento.

Todo e qualquer recurso virtual expressa os estados ou valores do respectivo equipamento por meio de entidades chamadas atributos, bem como disponibiliza seus serviços por meio de entidades chamadas métodos. Assim sendo, todos os recursos-virtuais apresentam a mesma forma de *feedback* para com um determinado sistema computacional. No domínio da manufatura, o meta-modelo em questão permite compor *software* de controle onde a representação das relações causais de controle se dá por meio de regras causais sobre os atributos e métodos dos recursos-virtuais. A Figura 3 ilustra um exemplo de uma regra PON na forma de conhecimento causal [9]. A semântica dessa regra refere-se ao controle das relações entre os equipamentos Lathe.1 (i.e. um torno mecânico), ERIII.1 (i.e. um robô de transporte de peças) e Table.3 (i.e. uma mesa para armazenamento temporário de peças), os quais compõem parte da célula de manufatura simulada no ANALYTICE II.

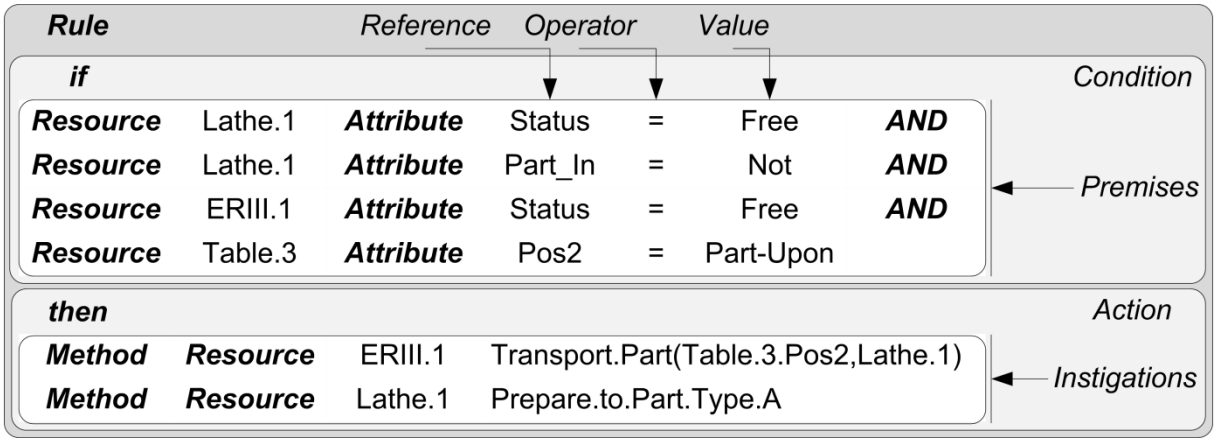

Figura 3. Exemplo de uma regra no PON

Em linhas gerais, uma regra consiste em uma condição e uma ação, na qual a condição representa a avaliação causal de uma regra, enquanto uma ação consiste no conjunto de instruções (comandos) executáveis de uma regra.

No exemplo, a condição da regra verifica se os recursos Lathe.1 e ERIII.1 estão livres, se o recurso Lathe.1 não tem peça dentro de si e se há algum produto sobre o recurso Table.3. Se esses estados forem constatados, a ação da regra faz

com que o robô ERIII.1 transporte o respectivo produto para ser manipulado pelo torno Lathe.1.

No meta-modelo do PON, cada regra causal é computacionalmente representada por um conjunto de entidades relacionadas. Dentre elas, o cerne é a entidade *Rule*. Uma *Rule* é decomposta em uma entidade *Condition* e uma entidade *Action*. De maneira similar, uma *Condition* é decomposta em uma ou mais entidades *Premises* e uma *Action* é decomposta em uma ou mais entidades *Instigations*. Cada *Resource* (Recurso) também é decomposto em entidades menores, os *Attributes* e os *Methods*.

Todas as entidades colaboram por meio de inferências baseadas em notificações, a fim de ativar as regras pertinentes para execução [11]. Essa abordagem é explicitada pelo esquema ilustrado na Figura 4 [9]. Nessa abordagem, cada *Resource* notifica o seu conhecimento *factual* por meio de capacidades reativas incorporadas às suas entidades *Attributes* às demais entidades envolvidas.

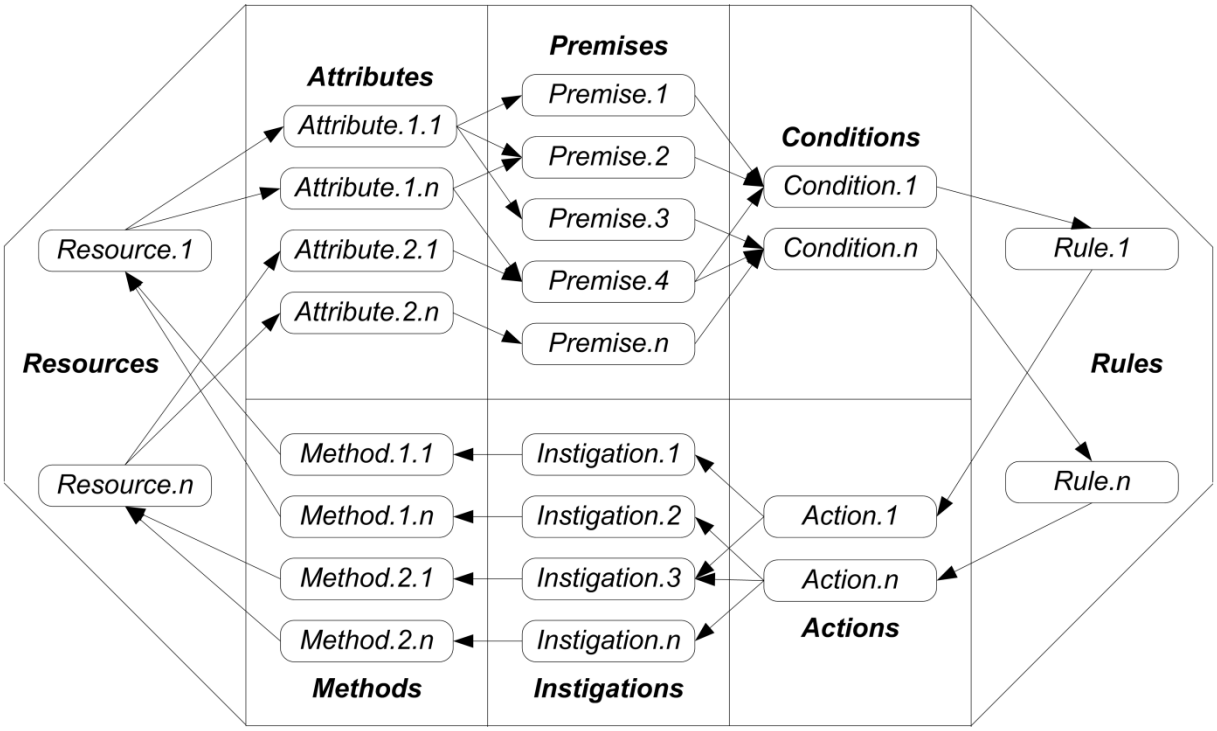

Figura 4. Colaboração por notificações

Sucintamente, a cada mudança no estado de um *Attribute*, ele próprio notifica imediatamente uma ou um conjunto de entidades *Premises* relacionadas para que essas reavaliem os seus estados lógicos, comparando o valor notificado

com outro valor (uma constante ou um valor notificado por outro *Attribute*) usando um operador lógico. Se o valor lógico da entidade *Premise* se alterar, ela notifica uma ou um conjunto de entidades *Conditions* conectadas para que seus estados lógicos sejam reavaliados. Desse modo, cada entidade *Condition* notificada reavalia o seu estado lógico de acordo com o valor recém notificado pela *Premise* em questão e os valores notificados previamente pelas demais *Premises* conectadas. Assim, quando todas as entidades *Premises* que compõem uma entidade *Condition* apresentam o estado lógico verdadeiro, a entidade *Condition* é satisfeita, decorrendo na aprovação da sua respectiva *Rule* para a execução. Com isso, a entidade *Action* conexa a essa *Rule* é executada, podendo invocar serviços (*Methods*) nos recursos por intermédio das entidades *Instigations* [9].

Portanto, por meio do mecanismo de inferência, o meta-modelo provê uma solução efetiva para compor e executar *software* de controle no domínio de sistemas de manufatura modernos. Tal fato foi confirmado pela implementação e análise do meta-modelo sobre diversas perspectivas no simulador ANALYTICE II [9] [19]. Nessas análises, o meta-modelo cumpriu as expectativas [8] [9] [20] [21] [19].

2.1.2. Mecanismo de notificações

O mecanismo de notificações consiste na estrutura interna de execução das instâncias do PON, que determina o fluxo de execução das aplicações. Por meio desse, as responsabilidades de uma aplicação são divididas entre seus objetos, que cooperam por meio de notificações, informando uns aos outros as suas contribuições para formar o fluxo de execução da aplicação. As relações pelas quais os objetos colaboram são ilustradas no diagrama de classes UML da Figura 5.

Os objetos das classes *Rule* e FBE se apresentam em extremidades opostas e se relacionam com o auxílio dos objetos colaboradores conforme as conexões modeladas [4]. Tais conexões são estabelecidas em tempo de execução à medida que os objetos são criados. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado como o seu *Reference*.

Na Figura 5, pode-se constatar a modelagem de dois fluxos opostos de notificações: o fluxo ativo e o passivo em relação aos objetos da extremidade. Por exemplo, o fluxo de notificações originado no FBE é ativo em relação ao FBE e passivo em relação à *Rule*. De maneira oposta, o fluxo de notificações originado na

Rule é ativo em relação à *Rule* e passivo em relação ao FBE. Esses fluxos são formados pela cooperação entre os objetos colaboradores dessas extremidades.

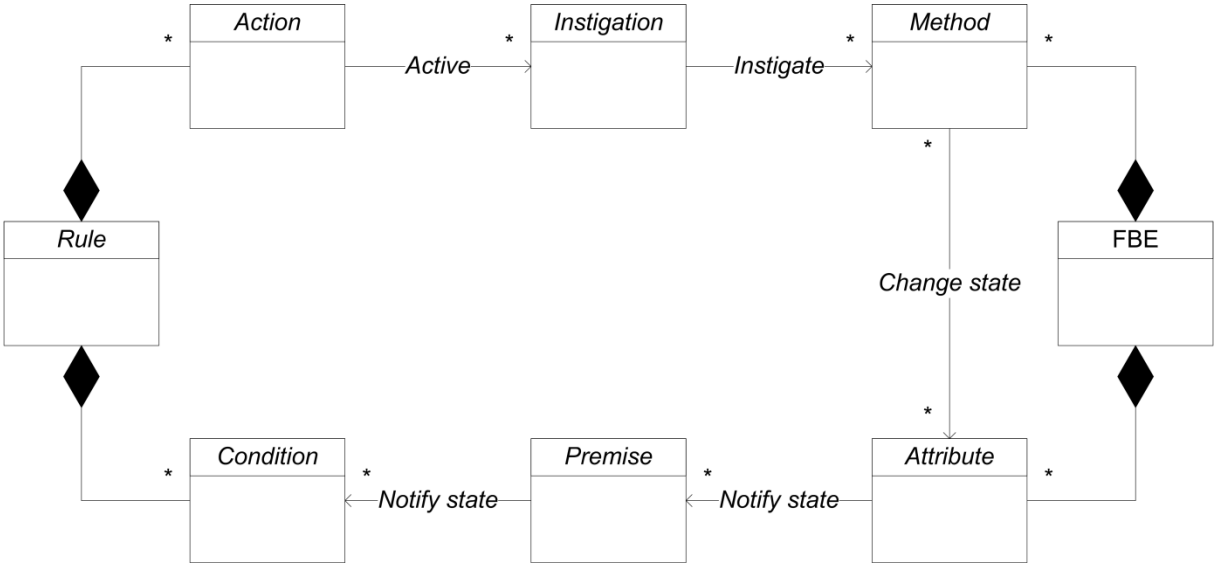

Figura 5. Relação entre os objetos no PON

Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente essa *Premise* como sendo interessada em receber notificações sobre o seu estado. Dessa forma, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando seus estados mudam. Da mesma forma, quando uma *Premise* é conectada a uma *Condition*, a *Premise* considera automaticamente essa *Condition* como interessada em receber notificações sobre o seu estado. Com isso, a *Premise* identifica todas as *Conditions* interessadas e notifica-as quando seus estados mudam. Assim, após as devidas conexões estabelecidas, tais objetos estão aptos a se comunicarem por meio de notificações [4].

A cooperação entre tais objetos se inicia a cada mudança no estado de um objeto *Attribute*. Devido à ocorrência desse evento, o próprio *Attribute* notifica uma ou um conjunto de *Premises* relacionadas para que essas reavaliem os seus estados lógicos. Desse modo, uma *Premise* realiza um cálculo lógico a cada momento em que recebe notificações de um *Attribute*, comparando o elemento *Reference* com o *Value*, usando o elemento *Operator*. Se o valor lógico da *Premise* se altera, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, que ocorre por meio da notificação sobre a mudança de seu

estado lógico. Logo, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico utilizado (conjunção ou disjunção). Igualmente, no caso de uma conjunção, quando todas as *Premises* que integram uma *Condition* são satisfeitas (estado verdadeiro), a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* para a execução [4].

O momento exato da execução de uma *Rule* é determinado após a resolução de conflito entre essa e as demais regras aprovadas, caso hajam conflitos. Um conflito ocorre quando duas ou mais regras referenciam um mesmo recurso e demandam exclusividade de acesso a esse recurso.

Em seguida, as regras concorrem para adquirir acesso exclusivo a esse recurso, sendo que somente uma dessas regras em conflito pode executar por vez (que obteve o acesso exclusivo). Com os conflitos solucionados, a respectiva *Rule* está pronta para executar o conteúdo da sua *Action*. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto FBE por meio dos seus objetos *Methods*. Comumente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça [4].

Portanto, percebe-se que objetos colaboradores se apresentam desacoplados ou “minimamente acoplados” devido à comunicação realizada por meio de notificações. Tais comportamentos (i.e. comunicações por notificações) e estruturas (i.e. desacoplamento) favorecem a aplicação do mecanismo de notificações para ambientes multiprocessados, pois se faz necessário que um objeto notificante conheça o endereço do objeto notificado para que uma notificação ocorra. Além do mais, as notificações são pontuais e necessárias devido à mudança de estado do *Attribute*. Tal fato colabora para reduzir as comunicações entre os nós de processamento e para otimizar o processamento em cada nó.

2.1.3. Paralelismo e distribuição no PON

As arquiteturas paralelas podem ser classificadas de acordo com o acesso à memória e a composição e distribuição das unidades de processamento. A principal categorização da distribuição da memória é feita entre memória compartilhada e memória distribuída [22].

No modelo de memória compartilhada todos os processadores dividem e acessam a mesma memória. Este é o caso dos processadores *multicore*. Trata-se de uma única máquina, em que há um processador com um ou mais núcleos. No modelo de memória distribuída, os processadores de máquinas distintas interagem por meio de uma interface de rede. Nesse modelo as máquinas distribuídas (e.g. *clusters*) podem ter memória, características de processamento e até SO diferentes. O essencial para o funcionamento dessa estrutura é haver um meio de comunicação entre os processadores por meio de um protocolo de comunicação comum.

A computação paralela pode ser vista como uma forma particular de computação distribuída, e a computação distribuída podem ser vista como uma forma de computação paralela. No entanto, é possível classificar sistemas concorrentes como “paralelos” ou “distribuídos” com os seguintes critérios [22]:

- Na computação paralela, todos os processadores têm acesso a uma memória compartilhada. Essa memória compartilhada pode ser usada para troca de informações entre os processadores, conforme ilustra a Figura 6.

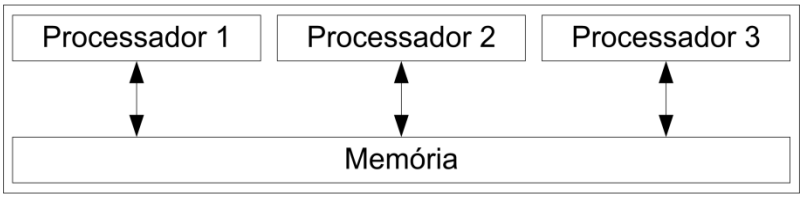

Figura 6. Paralelismo (ou sistema paralelo)

- Na computação distribuída, cada processador tem sua própria memória privada (memória distribuída). As informações são trocadas por passagem de mensagens entre os processadores, conforme ilustra a Figura 7.

No PON, com a possibilidade de particionar os objetos, como em termos de atributos e métodos, o programador pode se beneficiar do uso da computação paralela e distribuída, devido ao maior desacoplamento entre as partes dos objetos. Por exemplo, atributos e métodos podem ser separados para executarem paralelamente em diferentes nós de processamento caso seja necessário.

Uma das principais deficiências dos atuais paradigmas está relacionada aos seus mecanismos de execução, no percorrer sobre elementos passivos. Tais buscas, quando excessivas, afetam o desempenho das aplicações podendo

inviabilizar a implementação de certos *softwares* sobre um dado *hardware* disponível.

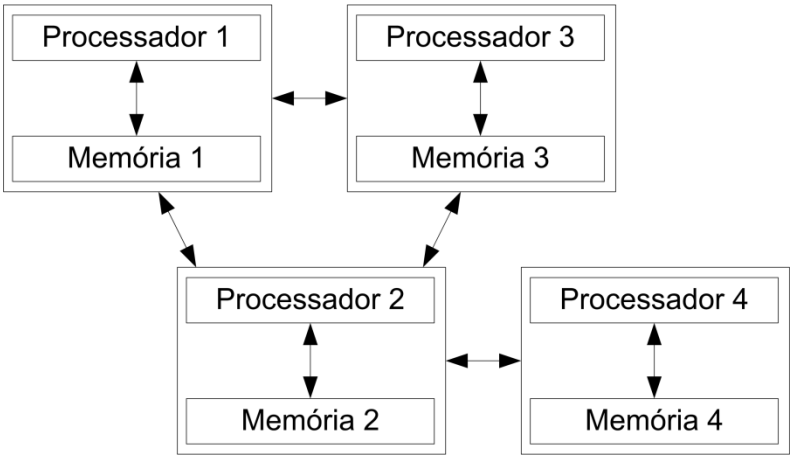

Figura 7. Distribuição (ou sistema distribuído)

O PON oferece uma solução que permite programar aplicações menos acopladas. Ao programar no estilo orientado a notificações, o programador automaticamente cria programas com alta capacidade de reuso dos objetos, principalmente dos objetos FBE. Os FBEs apresentam maior capacidade de reuso do que os objetos típicos do POO, uma vez que os relacionamentos de um elemento da base de fatos com outros ou objetos comuns do POO, são naturalmente reduzidos.

O PON também pode permitir que o programador obtenha os reais benefícios da computação paralela e distribuída. Esses benefícios se devem a organização da estrutura e comportamento dos componentes do PON, que são alcançáveis de forma transparente pelo programador. Ainda, as características declarativas do PON podem ser adotadas para poupar o programador das particularidades que envolvem a computação multiprocessada.

Os componentes do PON (i.e. FBE, *Rules* e suas decomposições) são estruturados e organizados para favorecer a execução paralela. Esses podem ser alocados independentemente a diferentes nós de processamento. Também apresentam comportamento que incentivam a execução nesses ambientes principalmente porque esses se comunicam de forma ativa e pontual por meio de notificações [9].

No PON, os componentes podem cooperar independentemente de suas localidades. Nesse paradigma, um componente pode notificar o seu estado para outro componente indiferentemente se esse está localizado na mesma região da memória, no mesmo computador ou na mesma sub-rede. Por exemplo, um componente notificante (e.g. *Attribute*) pode estar executando em um computador enquanto que o componente a ser notificado (e.g. *Premise*) pode estar executando no mesmo computador ou em outro. Para que a notificação ocorra entre os componentes, o componente notificante somente precisa conhecer o endereço na memória ou em outro computador onde o componente a ser notificado está alocado. Dessa forma, subentende-se que os componentes que compõem uma *Rule* ou um FBE podem estar distribuídos entre nós de processamento permitindo que esses sejam executados paralelamente, cooperando por meio de notificações. Assim sendo, o PON permite que vários fluxos de notificações ocorram de forma paralela.

Geralmente, o acesso concorrente aos atributos e métodos de um objeto é dificultado, principalmente porque os comandos (i.e. código de um método) e dados (i.e. variáveis) são fortemente relacionados e a execução desses comandos em relação aos dados ocorre de forma descoordenada, ou seja, sem o auxílio de um mecanismo como o de resolução de conflitos do PON. Por isso, o POO adota mecanismos de sincronização mais “caros” em termos de processamento (e.g. monitores e semáforo) para manter a execução dos programas, os quais são definidos explicitamente pelo programador, podendo gerar muitas falhas de programação devido à responsabilidade atribuída ao programador.

O PON permite uma nova forma de a concepção dos programas. Oferece uma nova forma de computar e programar, a qual permite amenizar as deficiências dos atuais paradigmas de programação. Essas deficiências são relativas à ineficiência de execução, dificuldades de programação e forte acoplamento entre as partes dos programas. O fato do PON reduzir o nível de acoplamento entre as partes das aplicações e otimizar suas interações pode permitir que essas sejam mais facilmente distribuíveis e melhor obtenham benefícios da computação multiprocessada. Novas pesquisas podem ser iniciadas para estudar a aplicação do PON em ambientes multiprocessados.

2.2. MEIOS DE ORGANIZAÇÃO DA EXECUÇÃO DE *SOFTWARE*

Segundo Tanenbaum [23], uma tarefa é definida como sendo a execução de um fluxo sequencial de instruções construído para atender uma finalidade específica. Desta forma, é um conceito dinâmico, que possui um estado bem definido a cada instante e, normalmente, são implementadas por *threads*. Ainda, as tarefas definem as atividades a serem realizadas dentro do *software*. Geralmente, existem mais tarefas a realizar que processadores disponíveis e as tarefas têm diferentes prioridades, desse modo há necessidade de gerenciamento dessas tarefas. Portanto, a gerência de tarefas tem grande importância dentro de um SO. Cabe ao SO organizar as tarefas definindo uma ordem para executá-las.

Os SOs atuais gerenciam suas tarefas por meio dos intitulados sistemas de tempo compartilhado (ou, em inglês, *time-sharing*) [23]. Na solução do gerenciamento de tarefas por meio dos sistemas de tempo compartilhado, cada atividade que detém o processador recebe um limite de tempo de processamento (i.e. *quantum*). Esgotado seu *quantum*, a tarefa em execução perde o processador e volta para uma fila de tarefas “prontas”, que estão na memória aguardando sua oportunidade de executar.

O diagrama de estados das tarefas com preempção por tempo que implementa a estratégia de tempo compartilhado é ilustrado na Figura 8. O diagrama ilustrado na Figura 8 é conhecido na literatura como “Ciclo de vida das tarefas” [24].

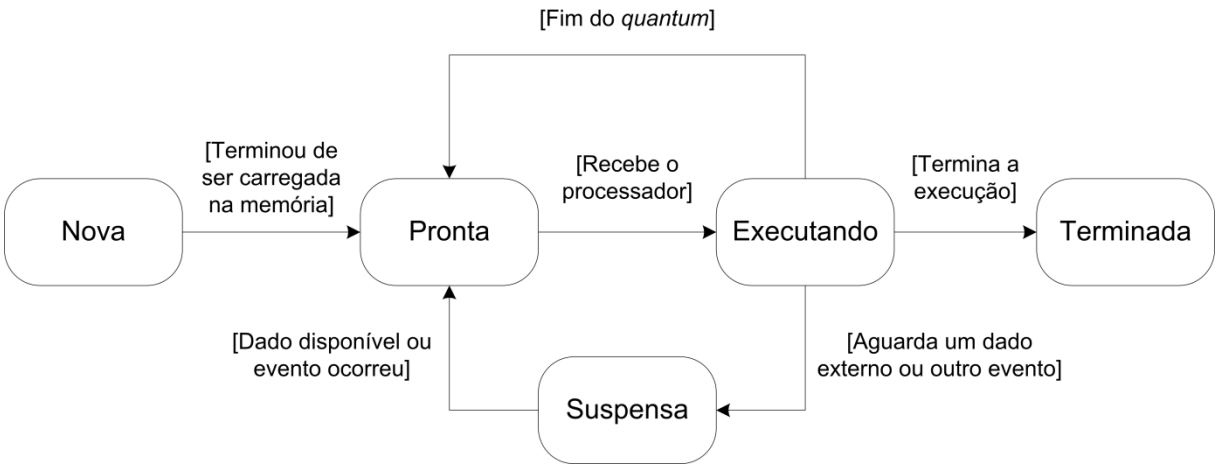

Figura 8. Tarefa em um sistema de tempo compartilhado

2.2.1. Processos

Um processo pode ser definido como um conjunto dos recursos alocados a uma tarefa para sua execução. Assim, para cada tarefa ativa em um *software*, um conjunto de recursos para executar e cumprir seu objetivo é necessário, além de seu próprio código [23].

Conforme Tanenbaum [23], os conceitos de processo e tarefa se confundem, principalmente porque os SOs mais antigos suportavam somente uma tarefa por processo. Mas, quase todos os SOs atuais suportam mais de uma tarefa por processo [2] [3].

Os SOs convencionais atuais associam uma tarefa por processo, o que corresponde à execução de um programa sequencial (i.e. um único fluxo de instruções dentro do processo). Caso se deseje associar mais tarefas ao mesmo processo, o desenvolvedor deve escrever o código necessário para tal [24].

2.2.2. *Threads*

De forma geral, cada fluxo de execução do sistema, seja associado a um processo ou no interior do núcleo, é denominado *thread* [23].

A necessidade de suportar aplicações com várias *threads* (i.e. *multithreaded*) levou os desenvolvedores a incorporar a gerência das *threads* ao núcleo do SO. Para cada *thread* (dita de usuário) foi definido uma *thread* correspondente dentro do núcleo. Assim, caso uma *thread* de usuário solicite um bloqueio qualquer, somente sua respectiva *thread* de núcleo será suspensa, sem afetar as demais. Tal forma de implementação é denominada “Modelo de *Threads* 1 para 1” e é a mais frequente nos SOs atuais [2] [3].

2.2.3. Escalonamento de tarefas

Segundo Tanenbaum [23], o escalonador (ou, em inglês, *task scheduler*), que decide a ordem de execução das tarefas “prontas”, é um dos componentes mais importantes da gerência de tarefas.

Para definir o comportamento do SO, permitindo obter *softwares* que tratem de forma mais eficiente e rápida as tarefas que devem ser executadas, o

escalonador utiliza-se de algum algoritmo. Dentre os critérios que compõem um algoritmo de escalonamento apresentam-se [23]:

- Eficiência: mantém o processador ocupado 100% do tempo.
- Imparcialidade: assegura que cada processo receba sua justa parte do processador.
- Tempo de resposta: minimiza o tempo de resposta para usuários interativos.
- *Throughput*: maximiza o número de *jobs* processados por hora.
- *Turnaround*: minimiza o tempo que os usuários em sequência devem esperar por suas saídas.

2.2.4. Algoritmos de escalonamento

Existem os algoritmos preemptivos e os não preemptivos. Os preemptivos são aqueles que permitem que uma tarefa seja interrompida durante sua execução. Após a interrupção dessa tarefa, ocorre a intitulada troca de contexto, que versa em salvar o conteúdo dos registradores, memória utilizada pelo processo e conceder a outra tarefa o privilégio de executar no processador; posteriormente restaurando assim o contexto deste último. Cabe ressaltar que nos algoritmos não preemptivos (utilizados em sistemas monoprocessados) esse fato não ocorre, sendo cada programa é executado até o fim (i.e., sem interrupção, de forma sequencial) [23]. Desta forma, a seguir apresentam-se exemplos de algoritmos de escalonamento que visam atribuição de tarefas aos processadores.

Algoritmo FCFS

O algoritmo FCFS (acrônimo de *First Come First Served*) é uma estrutura de dados que apresenta o seguinte critério: o primeiro elemento a chegar é o primeiro elemento a ser servido. É um algoritmo de escalonamento não preemptivo que entrega as tarefas ao processador pela ordem de chegada. Ele executa a tarefa como um todo do início ao fim não interrompendo a tarefa executada até ser finalizada. Assim, quando uma nova tarefa chega e ainda existe outra em execução, ela vai para uma fila de espera. Portanto, nada mais é do que uma fila que organiza as tarefas que chegam até elas serem atendidas pelo processador [23].

Neste escalonamento todas as tarefas tendem a serem atendidas, evitando o problema de *starvation* (ou inanição), que ocorre quando uma tarefa nunca é executada, pois outras de maior prioridade sempre a impedem de ser executada, a menos que uma tarefa possua um erro ou *loop* infinito. O *loop* infinito parará o computador, pois o FCFS não terá como dar continuidade à execução das tarefas que estão aguardando na fila de espera [24].

Algoritmo garantido

Conforme Tanenbaum [23], o algoritmo de escalonamento garantido assegura às tarefas sua execução, dando a todas elas a mesma quantidade de tempo de execução utilizando o processador. Se acontecer de uma tarefa utilizar menos tempo de execução do que lhe foi destinado, sua prioridade de execução é aumentada. Se outra tarefa utilizou mais, sua prioridade é diminuída.

Uma abordagem diferente para escalonamento é fazer promessas realistas aos usuários sobre o desempenho e, então, conviver com elas. Uma promessa que é realista e fácil de cumprir é que se houver “n” usuários conectados no momento em que se está trabalhando, este recebe aproximadamente “1/n” do poder do processador. De maneira semelhante, em um sistema monousuário com “n” tarefas executando, todas com a mesma prioridade, cada uma delas deve receber “1/n” dos ciclos do processador [23].

Algoritmo RR

O algoritmo de escalonamento RR (acrônimo de *Round Robin*) é um dos mais antigos e simples que atribui frações de tempo para cada tarefa em partes iguais e de forma circular, além de ser imune a problemas de *starvation*. É usado em projetos de SO multitarefa e foi projetado especialmente para sistemas *time-sharing*, pois o RR depende de um temporizador [24].

O funcionamento desse algoritmo inicia-se quando um *quantum* é definido pelo SO para determinar o período de tempo entre cada sinal de interrupção. Todas as tarefas são armazenadas em uma fila circular. Assim, cada tarefa é retirada da primeira posição da fila e o processador a recebe para sua execução. Se a tarefa não termina após o *quantum*, ocorre uma preempção e a tarefa vai para o fim da fila.

Se a tarefa termina antes do período de *quantum*, o processador é liberado para a execução da tarefa que estiver no início da fila. Toda tarefa que chega à fila é inserida no fim da mesma [24].

Algoritmo SJF

O SJF (acrônimo de *Shortest Job First* ou, em português, *job* mais curto primeiro) é um algoritmo de escalonamento que executa, dentre tarefas igualmente importantes, a mais curta por primeiro [23].

O escalonador SJF funciona a partir de um conceito simples, no qual as tarefas menores terão prioridade e, com isso, serão executadas por primeiro. O resultado disso é um tempo médio mínimo de espera para cada conjunto de tarefas a serem executadas. O cálculo de cada tempo médio é feito a partir da próxima alocação do processador. Assim, a tarefa que utilizar o processador por menos tempo será executada por primeiro [23].

2.3. *MULTICORE*

Esta seção fornece uma contextualização sobre a tecnologia *multicore*, apresentando a essência dos seus processadores, seus principais conceitos, além das arquiteturas paralelas e de técnicas para obtenção de desempenho.

2.3.1. Processadores *multicore*

Com o surgimento de dispositivos dedicados a operações de I/O de dados se deram os primeiros usos da programação concorrente no final da década de 60. No início, o uso da concorrência era para melhor aproveitar o tempo de processamento, pois não havia necessidade do processador permanecer ocioso (e.g. aguardar término de requisições), uma vez que dispositivos especializados responsabilizavam-se por executar operações de I/O [22].

A tática utilizada permitia a execução de fluxos de instruções de diferentes *softwares*, sobrepondo o fluxo de execução de um *software* com operações de I/O de outros. Uma das táticas, o compartilhamento de tempo, ainda é uma técnica utilizada. Porém, atualmente, o campo de aplicação da programação

concorrente é vasto, permitindo sua utilização em quase todos os recursos computacionais com apoio de diferentes ferramentas de programação.

Os processadores *multicore* refletem os um dos mais recentes avanços da arquitetura de computadores nos últimos tempos, aproximando o poder do processamento paralelo aos computadores domésticos. A arquitetura básica dos processadores *multicore* baseiam-se na arquitetura SMP (*Symmetric MultiProcessors*). Nessa arquitetura os diferentes núcleos são capazes de executar fluxos de instruções de forma independente e compartilham uma área de memória comum [25].

Um processador é dito *multicore* quando possui dois ou mais núcleos de processamento (i.e. processadores) no mesmo *chip* (que trocam informações via uma memória compartilhada, ou seja, arquitetura SMP) [25]. Essa forma de TLP (*Thread Level Parallelism*) é também comumente chamada de CMP (*Chip-level MultiProcessor*) [26]. *Multicore* é um nome popular ao CMP.

Processadores *multicore* possuem várias configurações (e.g. *dualcore*, *quadcore*). Algumas dessas são *multithreaded*. As abordagens de disposição e comunicação entre os processadores variam entre diferentes implementações. A seguir são apresentadas as três configurações mais comuns que utilizam o conceito do multiprocessamento.

A Figura 9 ilustra a configuração de um processador *hyperthreaded*. Essa configuração permite que duas ou mais *threads* executem em um único *chip*. Nessa os processadores são lógicos e não físicos. Logo, um processador *hyperthreaded* se apresenta ao SO como múltiplos processadores, quando na verdade há um único processador executando múltiplas *threads*.

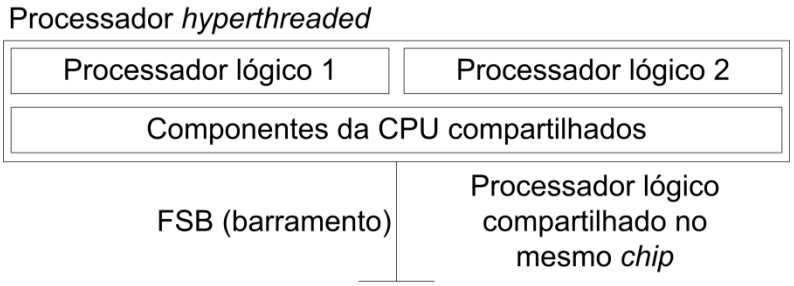

Figura 9. Processador *hyperthreaded*

Fonte: Adaptado de [27]

A Figura 10 ilustra a configuração de um multiprocessador clássico. Nessa cada processador está em um *chip* separado com seu próprio *hardware*.

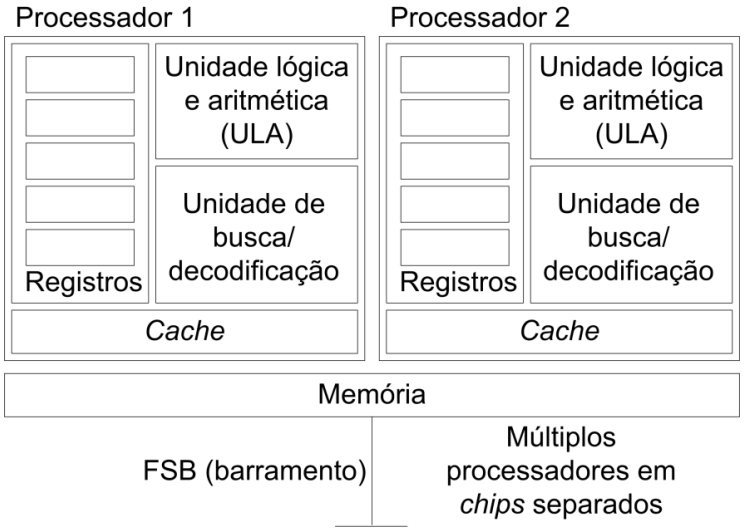

Figura 10. Multiprocessador clássico

Fonte: Adaptado de [27]

A Figura 11 ilustra a configuração atual dos multiprocessadores. Nessa os múltiplos processadores estão no mesmo *chip* e compartilham a mesma memória.

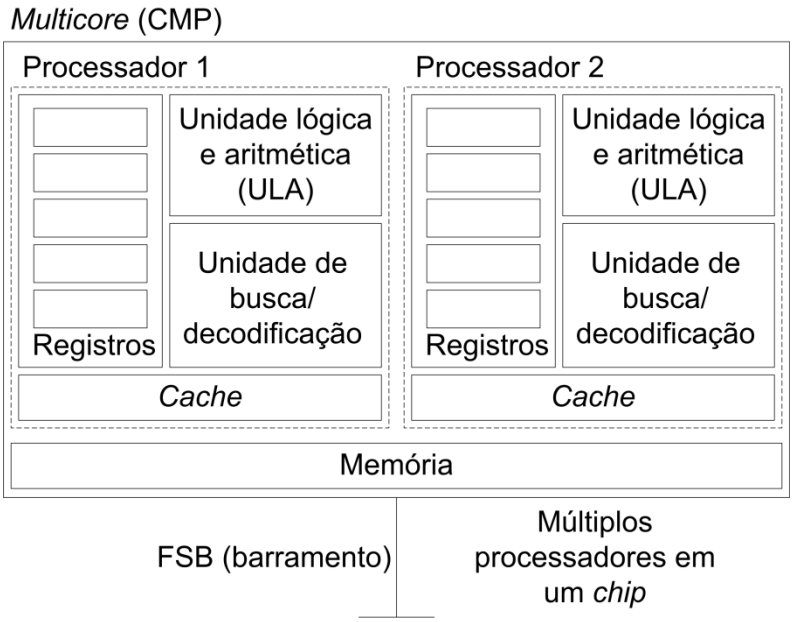

Figura 11. Arquitetura *multicore* (CMP)

Fonte: Adaptado de [27]

A programação *multicore* oferece recursos com os quais um *software* pode ser descrito em termos de diversos fluxos de execução (i.e. *threads*), capazes de executar de forma concorrente. Desse modo, um programa *multithreaded* possui diversos fluxos de execução que podem estar ativos em um determinado instante de tempo e, a exemplo de programas imperativos, cada fluxo possui uma área de endereçamento própria [1].

A diferença é que, além desse espaço de memória privado, as *threads* compartilham acesso a um espaço de endereçamento global. A concorrência, nesse caso, expressa não somente a disputa de recursos de processamento, como espaço de memória e tempo de processador, mas também a disputa das *threads* pelo acesso aos dados armazenados em memória.

A relação entre programação *multicore* e arquiteturas SMP é quase que direta. Enquanto o modelo da arquitetura física prevê a existência de mais de um processador e uma memória comum, o modelo de programa *multithreaded* prevê a coexistência de fluxos de execução, que podem usar um espaço de endereçamento compartilhado para comunicar dados entre si. Essa relação é explorada por algumas ferramentas de programação, as quais oferecem serviços para controle do número de *threads* ativas simultaneamente e para sincronizar as ações dessas no acesso aos dados compartilhados [1].

Com a popularização dos processadores *multicore*, a demanda por programação concorrente atingiu novos patamares, uma vez que usuários domésticos possuem arquiteturas paralelas à sua disposição. Como consequência, o desenvolvimento de novos *softwares* (e.g. elaborados *softwares* científicos, jogos) deve incorporar técnicas de programação *multicore*.

2.3.2. Arquiteturas paralelas

Arquiteturas *multicore* e arquiteturas paralelas estão intrinsecamente ligadas. Dessa forma, após a introdução sobre *multicore*, um embasamento em arquiteturas paralelas se faz necessário. Usar múltiplos processadores para melhorar desempenho não é recente.

Flynn [28] [29] sugeriu uma taxonomia para classificação de arquiteturas baseando-se no fluxo de instruções e dados observados. Essa taxonomia, ainda empregada, é composta por quatro categorias, que são:

- SISD (*Single Instruction Stream*, *Single Data Stream*): fluxo único de instruções operando sobre um fluxo único de dados. Nessa categoria estão os uniprocessadores.

- SIMD (*Single Instruction Stream*, *Multiple Data Stream*): uma mesma instrução pode operar sobre um conjunto de dados. Nessa categoria estão os computadores que exploram o paralelismo de dados (e.g. processadores multimídia, processadores vetoriais ou matriciais).

- MISD (*Multiple Instruction Stream*, *Single Data Stream*): múltiplos fluxos de instrução operando sobre um único fluxo de dados. Segundo Hennessy e Patterson [30] não existem exemplares dessa categoria no mercado; entretanto alguns autores classificam arquiteturas *pipelined* como MISD.

- MIMD (*Multiple Instruction Stream*, *Multiple Data Stream*): cada processador executa um fluxo de instruções independente que opera sobre um conjunto de dados particular. Computadores MIMD exploram o TLP.

As arquiteturas MIMD podem ainda ser classificadas em três categorias [1]. Essas subdivisões não fazem parte da taxonomia original proposta por Flynn. São elas:

- SMP: arquitetura baseada em múltiplos processadores que compartilham uma mesma memória. Também chamada de UMA (*Uniform Memory Access*; ou Multiprocessador; ou ainda MIMD Fortemente Acoplada). Nessa categoria, a memória compartilhada é centralizada.

- NUMA (*Non-Uniform Memory Access*): arquitetura baseada na existência de memórias dedicadas para cada processador ou grupo de processadores, conectados à memória por meio de um *crossbar* ou *switch* e uma área de memória compartilhada distribuída, intitulada DSM (*Distributed Shared Memory*). Essas arquiteturas podem ser consideradas fortemente acopladas[2].

- *Cluster*: arquitetura baseada em múltiplos computadores que trocam dados por meio de troca de mensagens via uma rede de interconexão,

[2] A memória dedicada tem como finalidade reduzir as latências de acesso aos dados para cada processador, enquanto a memória compartilhada permite que diferentes fluxos de instruções (i.e. processadores) possam acessar uma área comum de endereçamento.

intitulada MP (*Message Passing*); também chamada de multicomputador (ou MIMD Fracamente Acoplada).

É importante salientar que o paralelismo pode ser explorado em diferentes níveis e em praticamente todas as arquiteturas. Ainda, as formas de exploração de paralelismo podem ser combinadas (e.g. um *cluster* pode ser implementado com base em processadores superescalares) [31].

A exploração do paralelismo foi foco das otimizações na maioria dos microprocessadores durante muito tempo; isso graças ao aumento de desempenho obtido no *software* de maneira transparente (i.e. sem que esse precisasse ser modificado). Porém, essa técnica atingiu níveis de complexidade elevados e não apresenta mais os mesmos retornos em termos de desempenho [32].

Apresentadas então as arquiteturas paralelas, será agora abordado um compêndio das técnicas para obtenção de desempenho das arquiteturas *multicore*.

Técnicas para obtenção de desempenho

Devido à exploração do paralelismo de instrução em uma única *thread* de controle, resultados satisfatórios de desempenho foram conseguidos. Tais resultados foram alcançados com a aplicação de diversas técnicas (e.g. execução fora de ordem, microarquiteturais, entre outras) [1].

Estudos atuais e a própria indústria balizaram que esse caminho ficou saturado [30] [33] [34]. Igualmente, consumo e altas latências de acesso à memória têm se mostrado obstáculos difíceis de ser transpostos, levando em consideração o desempenho de uma única *thread*.

Foram discutidas durante anos, alternativas e técnicas para aumento de desempenho de processadores e, no final da década de 90, parecia existir uma concordância de que o foco das otimizações deveria voltar-se para a exploração do paralelismo de *threads* [35] [36] [37].

Naquela época diversas implementações de processadores já estavam no mercado (e.g. Pentium III, AMD K6-III). Tais processadores implementavam as complexas estruturas necessárias para explorar o paralelismo no nível de instrução, mas, em eficiência, estavam muito abaixo do que poderiam alcançar. Em outras palavras, o volume de *hardware* existente fazia com que esses processadores não desempenhassem seu papel em relação ao desempenho ideal.

Uma forma de aumentar a eficiência seria criar uma lógica capaz de gerenciar instruções oriundas de várias *threads* em execução. Um dos primeiros trabalhos foi realizado por Donalson *et al* [38] e posteriormente avançado por Tullsen, Eggers e Levy [26], que foram quem introduziram o termo SMT (*Simultaneous Multithreaded*).

Um processador SMT pode executar instruções em paralelo; pode também simultaneamente extrair instruções de várias *threads* de controle. Tal modelo chamou a atenção de fabricantes, pois esses tinham conhecimento de que diversos recursos já existentes não estavam sendo usados de forma eficiente, devido às limitações da exploração de paralelismo no interior de uma única *thread*. O que os motivou teve início ao fato de que tais recursos poderiam ser mais bem usados, se existisse outras origens de onde se extraíssem instruções independentes [1].

Entretanto, para implementar um processador SMT, seria necessário garantir que instruções oriundas de uma *thread* não acarretariam nenhum efeito negativo para instruções de outras *threads*. Portanto, a solução envolveu replicar o banco de registradores e marcar as instruções, de forma que os resultados produzidos por instruções de uma *thread* nunca se misturassem aos resultados de instruções das demais. Porém, esse não foi o único problema. O aumento da disponibilidade de instruções originava também uma pressão maior na memória, pois a frequência de acessos a essa era maior [26]. O termo SMT é também conhecido como *HyperThreaded*, conforme ilustração da Figura 19 (Arquitetura *multicore*). Máquinas *HyperThreaded* são comumente referenciadas como HT pelas fabricantes, a exemplo do processor Pentium 4 HT (2 *threads*) da Intel. Tiveram poucas máquinas puramente SMT. Ainda que as implementações SMT/HT suplantassem em parte o problema inicial, a complexidade do processador em si continuava aumentando. Por conta disso, pesquisas relacionadas às arquiteturas *multicore* iniciaram-se.

2.4. RECURSOS DE PROGRAMAÇÃO PARA AMBIENTE *MULTICORE*

Esta seção apresenta alguns dos recursos de programação existentes para ambiente *multicore*. São eles a API POSIX *Threads* (*PThreads*), a linguagem

de programação de propósito geral *Cilk*, a biblioteca *Threading Building Blocks* (TBB), além dos *pragmas* de compilador *Open MultiProcessing* (*OpenMP*) e CUDA.

2.4.1. API POSIX *Threads*

PThreads é um padrão POSIX para *threads* que define uma API para criação e manipulação de *threads*. POSIX é acrônimo de *Portable Operating System Interface* que, em português, significa Interface Portável entre Sistemas Operacionais. O “X” representa a herança que a interface tem do sistema UNIX. POSIX é uma família de normas definidas pelo IEEE que tem como objetivo garantir a portabilidade do código-fonte de um programa a partir de um SO que atenda as normas POSIX para outro sistema POSIX. Desta forma, as bibliotecas da API atuam como uma interface entre SOs distintos [39].

O controle das *threads* é feito pelo desenvolvedor, que precisa saber gerenciar o código para evitar programações incorretas (e.g. *deadlocks*). A biblioteca oferece recursos de controle e sincronização das *threads*. Entretanto, esse controle e sincronização, bem como o balanceamento de carga, disponibilizados pela *PThreads*, devem ser são realizados de forma manual pelo desenvolvedor [40].

2.4.2. Linguagem de programação de propósito geral *Cilk*

Cilk é uma linguagem de programação de propósito geral desenvolvida pelo *Supertech Research Group* do *Massachusetts Institute of Technology* (MIT). Baseia-se na linguagem de programação C (ANSI), possuindo versões para os SOs Linux, MacOS X e Windows [41].

Cilk apresenta um sistema de execução que se encarrega de fazer o balanceamento de carga e de escalonar as tarefas criadas para que essas executem em paralelo entre os processadores. O problema do balanceamento de carga é que o escalonador de tarefas penaliza o processamento a cada criação de uma nova *thread* [41].

2.4.3. Biblioteca *Threading Building Blocks*

Threading Building Blocks (TBB) é uma biblioteca para a linguagem de programação C++ concebida pela Intel para o desenvolvimento de *softwares* que visam à utilização do ambiente *multicore*. Sua versão comercial suporta os SOs Linux, MacOS X e Windows [42].

A divisão e o escalonamento de tarefas realizado por meio de *threads* foi implementado de modo a maximizar a utilização do processador. O intuito da TBB é tornar a concepção de *software* paralelo mais fácil e ágil, tirando a responsabilidade do desenvolvedor na programação das *threads*. Ou seja, a biblioteca faz isso de forma transparente, pois, normalmente, isso demandaria muito tempo na construção dos *softwares*. Uma desvantagem no seu uso é o número fixo de *threads* definidas no código [42].

2.4.4. *Pragma* de compilador *Open MultiProcessing*

OpenMP admite a paralelização de código sequencial. *Threads* são concebidas para a execução concorrente de códigos que foram indicados para serem executados de forma paralela. Para que isso ocorra o compilador deve suportar para *pragmas* [43].

OpenMP só pode ser utilizado em ambientes com memória compartilhada, sendo uma de suas desvantagens. Sua simplicidade faz com que seja frequentemente utilizado, especialmente, em laços iterativos [43].

2.4.5. *Pragma* de compilador CUDA

CUDA é uma arquitetura de computação paralela e também um *pragma* de compilador que possibilita aumentos significativos no desempenho de computação pelo aproveitamento da potência da GPU (*Graphics Processing Unit*) [44].

A utilização da GPU viabiliza o desenvolvimento de *softwares* capazes de utilizar a arquitetura para a programação de alto desempenho das placas de vídeo. Essa tecnologia está disponível nas linhas *GeForce*, especialmente para computação de alta performance [44].

Sua desvantagem está no fato de não suportar funções recursivas, além de apresentar gargalo na comunicação entre a CPU e a GPU, quando implementações que fazem uso de ambos os processadores são desenvolvidas [44].

2.5. *EMBEDDED PARALLEL OPERATING SYSTEM*

O *Embedded Parallel Operating System* (EPOS) é um SO paralelo para plataformas embarcadas baseado em componentes que possui uma arquitetura escalável moldada para suportar necessidades que nele são executadas [45]. É um *framework* para geração de sistemas que provê suporte a *softwares* dedicados e independentes de plataforma [46]. Ele implementa aspectos por meio de adaptadores de cenários [47], que quando combinados com outros componentes do sistema, formam diferentes arquiteturas de *software*.

Dessa forma, implementa um *framework* que define como os componentes são organizados de forma a gerar um sistema funcional. Está materializado na linguagem de programação C++, sendo executado na compilação do sistema, adaptando os componentes selecionados para coexistir entre eles, além dos aspectos e do(s) *software(s)*, conforme ilustra a Figura 12.

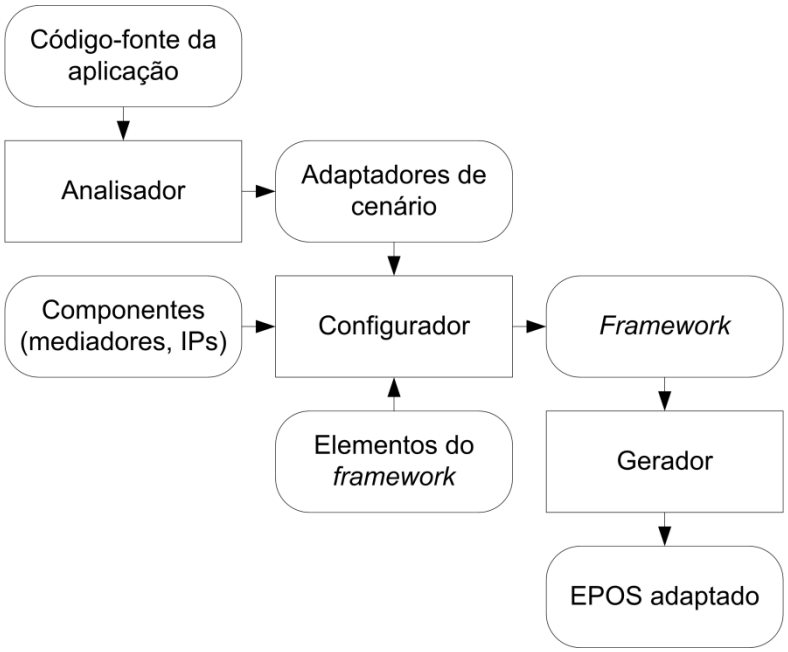

Figura 12. Cadeia de ferramentas do EPOS

Fonte: Adaptado de [45] [47]

Originados posteriormente a um processo de engenharia de domínio[3], os componentes formam a base e suporte necessário ao sistema, tanto em *software* quanto em *hardware*. Tal engenharia de domínio consiste em um desenvolvimento sistemático de um modelo e sua implementação [48].

O EPOS é indicado para projetos de sistemas embarcados de baixo custo, com suporte para diferentes arquiteturas de *hardware* que podem variar desde microcontroladores de 8 *bits* até arquiteturas de 32 *bits* (e.g. IA32).

Projetos de Sistemas Embarcados Dirigidos à Aplicação (*Application-Driven Embedded System Design* ou ADESD) propõem analisar o domínio do problema, identificando as entidades mais significativas e organizando-as em famílias de abstrações. Tais interfaces criam uma visão de cada família enquanto adaptadores de cenário ajustam abstrações a dados cenários, simplificando a programação, uma vez que só existe uma interface por família e a abstração que será usada só será escolhida durante a geração do sistema. O EPOS foi concebido adotando-se tal metodologia [45]. Esse suporte (i.e. família de abstrações) advém das diferenças e afinidades identificadas no domínio dos SOs para sistemas embarcados (e.g. gerência de memória, políticas de escalonamento, sincronização, temporização, tratamento de I/O de dados, tratamento de interrupções). Como tais características são configuráveis, certo nível de portabilidade é alcançado, permitindo uma melhor adaptação para diferentes tipos de *hardware* e *software* [48]. Desse modo, o EPOS procura ser o mais completo possível.

Para fazer interface com o *hardware*, o EPOS usa *mediadores de hardware*. Tais mediadores estabelecem um contrato de interface entre componentes de *hardware* e *software*. Essa camada de abstração de *hardware* procura não gerar código desnecessário. Na sua maior parte são meta-programados, portanto dissolvem-se dentro dos componentes, o que resulta em baixo ou nenhum sobrecusto.

Mediadores exportam as funcionalidades necessárias para as abstrações do sistema de nível mais alto, por meio do contrato de interface. Tais abstrações representam os serviços dos SOs citados anteriormente [49]. No EPOS, abstrações

[3] A engenharia de domínio tem como objetivo identificar, construir, catalogar e disseminar um conjunto de componentes de *software* que tenham aplicabilidade em *softwares* existentes e futuros, dentro de um domínio de aplicação específico. Um domínio de aplicação é como uma família de produtos (aplicações com funcionalidades similares). O objetivo é estabelecer um mecanismo em que engenheiros de *software* possam partilhar tais componentes utilizando-os em sistemas futuros.

são projetadas e implementadas de modo independente de seus cenários de execução e arquiteturas. As unidades de *hardware* dependentes de arquitetura são abstraídas como mediadores.

Graças ao uso de meta-programação estática, mediadores implementam suas funcionalidades sem formar uma Camada de Abstração de *Hardware* (*Hardware Abstraction Layer* ou HAL) tradicional [45].

Processos são gerenciados pelas abstrações *Thread* e *Task* no EPOS. Cada *thread* armazena seu contexto em sua própria pilha. A abstração de contexto define o conjunto de dados que precisa ser armazenado para um fluxo de execução e, dessa forma, cada arquitetura define seu próprio contexto.

Tempo é tratado pela família de abstrações *Timepiece*. Tais abstrações são suportadas por meio dos mediadores *Timer*, *Timestamp Counter* (TSC) e *Real-Time Clock* (RTC). A abstração *Clock* é responsável por um controle estrito de tempo e está disponível em sistemas que possuam um dispositivo de RTC. A abstração *Alarm* pode ser usada para gerar eventos que acordem uma *thread* ou chamem uma função. Alarmes têm ainda um evento principal de alta prioridade que está associado com um período de tempo pré-definido. Esse é usado para acionar o algoritmo de escalonamento do sistema quando o tempo limite de execução de um processo (i.e. *quantum* de escalonamento) é atingido, nos casos em que uma configuração com um escalonador ativo é usada. A abstração *Chronometer* é usada para realizar operações de medição de tempo [45].

A família de abstrações *Synchronizer* fornece mecanismos que garantem a consistência de dados em ambientes com processos concorrentes. A abstração *Mutex* implementa um mecanismo de exclusão mútua que entrega duas operações atômicas (*lock* e *unlock*). A abstração *Semaphore* implementa um semáforo, que é uma variável inteira cujo valor apenas pode ser manipulado indiretamente por meio de operações atômicas. A abstração *Condition* implementa o conceito de variável de condição, que permite a uma *thread* esperar que um predicado se torne válido.

Detalhes referentes à proteção e tradução de espaços de endereçamento, assim como alocação de memória, são abstraídas por meio da família de mediadores de *hardware Memory Management Unit* (MMU). A abstração *Address Space* é um contêiner para “fatias” de memória física chamados de *Segments*. Ela não implementa tarefas de proteção, tradução ou alocação de endereços de memória, deixando esse papel para o mediador da MMU. A abstração *Flat Address*

Space define um modelo de memória no qual endereços lógicos e físicos coincidem, eliminando a necessidade de *hardware* para MMU. Em plataformas que não dispõem de MMU, um mediador para MMU simplesmente mantém o contrato de interface com o *Flat Address Space*, realizando implementações vazias de métodos quando necessário. Métodos relativos à alocação de memória operam sobre *bytes* de modo similar ao que é feito pela função de alocação de memória da *libc* [45].

Controles de I/O de dispositivos periféricos são disponibilizados no EPOS pelo mediador de *hardware* correspondente. O mediador *Machine* armazena as regiões de I/O e trata o registro dinâmico de interrupções. O mediador *Interrupt Controller* trata a ativação ou desativação de interrupções individuais. Para tratar as diversas interrupções existentes em diferentes plataformas e contextos, o EPOS atribui um nome e uma sintaxe independente de plataforma às interrupções pertinentes ao sistema (e.g. interrupção de *timer*).

2.6. CONCLUSÕES DO CAPÍTULO

Em relação ao PON, pode-se dizer que os paradigmas e linguagens de programação ditos convencionais não apresentam facilidades nos seus usos na computação paralela e distribuída, graças ao alto acoplamento entre as partes do *software*. O uso do PON para a resolução desse problema certamente facilitará o paralelismo e distribuição. Uma das principais deficiências dos paradigmas convencionais está relacionada aos seus mecanismos de execução no tocante aos “elementos passivos”. Buscas excessivas afetam o desempenho das aplicações inviabilizando a implementação de alguns *softwares* sobre determinados *hardwares* disponíveis. O PON, por natureza, se mostra favorável na construção de *softwares* multiprocessados. Porém, nesses ambientes, ainda deve ser mais bem estudado, por exemplo, para criar estratégias de balanceamento de carga, sendo o objetivo desta tese.

Em relação aos meios de organização da execução de *software*, pode-se dizer que ainda há confusão em relação aos conceitos de tarefa e processo, sobretudo porque os SOs mais antigos, suportavam somente uma tarefa para cada processo. Algumas referências mantém isso até os dias de hoje. No entanto, os SOs contemporâneos suportam mais de uma tarefa por processo (e.g. Windows XP e superiores). Assim, hoje em dia o processo deve ser visto como um contêiner de

recursos utilizados por uma ou mais tarefas para sua execução e pela própria gerência de tarefas, impedindo que uma tarefa em execução em um determinado processo acesse um recurso atribuído a outro processo. Ainda, o escalonamento de tarefas realizado pelos SOs divide a carga de processamento das aplicações.

Em relação ao *multicore*, pode-se dizer que sua arquitetura baseia-se na SMP, na qual os diferentes núcleos executam fluxos de instruções independentemente, compartilhando uma mesma memória. As *threads*, além dessa memória, compartilham acesso a um espaço de endereçamento global. A classificação MIMD, na qual cada processador executa um fluxo de instruções independente que opera sobre um único conjunto de dados, foco de estudo e aplicação desta tese, engloba a arquitetura SMP, que se baseia em múltiplos processadores que compartilham uma mesma memória. Ou seja, a memória compartilhada é centralizada.

Em relação aos recursos de programação para ambiente *multicore*, foram vistas pontualmente as principais características da API *PThreads*, da linguagem de programação de propósito geral *Cilk*, da biblioteca TBB, além dos pragmas de compilador *OpenMP* e CUDA. Realmente, tais recursos amenizam o trabalho do desenvolvedor, entretanto, apresentam desvantagens conforme apontado em cada recurso.

Em relação ao EPOS, pode-se dizer que ele é um SO paralelo para plataformas embarcadas baseado em componentes. É um *framework* para geração de sistemas que provê suporte a *softwares* dedicados e independentes de plataforma. Nele os processos são gerenciados pelas abstrações *Thread* e *Task*.

3. MÉTODO PROPOSTO

Este capítulo descreve os esforços desta pesquisa, a qual visa contribuir para a distribuição da carga de trabalho de *software* por meio da proposta de um método para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore,* contribuindo para o melhor aproveitamento da capacidade de processamento do *hardware* disponível [50]. Desta forma, neste dado contexto, a dissociação entre os componentes de *software* fornecidos naturalmente pelo PON se constitui na chave para permitir a distribuição da carga de trabalho, sendo tal tema explorado no decorrer deste capítulo.

3.1. DELIMITAÇÃO DO ESCOPO

A área de multiprocessamento é bastante ampla, tendo sua origem datada no final da década de 60 com os primeiros usos da programação concorrente. Atualmente, avanços significativos aconteceram na indústria dos computadores como os processadores *multicore*. Devido a essas pesquisas, surgiram diversas arquiteturas visando o melhor uso do poder de processamento de tais processadores *multicore* [1]. Com isso, devido aos diferentes tipos de arquiteturas e suas classificações, conforme seção 2.3, faz-se necessário uma delimitação de escopo para essa pesquisa.

Em relação à arquitetura de *hardware multicore*, a arquitetura de processadores definida foi a SMP (acrônimo de *Symmetric MultiProcessors*). Conforme seção 2.3, geralmente, a arquitetura básica dos processadores *multicore* baseiam-se na arquitetura SMP. Na arquitetura SMP, os diversos núcleos do processador são idênticos em arquitetura e viabilizam o TLP (acrônimo de *Thread Level Parallelism*) na medida em que podem executar diferentes *threads* concorrentemente. Dado que cada *thread* pode ter uma lógica computacional diferente e atuar sobre um conjunto diferente de dados, é possível enquadrar a arquitetura SMP na classificação MIMD (acrônimo de *Multiple Instruction Stream*, *Multiple Data Stream*). Na classificação MIMD cada processador executa um fluxo de instruções independente que opera sobre um conjunto de dados particular e computadores com essa arquitetura exploram o TLP [25] [26] [27] [28] [29].

Em relação à arquitetura de *software*, definiu-se primeiramente o SO. Optou-se pelo EPOS (acrônimo de *Embedded Parallel Operating System*) como plataforma para ensaios desta tese. Esse SO é paralelo e útil para plataformas embarcadas que possui arquitetura escalável, além de ser um *framework* para geração de sistemas com suporte a *softwares* dedicados. Ele foi construído usando-se a linguagem de programação C++.

O EPOS é indicado para projetos de sistemas embarcados com suporte para diferentes arquiteturas de *hardware* e pode ser emulado (por meio do QEMU no Linux [51]). Ainda, o EPOS suporta uma família de abstrações de domínio dos SOs, que são: gerência de memória, políticas de escalonamento, sincronização, temporização, tratamento de I/O de dados e tratamento de interrupções.

Quando emulado, abstrações no EPOS são projetadas e implementadas imitando seus cenários de execução e arquiteturas. Unidades de *hardware* dependentes de arquitetura são abstraídas como mediadores.

Diferentemente dos SOs convencionais, o EPOS trabalha apenas com um processo e esse pode conter várias *threads*. Desta maneira, sua escolha como plataforma para ensaios deu-se por alguns motivos, tais como:

- Foi construído na linguagem de programação C++, como o PON, o que o aproxima da realidade estudada nessa tese.
- Pode ser emulado. Com isso, não são necessários investimentos financeiros em *hardware*. Além disso, diferentes cenários podem ser configurados e simulados, enriquecendo essa pesquisa.
- Provê suporte ao que parece ser necessário aos ensaios dessa tese, em se tratando da contribuição em ambiente *multicore* (e.g. gerência de memória, políticas de escalonamento).
- O processo único do SO é composto por *threads* e, em se tratando da distribuição das entidades PON em *multicore*, essas também se darão por meio de *threads*, que são uma forma natural de distribuição.
- O EPOS é um SO que foi desenvolvido e é mantido pela comunidade acadêmica brasileira [51]. Assim, utilizá-lo é uma forma de valorização da pesquisa nacional, estreitamento de trabalhos correlacionados, validação do *framework* e uma maneira de dar *feedback* a tal estudo por meio dos ensaios realizados.

Em resumo, a arquitetura de *hardware* definida para o escopo desse trabalho foi a *multicore*. Tal arquitetura de processadores é a SMP que é uma forma de TLP. Ainda, SMP enquadra-se na arquitetura paralela MIMD.

Em relação à arquitetura de *software*, a plataforma (i.e. SO) para ensaios desse trabalho será o EPOS. Assim, o método será operacionalizado por meio de *threads*, pois, devido ao fato da memória ser compartilhada, o paralelismo em nível de *threads* como uma extensão do *Framework* PON apresenta-se como o mais adequado.

3.2. DESCRIÇÃO DO MÉTODO PROPOSTO

Pretende-se, por meio do PON, facilitar a distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*. Para isso, propõe-se um método que contribuirá para o melhor aproveitamento da capacidade de processamento do *hardware* disponível. Tal método foi organizado em cinco etapas, que são: “1. Alocação inicial da aplicação PON”, “2. Monitoramento da Carga de Trabalho”, “3. Análise dinâmica de *clusters*”, “4. Balanceamento de Carga de Trabalho” e “5. Realocação da aplicação PON”, conforme ilustra a Figura 13.

As cinco etapas do método proposto foram divididas em duas fases, que são: “Análise e alocação estática” e “Análise e alocação dinâmica”, conforme ilustrado na Figura 15. Na “Análise e alocação estática”, a atribuição de *threads* aos núcleos é realizada antes do início da execução do *software*, sendo necessário que se tenha informações acerca do custo de processamento das *threads* e dos núcleos disponíveis em tempo de compilação. Na “Análise e alocação dinâmica”, tais informações (custo de processamento das *threads* e dos núcleos disponíveis) podem ser obtidas em tempo de execução. As subseções a seguir descrevem cada uma das etapas dessas fases.

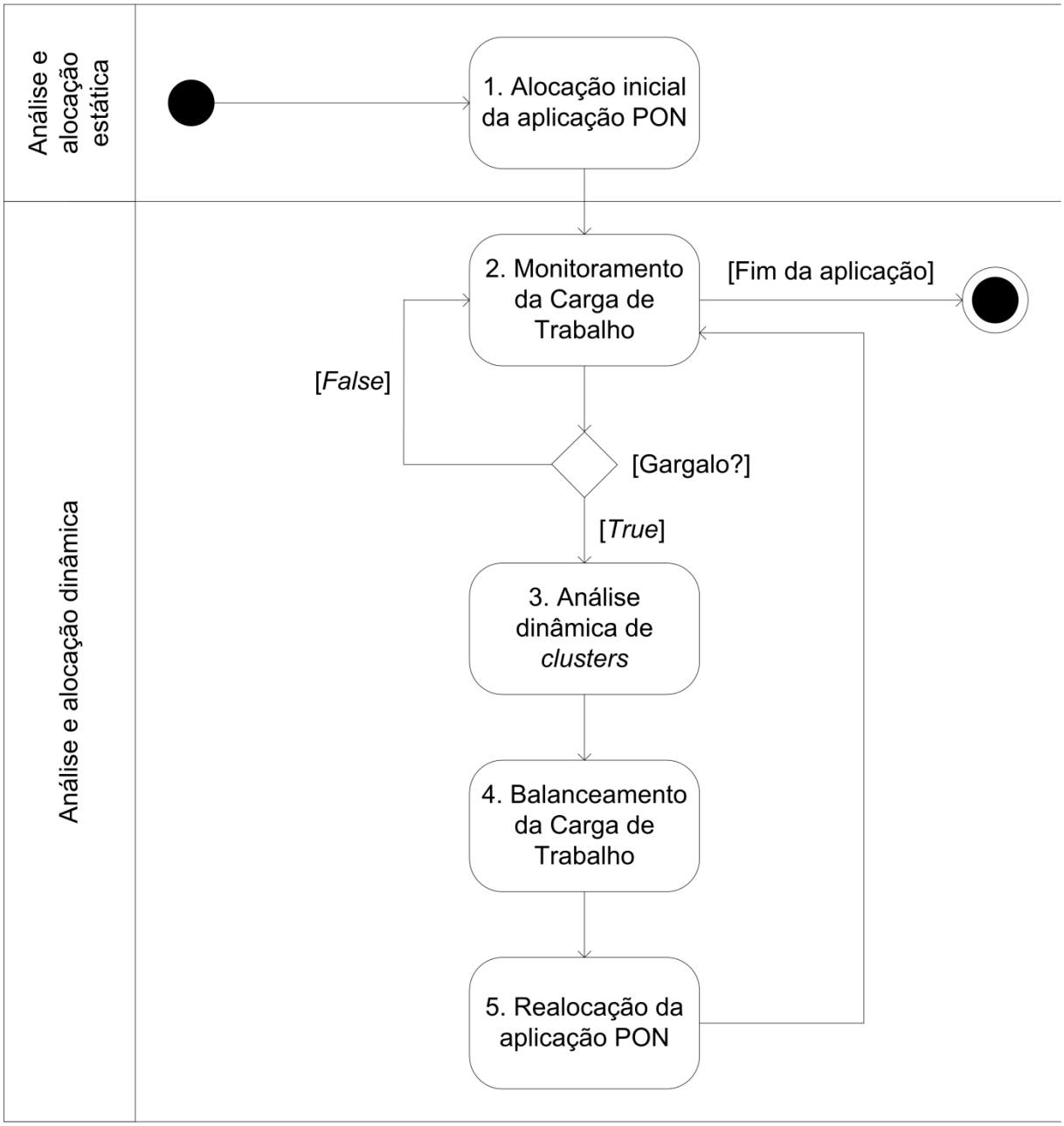

Figura 13. Visão geral do método

3.2.1. Alocação inicial da aplicação PON

Essa etapa é responsável por distribuir o *software* PON (i.e. as partes do *software* PON) aos núcleos do processador para que ele seja executado. De acordo com as dependências e índices de acoplamento entre as entidades PON, pode-se estabelecer grupos (i.e. *clusters*) de regras e alocá-las a diferentes núcleos.

Com o intuito de facilitar a identificação e o levantamento das regras de uma determinada aplicação PON, essa etapa do método inicia-se com a definição

de casos de uso (*use cases*). Sabe-se que um caso de uso (i.e. serviço) representa uma utilização do sistema, descrevendo como um usuário interage com o sistema e definindo os passos necessários para atingir um objetivo específico. Variações na sequência de passos descrevem vários cenários. Segundo Pressman [52], um diagrama de caso de uso UML é uma visão geral de todos os casos de uso e como estão relacionados. Portanto, tal diagrama fornece uma visão geral da funcionalidade do sistema.

Cada caso de uso provido pela aplicação tem uma colaboração para realizá-lo e, da mesma forma, cada colaboração pode ser "composta" pela execução de um conjunto de regras PON (i.e. *Rules*) que realiza essa colaboração. Com isso, a etapa "1. Alocação inicial da aplicação PON" envolve as seguintes atividades.

- **Atividade *i*. Levantar os casos de uso que compõem a aplicação e estimar a frequência de uso de cada *Use Case* (UC), empiricamente**.

Essa estimativa é um parâmetro configurável para cada UC (e.g. UC1 tem uma estimativa de frequência de uso de uma vez a cada 1000ms). A título de exemplo, suponha-se que para um determinado subsistema foram identificados 3 (três) UCs, os quais apresentam as seguintes frequências de uso (realização da atividade *i*): UC1=1000ms, UC2=500ms e UC3=100ms.

Para exemplificar as *Rules* (dessa dada aplicação PON), suponha-se a seguinte estrutura de entidades PON, que representam parte de um *software*, conforme ilustra a Figura 14. Tal estrutura é baseada na ilustração da Figura 4 (Colaboração por notificações).

Para facilitar a visibilidade e a legibilidade da ilustração da Figura 14, optou-se em utilizar os dois primeiros caracteres dos nomes das entidades PON (e.g. '*At*' representa a entidade *Attribute*, '*Pr*' representa a entidade *Premise*) seguidos de um número, o que as distinguem umas das outras, mostrando que tais entidades não são as mesmas. A parte em destaque, na cor cinza, representa uma cadeia de notificações. Oportunamente, uma cadeia de notificações representa uma *Rule* e vice-versa. Igualmente, pode-se notar que tal ilustração pode ser representada computacionalmente por meio de um grafo (na verdade, o modelo que tal ilustração representa).

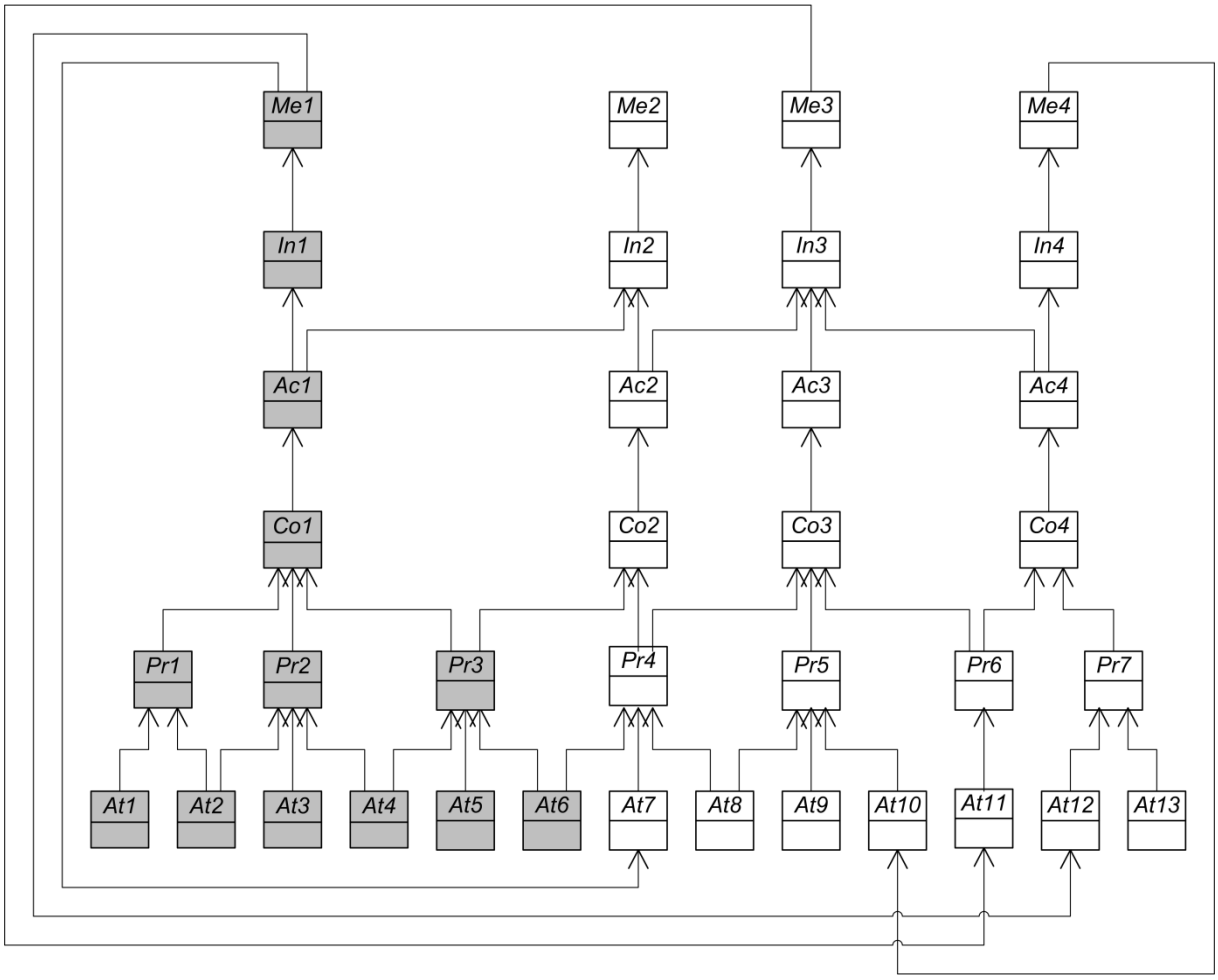

Figura 14. Representação de parte de um *software* PON

- **Atividade *ii*. Enumerar e definir quais *Rules* realizam uma determinada colaboração que, por sua vez, realizam um UC**.

A atividade *ii*, assim como a *i*, deve ser configurável (i.e. deve-se informar as *Rules* que colaboram para as realizações dos UCs). Para realizar a atividade *ii*, primeiramente, é necessário estabelecer conjuntos de entidades PON para cada *Rule* (e.g. *R1*). Tais conjuntos são baseados na ilustração da Figura 14 e são listados a seguir:

- *R1* = {*At1*, *At2*, *At3*, *At4*, *At5*, *At6*, *Pr1*, *Pr2*, *Pr3*, *Co1*, *Ac1*, *In1*, *Me1*}.
- *R2* = {*At4*, *At5*, *At6*, *At7*, *At8*, *Pr3*, *Pr4*, *Co2*, *Ac2*, *In2*, *Me2*}.
- *R3* = {*At6*, *At7*, *At8*, *At9*, *At10*, *At11*, *Pr4*, *Pr5*, *Pr6*, *Co3*, *Ac3*, *In3*, *Me3*}.
- *R4* = {*At11*, *At12*, *At13*, *Pr6*, *Pr7*, *Co4*, *Ac4*, *In4*, *Me4*}.

Informadas as *Rules*, pode-se associá-las aos UCs definidos na atividade *i*. Como na definição dos UCs cada UC tem uma colaboração para realizá-lo e cada

colaboração é realizada por várias *Rules*, a associação das *Rules* aos UCs pode ser exemplificada conforme segue:

- UC1 é realizado pelas *Rules R1* e *R2*.
- UC2 é realizado pelas *Rules R2* e *R3*.
- UC3 é realizado pela *Rule R4*.

Após a realização das atividades *i* e *ii*, continua-se o desenvolvimento das outras atividades da etapa “1. Alocação inicial da aplicação PON”.

- **Atividade *iii*. Descobrir a quantidade de núcleos existentes no processador**.

Para a realização da atividade *iii*, o método utilizará bibliotecas específicas do SO. Suponha-se que, por meio da utilização de alguma dessas bibliotecas, o processador identificado seja um *dualcore*. Considere-se, neste caso, que existem dois núcleos de processamento e, portanto, o parâmetro CN (acrônimo de *Cores Number*) seja igual a 2 (dois).

- **Atividade *iv*. Averiguar a taxa de utilização do processador e, então, alocar o *software* PON**.

Para a realização da verificação da taxa de utilização do processador e, finalmente, alocação do *software* PON, deve-se:

- Levantar a taxa de utilização do processador (i.e. dos núcleos), comumente intitulada taxa de ocupação ou *Occupancy Rate* (OR). A informação da taxa de utilização do processador também se dará por meio de bibliotecas específicas do SO.
- Baseando-se nessa taxa de ocupação, é possível calcular a taxa de disponibilidade ou *Availability Rate* (AR), que é a informação necessária para a alocação da aplicação.
- Alocar a aplicação (*software* PON), propriamente dita, no núcleo com maior taxa de disponibilidade.

Exemplificando, sabe-se que CN = 2, pois se trata de um processador *dualcore*, conforme atividade *iii*.

Suponham-se as taxas de ocupação desses dois núcleos a 80% e 40%, por exemplo. Logo, as taxas de disponibilidade são, respectivamente, 20% e 60%. Desta forma, pode-se notar que a taxa de disponibilidade do núcleo 1 (AR1 = 20%) é de 25%, ou seja, tal núcleo apresenta 1/4 de disponibilidade da sua capacidade de processamento. Da mesma maneira, a taxa de disponibilidade do núcleo 2 (AR2 = 60%) é de 75% (i.e. 3/4 de disponibilidade da sua capacidade).

Com isso, por meio da verificação da taxa de utilização dos dois núcleos do processador, aloca-se o *software* PON no núcleo com maior taxa de disponibilidade, no caso o núcleo 2 e, assim, tem-se a alocação inicial da aplicação.

Nessa primeira etapa não houve distribuição da aplicação PON e sim sua alocação integral a apenas um núcleo. Isso se deve ao fato de não se saber se sua capacidade será ou não suficiente para o processamento da aplicação PON.

3.2.2. Monitoramento da Carga de Trabalho

Nesta etapa inicia-se a análise e alocação dinâmica por meio do “2. Monitoramento da Carga de Trabalho”. Essa atividade de monitoramento será realizada por um módulo de *software* responsável por monitorar dinamicamente as cargas de trabalho do processador, denominado MCT (acrônimo de Monitoramento da Carga de Trabalho).

Em tal monitoramento, caso seja identificado algum gargalo (e.g. algum núcleo esteja sobrecarregado enquanto outro desocupado) em um determinado intervalo de tempo (e.g. considerando um intervalo de monitoramento de 1000ms), avança-se à etapa “3. Análise dinâmica de *clusters*”. Caso contrário, o MCT continua o monitoramento dinâmico das cargas de trabalho do *software* PON que está em execução nos núcleos do processador. Ainda, caso a aplicação seja encerrada, o processo de análise e alocação dinâmica é finalizado.

Essa etapa envolve a seguinte atividade.

- **Atividade *v*. Analisar a evolução da carga de trabalho dos núcleos, verificar a proximidade de gargalo segundo algum limite e produzir a detecção de gargalo**.

O intervalo de tempo de monitoramento da carga de trabalho deverá ser configurável para cada tipo de aplicação (i.e. *software*), podendo ser ajustado automaticamente segundo alguma estratégia, provavelmente empiricamente. A título de exemplo, suponha-se que o intervalo de tempo de monitoramento ou *Time Tracking* (TT) configurado para tal *software* PON foi de 1000ms.

Para a análise da evolução da carga de trabalho dos núcleos serão utilizadas bibliotecas específicas do SO, da mesma forma que na realização da atividade *iv*, verificação da taxa de utilização do processador.

Baseando-se na taxa de ocupação, ao se chegar ao valor de proximidade de gargalo informado (e.g. 95%, valor configurado previamente), do(s) núcleo(s) do processador no qual a aplicação PON esteja alocada, produz-se a sua detecção propriamente dita. A detecção de gargalo nada mais é do que o avanço para a próxima etapa, “3. Análise dinâmica de *clusters*”, objetivando analisar e estabelecer agrupamentos de entidades PON para então dividir a aplicação segundo alguma estratégia.

Nesse estudo considera-se apenas um *software* PON em execução (i.e. “premissa de processo PON único”). De outra forma, seria necessário trabalhar no gerenciamento de múltiplas aplicações, o que ultrapassaria o escopo de estudo dessa pesquisa, mas que será objeto de estudos futuros.

3.2.3. Análise dinâmica de *clusters*

Essa etapa é responsável pela análise e estabelecimento de *clusters* (agrupamentos) de entidades PON, levando em consideração a dependência entre essas entidades e seus respectivos índices de acoplamento.

Por meio da fundamentação matemática e da teoria de grafos, são estabelecidos *clusters* de *Rules* do *software* PON em execução. Assim, essa etapa envolve as seguintes atividades.

- **Atividade *vi*. Determinar os índices de acoplamento interno ou *Internal Coupling* (IC) e acoplamento externo ou *External Coupling* (EC) entre as *Rules***.

Inicialmente, para realizar a etapa “3. Análise dinâmica de *clusters*”, é necessária a criação de uma sociomatriz (i.e. matriz de adjacência), que tem por finalidade o cálculo dos índices de IC e EC. A Figura 15 ilustra parte da sociomatriz gerada com base nas *Rules* (conjuntos de entidades PON) da Figura 14.

	At1	***...***	***At13***	***Pr1***	***Pr2***	***Pr3***	***Pr4***	***Pr5***	***Pr6***	***Pr7***	***...***
At1				*R1*							
At2				*R1*	*R1*						
At3					*R1*						
At4					*R1*	*R1*, *R2*					
At5						*R1*, *R2*					
At6						*R1*, *R2*	*R2*, *R3*				
At7							*R2*, *R3*				
At8							*R2*, *R3*	*R3*			
At9								*R3*			
At10								*R3*			
At11									*R3*, *R4*		
At12										*R4*	
At13										*R4*	
Pr1											
...											

Figura 15. Parte da sociomatriz do grafo da Figura 14

As ligações na sociomatriz dão-se entre entidades PON e elas são representadas por meio do conceito de díades (ligações bidimensionais). Tais díades constituem relações de “notificações” (i.e. possíveis notificações) entre essas entidades.

As células da sociomatriz ilustrada na Figura 15 apresentam as *Rules* que são compostas pelas ligações (possíveis notificações) entre entidades PON. Assim, o IC é medido por meio das díades entre as entidades PON que compõem uma mesma *Rule*.

Para exemplificar, a Figura 16 ilustra a *Rule* 1 (*R1*). As díades entre as entidades PON da *R1* foram numeradas com o objetivo de mostrar quantas possíveis notificações distintas podem ocorrer nessa *Rule*. Posteriormente a Figura 16, apresenta-se a Tabela 1, que exibe o IC por *Rule*.

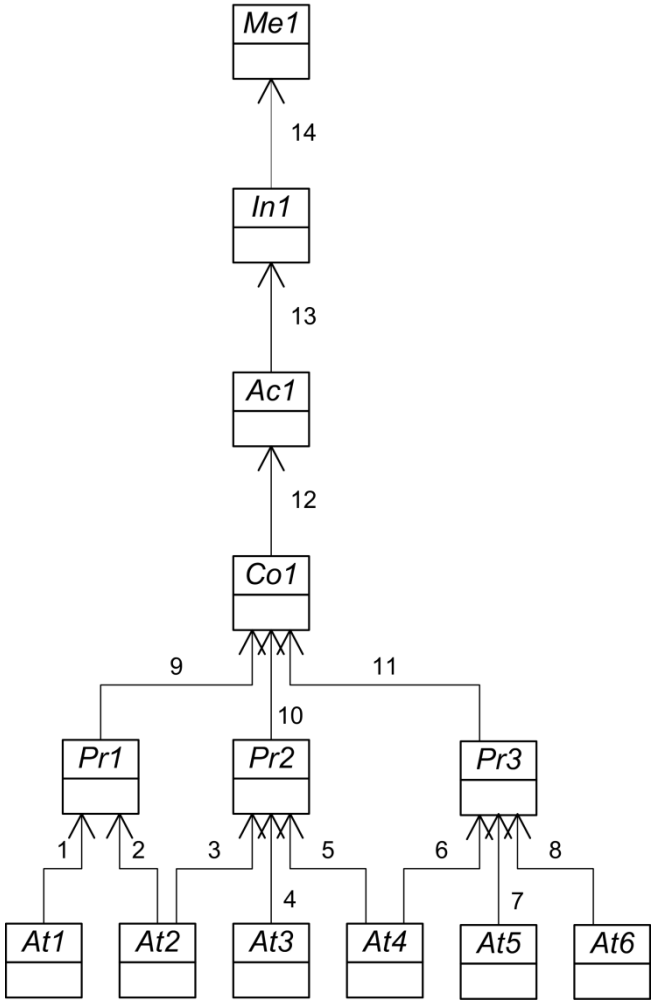

Figura 16. *R1* com suas respectivas díades numeradas

Pode-se notar que na Figura 16 existem 14 díades (i.e. possíveis notificações distintas) entre as entidades PON e, portanto, na Tabela 1, o IC da *R1* é igual a 14.

Tabela 1. IC por *Rule*

Rule	**IC**
R1	14
R2	13
R3	15
R4	10

Por sua vez, o EC é calculado por meio das díades entre entidades PON que não pertencem a uma mesma *Rule*. Além disso, entidades PON que são compartilhadas por duas ou mais *Rules* também são consideradas no cálculo do EC.

Ainda, a estratégia adota pelo método, independentemente da quantidade de núcleos disponíveis no processador é: “sempre que um determinado núcleo apresentar sobrecarga, a parte do *software* PON que nele executa poderá ser dividida em duas partes, estabelecendo assim uma relação entre dois novos *clusters* de *Rules*. Tais *clusters* podem ser compostos por uma ou mais *Rules*”.

Para exemplificar, a Figura 17 ilustra a relação entre *R4* e (*R1*, *R2*, *R3*). Visando facilitar seu entendimento, destacou-se na cor cinza a *R4*. Da mesma forma que anteriormente, as díades entre as entidades PON da relação foram identificadas com o objetivo de mostrar a quantidade de possíveis notificações distintas que podem ocorrer nessa relação. Posteriormente a Figura 17, apresenta-se a Tabela 2, que exibe o EC por *Rules*.

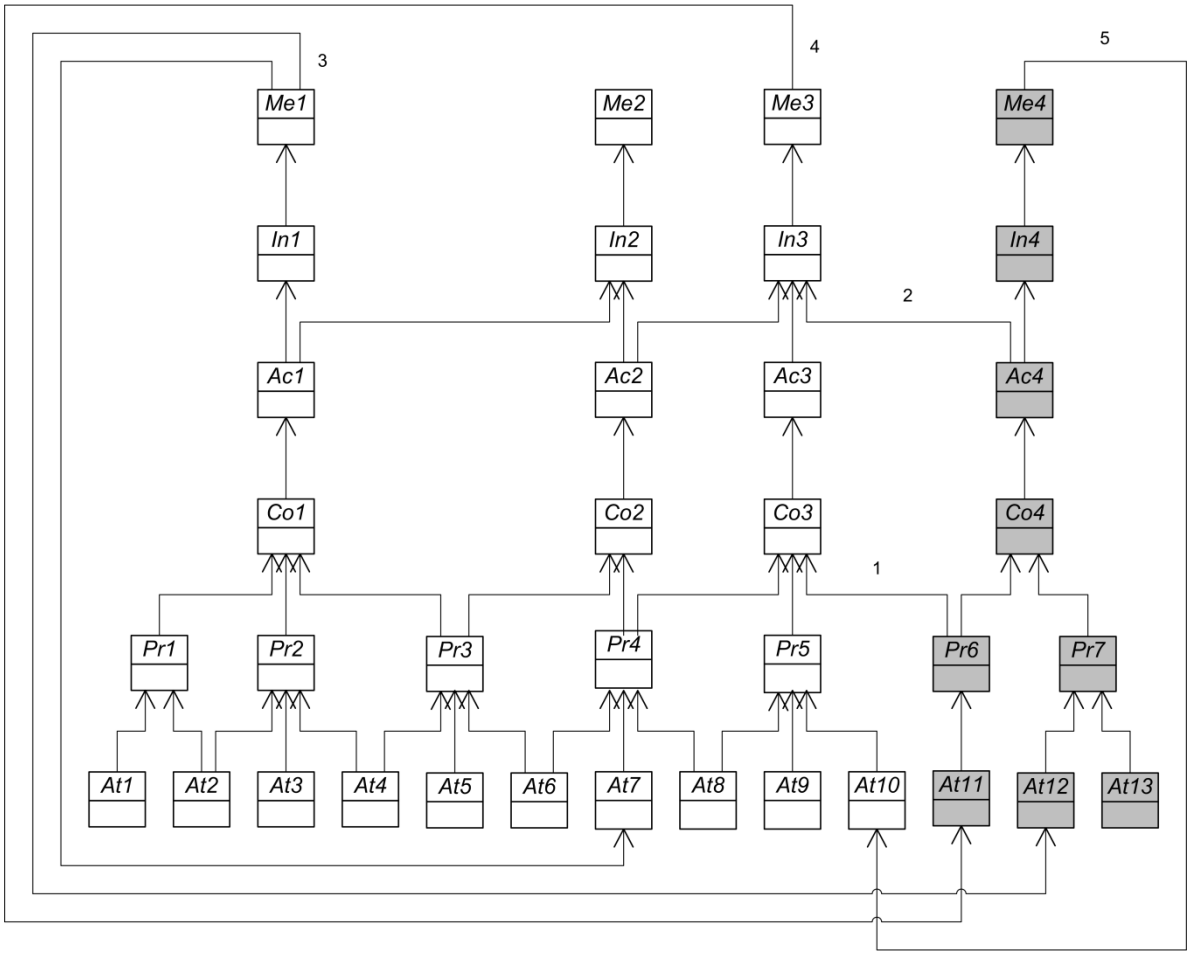

Figura 17. Relação entre *R4* e (*R1*, *R2*, *R3*)

Pode-se notar que na Figura 17 existem 5 díades identificadas entre as entidades PON (sendo duas ligações entre entidades compartilhadas entre *Rules*, díades 1 e 2, e três ligações com entidades externas à *Rule*, díades 3, 4 e 5) e, consequentemente, na Tabela 2, o EC da relação *R4*x(*R1*, *R2*, *R3*) é igual a 5.

Tabela 2. EC por *Rules*

Rules	**EC**
*R1*x(*R2*, *R3*, *R4*)	5
*R2*x(*R1*, *R3*, *R4*)	7
*R3*x(*R1*, *R2*, *R4*)	5
*R4*x(*R1*, *R2*, *R3*)	5
(*R1*, *R2*)x(*R3*, *R4*)	2
(*R1*, *R3*)x(*R2*, *R4*)	12
(*R1*, *R4*)x(*R2*, *R3*)	8
R1, *R2*, *R3*, *R4*	0

- **Atividade *vii*. Examinar a taxa de utilização do processador**.

Para realizar a verificação da taxa de utilização do processador (dinamicamente), deve-se:

- Levantar novamente a OR.

- Por meio da OR calcular a AR.

Suponha-se que as ORs alternaram-se para 40% e 70% (após gargalo detectado na atividade *v*), por exemplo. Logo, as ARs são, respectivamente, 60% e 30%. Desta forma, a taxa de disponibilidade do núcleo 1 é de 66,67% (i.e. 2/3 de disponibilidade) e do núcleo 2 é de 33,33% (i.e. 1/3 de disponibilidade).

Com isso, a próxima etapa do método, após os cálculos dos ICs e ECs e a verificação da taxa de utilização do processador, é adotar uma estratégia para balancear a carga de trabalho.

3.2.4. Balanceamento de Carga de Trabalho

Baseada na etapa “3. Análise dinâmica de *clusters*”, essa etapa é responsável por definir uma estratégia de distribuição do *software* PON. Para tanto, utilizam-se os resultados da análise da carga de trabalho do processador, em tempo de execução.

Com isso, por meio do Módulo de Distribuição (MD), essa etapa é responsável por decidir em quais núcleos as entidades PON serão executadas. O MD é o módulo de *software* responsável pela aplicação de um algoritmo de balanceamento de carga. Assim sendo, o MD utilizará uma estratégia para a distribuição da aplicação. Essa etapa envolve as seguintes atividades.

- **Atividade *viii*. Avaliar as notificações entre entidades PON internas a uma mesma *Rule* e os *overheads* entre *Rules* em diferentes núcleos**.

Os custos por notificação (entre entidades PON internas a uma mesma *Rule*) e do *overhead* (entre *Rules* em diferentes núcleos) devem ser estimados (i.e. informados) na primeira vez em que o *software* PON estiver em execução.

Ao final dessa etapa (“4. Balanceamento de Carga de Trabalho”), tais valores (notificação e *overhead*) podem ser atualizados com os devidos cálculos efetivos (e não mais estimados), pois já terá ocorrido a distribuição das partes do *software* PON nos núcleos.

Suponha-se que o custo por notificação (*notification*) informado foi de 1ms e o do *overhead* (OH) de 2ms. Ambos os custos são medidos em intervalos de tempo (em milissegundos).

- **Atividade *ix*. Calcular os custos de processamento por intervalo de tempo de monitoramento (TT) por *Rule* (IC) e entre *Rules* em diferentes núcleos (EC)**.

Para a realização dos cálculos dos custos de processamento por TT por *Rule* (IC), deve-se:

- Multiplicar o índice de acoplamento interno (IC) por *Rule* pelo custo por notificação (entre entidades PON internas a uma mesma *Rule*) para cada *Rule* (das *Rules* exibidas na Tabela 1) (IC x *notification*).
- Dividir o TT pela frequência de uso de cada UC e, assim, obter o número de execuções de cada *Rule* ou *Rule executions number* (*Ren*) (uma vez que *Rules* realizam uma determinada colaboração que, por sua vez, realizam um UC).
- Multiplicar o resultado de (IC x *notification*) por *Ren* obtendo o custo total de processamento por *Rule* ou *total processing cost* (*tpc*).

Desta forma, o resultado do cálculo do custo de processamento por TT por *Rule* (i.e. o custo total de processamento do IC por *Rule*, representado por *tpc*), para o exemplo considerado, é apresentado na Tabela 3.

Tabela 3. Custo de processamento por tempo de monitoramento por *Rule*

Rule	**IC x *notification***	***Ren* = TT / UC**	***tpc* = (IC x *notification*) x *Ren***
R1	14	1	14
R2	13	3	39
R3	15	2	30
R4	10	10	100
		16	183

Pode-se notar na última linha da Tabela 3 o valor 16. Tal valor representa o número de execuções da aplicação PON, ou seja, NOP*en* (acrônimo de NOP *executions number*), e seu cálculo se dá por meio do somatório do número de execuções de cada *Rule* que compõe tal aplicação. Da mesma forma, o valor 183 representa o custo total de processamento da aplicação PON, ou seja, NOP*tpc* (acrônimo de NOP *total processing cost*), e seu cálculo se dá por meio do somatório do custo total de processamento de cada *Rule* que compõe tal aplicação.

Exemplificando, para o cálculo da *R2* exibida na Tabela 3, a obtenção do valor 13 (IC x *notification*) deu-se da seguinte maneira: como o índice de acoplamento interno por *Rule* (IC) para *R2* é igual a 13 e o custo por notificação (*notification*) é igual a 1, então: IC x *notification* = 13 x 1 = 13.

Para a obtenção do valor 3, número de execuções de cada *Rule* (*Ren*), dividiu-se o intervalo de tempo de monitoramento (TT) pela frequência de uso de

cada UC (e, nesse caso, tanto o UC1, como o UC2, são realizados por colaborações que, por sua vez, são realizadas em partes pela *R2*[4]). Desta forma:

- *Ren* = TT / UC;
- TT = 1000;
- UC1 = 1000 e UC2 = 500;
- *Ren* = (TT / UC1) + (TT / UC2) = (1000 / 1000) + (1000 / 500);
- *Ren* = 1 + 2 = 3.

E, finalmente, para obter o valor 39, custo total de processamento por *Rule* (*tpc*), multiplica-se os valores obtidos anteriormente, conforme segue: *tpc* = 13 x 3 = 39. Assim, o custo total de processamento por intervalo de tempo de monitoramento para a *R2* (acoplamento interno) é igual a 39.

Para a realização dos cálculos dos custos de processamento entre *Rules* em diferentes núcleos (EC), deve-se:

- Multiplicar o índice de acoplamento externo (EC) entre *Rules* em diferentes núcleos pelo custo do *overhead* (OH) (entre núcleos) para cada relação entre *Rules* (das relações exibidas na Tabela 2) e, assim, obter o custo do acoplamento externo ou EC *cost* (EC*c*), conforme segue: EC*c* = EC x OH.
- Multiplicar o número de execuções do *software* PON (NOP*en*) pelo custo do acoplamento externo (EC*c*) (NOP*en* x EC*c*).
- Dividir o resultado de (NOP*en* x EC*c*) por dois, conforme estratégia adota pelo método.

Deste modo, o resultado do cálculo do custo de processamento entre *Rules* em diferentes núcleos, ou seja, o custo total de processamento do EC (representado por *tpc*EC) é apresentado na Tabela 4.

Tabela 4. Custo de processamento entre *Rules* em diferentes núcleos

Rules	**EC*c***	**NOP*en* x EC*c***	***tpc*EC = (NOP*en* x EC*c*) / 2**
***R1*x(*R2*, *R3*, *R4*)**	10	160	80
***R2*x(*R1*, *R3*, *R4*)**	14	224	112
***R3*x(*R1*, *R2*, *R4*)**	10	160	80
***R4*x(*R1*, *R2*, *R3*)**	10	160	80
(*R1*, *R2*)x(*R3*, *R4*)	4	64	32

[4] Realizadas em partes pela *R2*, pois o UC1 é também realizado pela *R1* e o UC2 pela *R3*.

(*R1*, *R3*)x(*R2*, *R4*)	24	384	192
(*R1*, *R4*)x(*R2*, *R3*)	16	256	128

Por exemplo, para o cálculo da relação (*R1*, *R2*)x(*R3*, *R4*) exibida na Tabela 4, a obtenção do valor 4 (EC*c*) deu-se da seguinte maneira: como o índice de acoplamento externo (EC) entre *Rules* em diferentes núcleos é igual a 2 e o custo do *overhead* (OH) (entre núcleos) para cada relação entre *Rules* é igual a 2, então: EC x OH = 2 x 2 = 4.

Para a obtenção do valor 64, multiplicou-se o número de execuções do *software* PON (NOP*en*) pelo custo do acoplamento externo (EC*c*) e, desta forma: NOP*en* x EC*c* = 16 x 4 = 64.

Por fim, para obter o valor 32, custo total de processamento do acoplamento externo (*tpc*EC), divide-se o resultado de (NOP*en* x EC*c*) por dois (estratégia adota pelo método), conforme segue: *tpc*EC = 64 / 2 = 32. Assim, o custo total de processamento por intervalo de tempo de monitoramento para a relação (*R1*, *R2*)x(*R3*, *R4*) (acoplamento externo) é igual a 32.

Pode-se notar que na Tabela 4 não consta a relação *R1*, *R2*, *R3*, *R4*. Essa relação não é considerada no cálculo do custo de processamento entre *Rules* em diferentes núcleos, pois todas as *Rules* apresentam-se no mesmo núcleo (uma vez que, conforme Tabela 3, para a relação supramencionada, EC = 0).

Entretanto, para a escolha da melhor distribuição das *Rules* nos *clusters* (realização da atividade *xi*) a mesma será considerada, mas com algumas restrições e, no momento oportuno, será explicada.

- **Atividade *x*. Distribuir as partes do *software* PON dentre as possibilidades de *clusters*, por meio das relações utilizadas para o cálculo do EC e também por meio do cálculo do IC de cada *Rule***.

Para distribuir as *Rules* dentre as possibilidades de *clusters* por meio das relações utilizadas para o cálculo do EC, com base nas Tabelas 3 e 4, deve-se:

- Separar em dois *clusters* (estratégia adota pelo método na atividade *vi*) cada relação entre *Rules* (das relações exibidas na Tabela 4).
- Ponderá-las por meio dos custos de processamento total do IC (*tpc*, calculado na Tabela 3) e do EC (*tpc*EC, calculado na Tabela 4).

Assim sendo, o resultado da distribuição das *Rules* dentre as possibilidades de *clusters* é apresentado na Tabela 5.

Tabela 5. Distribuição das *Rules* nos *clusters*

Rules	***Cluster* 1**	***Cluster* 2**
***R1*x(*R2*, *R3*, *R4*)**	94[5]	249[6]
***R2*x(*R1*, *R3*, *R4*)**	151	256
***R3*x(*R1*, *R2*, *R4*)**	110	233
***R4*x(*R1*, *R2*, *R3*)**	180	163
(*R1*, *R2*)x(*R3*, *R4*)	85	162
(*R1*, *R3*)x(*R2*, *R4*)	236	331
(*R1*, *R4*)x(*R2*, *R3*)	242	197

A fim de elucidar os resultados dos cálculos exibidos na Tabela 5, esses se dão por meio da soma do *tpc* (de todos ICs de todas as *Rules* que compõem cada *cluster*) e do *tpcEC* (de todos ECs entre os *clusters*).

O cálculo da relação *R1*x(*R2*, *R3*, *R4*) pode ser exemplificado da seguinte maneira: *Cluster* 1 (*R1*) = 14 (*tpc*) + 80 (*tpc*EC) = 94 e *Cluster* 2 (*R2*, *R3*, *R4*) = 39 + 30 + 100 (*tpc*) + 80 (*tpc*EC) = 249.

Outro exemplo é a relação (*R1*, *R2*)x(*R3*, *R4*), conforme segue: *Cluster* 1 (*R1*, *R2*) = 14 + 39 (*tpc*) + 32 (*tpc*EC) = 85 e *Cluster* 2 (*R3*, *R4*) = 30 + 100 (*tpc*) + 32 (*tpc*EC) = 162.

- **Atividade *xi*. Escolher a melhor relação custo e benefício (i.e. melhor distribuição) das *Rules* nos *clusters* dentre as opções apresentadas na atividade *x***.

Por fim, para escolher a melhor distribuição de *Rules* nos *clusters* (dentre as opções definidas na Tabela 5), deve-se calcular:

- A razão de disponibilidade entre os núcleos.

- A razão de custo de processamento entre os *clusters*.

[5] Custo de processamento de *R1*.
[6] Custo de processamento de (*R2*, *R3*, *R4*).

- O módulo (ou valor absoluto[7]) da subtração entre as razões de disponibilidade entre os núcleos e de custo de processamento entre os *clusters*. Com isso, serão obtidos os valores finais de estratégia.

Tanto a razão de custo de processamento como a razão de disponibilidade está relacionada à razão de proporcionalidade que, na matemática, é a mais simples e comum relação entre grandezas. A proporcionalidade direta é um conceito matemático amplamente difundido, pois é útil e de fácil resolução por meio da "regra de três". Quando existe proporcionalidade direta, a razão (i.e. divisão) entre os valores correspondentes (das duas grandezas relacionadas) é uma constante e a essa se dá o nome de constante de proporcionalidade [53]. Tal constante de proporcionalidade é obtida por meio da divisão do maior valor pelo menor.

Desta forma, por meio da atividade *vii*, a última e atual análise das taxas de disponibilidade dos núcleos do processador (que foi realizada) é a seguinte: AR1 = 67% e AR2 = 33%. Assim, a razão de disponibilidade entre os núcleos ou *Availability RATIO* (ARATIO) é calculada da seguinte maneira: ARATIO = AR1 / AR2 = 0,67 / 0,33 = 2,030303.

Para o cálculo da razão de custo de processamento entre os *clusters* ou *processing cost RATIO* (*pc*RATIO) divide-se o maior valor pelo menor entre eles (i.e. *clusters* 1 e 2) e, então, obtêm-se o *pc*RATIO. Deste modo, por exemplo, pode-se verificar que para a relação *R1*x(*R2*, *R3*, *R4*), tem-se: *pc*RATIO = *Cluster* 2 / *Cluster* 1 = 249 / 94 = 2,648936.

Os valores finais de estratégia ou *Strategy Final Value* (SFV) são computados subtraindo-se ARATIO de *pc*RATIO. Dessa maneira, exemplificando-se para a mesma relação anteriormente mencionada, i.e. *R1*x(*R2*, *R3*, *R4*), tem-se: SFV = |ARATIO – *pc*RATIO| = |2,030303 – 2,648936| = 0,618633.

Assim sendo, as razões de custo de processamento entre os *clusters* e os valores finais de estratégia são apresentados na Tabela 6.

Tabela 6. Valores finais de estratégia

Rules	***pc*RATIO**	**SFV**
***R1*x(*R2*, *R3*, *R4*)**	2,648936	0,618633

[7] O módulo ou valor absoluto de um número é o próprio número sem sinal (i.e. o número positivo).

***R2*x(*R1*, *R3*, *R4*)**	1,695364	0,334939
***R3*x(*R1*, *R2*, *R4*)**	2,118182	0,087879
***R4*x(*R1*, *R2*, *R3*)**	1,104294	0,926009
(*R1*, *R2*)x(*R3*, *R4*)	1,905882	0,124421
(*R1*, *R3*)x(*R2*, *R4*)	1,402542	0,627761
(*R1*, *R4*)x(*R2*, *R3*)	1,228426	0,801877
R1*, *R2*, *R3*, *R4	-	1,5

Analisando os dados exibidos na Tabela 6, pode-se verificar que quanto mais próximo ARATIO de *pc*RATIO, menor o valor de SFV, o que significa que tal valor apresenta a melhor estratégia de distribuição. Para o exemplo considerado, o menor valor de SFV é 0,087879. Ou seja, a relação *R3*x(*R1*, *R2*, *R4*) é a mais adequada às atuais taxas de disponibilidade dos núcleos do processador, AR1 = 67% e AR2 = 33%, pois há processamento disponível (o melhor dentre as opções disponíveis) no núcleo 1 para as *Rules R1*, *R2*, *R4* e, no núcleo 2 para a *Rule R3*. Além disso, tal estratégia remete-se a distribuição *Cluster* 1 (*R3*) = 110 e *Cluster* 2 (*R1*, *R2*, *R4*) = 233. Com isso, na próxima etapa do método, os *clusters* 1 e 2 serão realocados nos núcleos do processador conforme tais cálculos. Isso se dá, pois como SFV é resultado da subtração de ARATIO e *pc*RATIO, quanto mais SFV é próximo a 0, quer dizer que *pc*RATIO (i.e. a razão de custo de processamento entre os *clusters*) é equivalente a ARATIO (i.e. razão de disponibilidade entre os núcleos) e, portanto, quanto mais esse valor aproximar-se do valor 0 (zero), mais otimizado será o balanceamento de carga de trabalho.

Cabe ressaltar que há um caso especial no qual todo o *software* PON (i.e. *R1*, *R2*, *R3*, *R4*), independentemente de distribuições já terem ocorrido, pode permanecer em um mesmo núcleo. Esse caso independerá do valor de *pc*RATIO, devendo então ser estimado o SFV.

Por exemplo, suponha-se agora que a taxa de disponibilidade do núcleo 1 seja igual a 85% e do núcleo 2 igual a 15%. Mesmo o núcleo 2 tendo 15% de processamento disponível, pode não ser interessante alocar parte do *software* PON nele. O método alocaria a *R1* no núcleo 2 e o restante do *software* PON (*R2*, *R3*, *R4*) no núcleo 1. Para isso, há a possibilidade de informar o SFV e, assim, assegurar que o *software* PON não seja distribuído. Para esse exemplo esse valor poderia ser

igual a 2, pois o menor SFV calculado pelo método foi igual a 3,017731 (na qual *R1* seria alocado no núcleo 2 e *R2, R3, R4* no núcleo 1, conforme dito anteriormente).

Portanto, o SFV deve ser configurado com um valor alto (e.g. 10) se o intuito for distribuir as *Rules* em diferentes núcleos de processamento e, caso contrário (i.e. objetive-se não quebrar o *software* PON nos núcleos), iguala-se seu valor a 0 (zero).

Só haverá estratégia se for possível, ou seja, se houver um *cluster* que puder ser redistribuído. Senão, será verificado se o *cluster* que mais processa poderia ser ele movido para outro núcleo, pois existem outros processos usando-o também e não se pode interferir neles segundo a premissa de processo PON único e de não intervenção nos processos não PON (i.e. “premissa de não intervenção nos processos não PON”).

3.2.5. Realocação da aplicação PON

Essa etapa é responsável pela realocação da aplicação e, para isso, baseia-se na etapa anterior, “4. Balanceamento de Carga de Trabalho”, mais especificamente na estratégia de distribuição adotada.

Utilizando-se o mesmo exemplo da etapa 4, como a taxa de disponibilidade do Núcleo 1 (AR1) = 67% e a do Núcleo 2 (AR2) = 33%, deve-se alocar o *Cluster* 1 no Núcleo 2 e, consequentemente, o *Cluster* 2 no Núcleo 1, conforme ilustra a Figura 18.

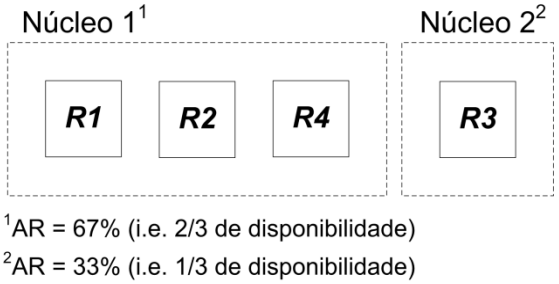

Figura 18. Distribuição dos *clusters* nos núcleos do processador

Essa etapa envolve a seguinte atividade.

- **Atividade *xii*. Reposicionar as partes do *software* PON (i.e. *Rules*) nos núcleos redefinidos**.

Para a realização da atividade *xii*, modificações na estrutura interna do *Framework* PON se farão necessárias, de modo que ao serem notificadas, as entidades PON não executem imediatamente seu código de controle. Ao invés disso, as entidades se registram em controladores de núcleos, os quais armazenam referências para essas, de modo a gerenciar a execução paralela do sistema, definindo em quais núcleos do processador tais entidades executarão.

Basicamente, os controladores de núcleos representam uma espécie de escalonador de entidades. Ainda, serão instanciados um controlador para cada núcleo presente no processador do respectivo ambiente multiprocessado. Sendo assim, cada entidade ao receber uma notificação se registra no respectivo controlador de núcleo, baseado na estratégia definida pelo método em questão.

Por fim, os controladores de núcleos executam as entidades no momento em que estiverem livres, mantendo a execução do sistema de maneira paralela, sempre que existirem entidades em ambos os núcleos.

Finalmente, volta-se à etapa “2. Monitoramento da Carga de Trabalho”, na qual o módulo de *software* continua o monitoramento dinâmico das cargas de trabalho do processador.

3.3. CONCLUSÕES DO CAPÍTULO

Esse capítulo propôs o método para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*, que visa contribuir para o melhor aproveitamento da capacidade de processamento do *hardware* disponível.

A prática de programação baseada na implementação de códigos eficientes e com facilidades de distribuição não deve exigir grandes esforços dos programadores, diferentemente do que ocorre na construção de *software* multiprocessado usando os conceitos relativos aos atuais paradigmas.

No PON, com a possibilidade de particionar os elementos em entidades, como em termos de regras, o programador pode se beneficiar das reais vantagens do uso da computação paralela e distribuída, devido ao maior desacoplamento entre as partes das regras. Ao programar no estilo orientado a notificações, o

desenvolvedor cria programas minimamente acoplados e isso corrobora e viabiliza a aplicação do método proposto.

Em relação às decisões relacionadas ao balanceamento de carga, essas podem ser adotadas de forma centralizada, distribuída ou por uma combinação de ambas. Neste trabalho, em um primeiro momento, se utilizará a decisão por meio do balanceamento de carga centralizado. Isso, devido ao fato do também proposto MCT ser um módulo de *software* e, portanto, ficar alocado em apenas um núcleo do processador.

Com relação aos objetivos propostos inicialmente, o método incorpora técnicas que são apresentadas na Tabela 7. Nessa tabela, para cada etapa proposta (i.e. atividades), apresentam-se respectivamente as técnicas e conclusões baseadas nos estudos realizados.

Tabela 7. Objetivos do método proposto x técnicas e conclusões

Proposta	Conclusão
Levantar os UC que compõem a aplicação e estimar empiricamente suas frequências de uso. Definir quais *Rules* realizam uma determinada colaboração que, por sua vez, realizam um UC. Averiguar a taxa de utilização do processador e alocar o *software* PON.	Um UC representa uma unidade funcional provida ou que será construída em um *software*. Percebeu-se que um UC é realizado por *Rules*. Essa abstração, mesmo ao desenvolvedor da aplicação, facilita a concepção do *software*. Assim, ao invés do levantamento de regras, o desenvolvedor utilizará UC para o levantamento de requisitos, como de costume. Com isso, por ser uma técnica já utilizada e bem sucedida, facilitará e agilizará o trabalho do desenvolvedor.
Analisar a evolução da carga de trabalho dos núcleos. Verificar a proximidade de gargalo segundo algum limite e produzir a detecção de gargalo.	Dinamicamente, o *software* PON estará em monitoramento constante nos processadores e, desta forma, apresentando quaisquer instabilidades (i.e. gargalos), novas análises e agrupamentos se farão necessários. O MCT apresenta-se como um novo monitor que aproveitará efetivamente a capacidade do *hardware* disponível.

Determinar o IC e o EC entre as *Rules*. Examinar a taxa de utilização do processador.	Nesta etapa estabeleceu-se a estratégia adota pelo método, que consiste de sempre que um determinado núcleo apresentar sobrecarga, a parte do *software* PON que nele executa poder ser dividida em duas partes, estabelecendo assim uma relação entre dois novos *clusters* de *Rules*. Entretanto, apesar da possibilidade de distribuição do *software*, isso não é necessariamente viável em alguns casos, ou seja, caso tenha-se disponibilidade de *hardware* para processamento, mantém-se todo o *software* PON em um mesmo *cluster*.
Avaliar o custo por notificação entre entidades PON internas a uma mesma *Rule* e o custo do *overhead* entre *Rules* em diferentes núcleos. Calcular os custos de processamento por TT por *Rule* (IC) e entre *Rules* em diferentes núcleos (EC). Distribuir as *Rules* dentre as possibilidades de *clusters*. Escolher a melhor distribuição das *Rules* nos *clusters*.	O valor final de estratégia, obtido por meio da subtração da razão de disponibilidade entre os núcleos e da razão de custo de processamento entre os *clusters*, utilizando-se do conceito de proporcionalidade direta, apresenta-se como uma solução adequada e satisfatória para a distribuição de *Rules* entre os *clusters*.
Reposicionar as *Rules* nos núcleos redefinidos.	As modificações efetuadas no *Framework* PON viabilizarão uma distribuição e execução eficiente, de maneira que o *software* não precisará ser pausado para que ocorram as devidas distribuições. Na verdade, a distribuição não será especificamente definida na construção do *software*, mas sim a execução pode ser realizada de maneira paralela, uma vez que as entidades PON apresentam como uma de suas principais características a independência de execução.

Portanto, baseado no método proposto, a criação de um ambiente (i.e. uma possível extensão do *Framework* PON) destinado à distribuição da carga de trabalho dos *softwares* PON em *multicore* mostra-se viável. Desta forma, a continuidade desse trabalho de tese é a efetiva criação dessa extensão e sua avaliação por meio de experimentações.

4. EXPERIMENTAÇÃO

Este capítulo apresenta ensaios por meio de experimentos que tiveram por finalidade a busca do conhecimento de técnicas relacionadas com os objetivos desta pesquisa.

Propõe-se para as experimentações o desenvolvimento de um caso de estudo que é um simulador para o controle de operação de um avião. Basicamente, o caso de estudo simula o controle de operação de uma aeronave e também o tráfego de uma ou mais aeronaves. Assim, o caso de estudo, foi realizado em etapas (i.e. estudos).

O primeiro estudo se deu no POO e sua implementação aconteceu utilizando-se a distribuição da aplicação em ambiente *multicore*, por meio de *threads*. Com essa estratégia buscou-se obtenção de maior conhecimento sobre o domínio da aplicação, melhorando sua compreensão e ajudando no processo de extração de requisitos, bem como no entendimento da programação distribuída em *multicore*. Além disso, esse estudo servirá também para comparativos com as futuras implementações no PON.

O segundo estudo se deu no PON e sua implementação aconteceu utilizando-se o *Framework* PON (i.e. versão original, para ambiente monoprocessado). Com essa estratégia buscou-se entendimento da programação no PON. Ambas as implementações, tanto no POO (*multicore*) quanto no PON (monoprocessado) utilizaram-se do SO Linux.

Conforme seção 3.1, o EPOS será o ambiente de ensaio das experimentações desta tese. Assim, no terceiro estudo, ajustou-se a implementação no POO com *threads* para o EPOS (que pode ser emulado no Linux). Tal implementação deu-se parcialmente (i.e. implementou-se apenas o cerne da aplicação). Com essa estratégia buscou-se conhecimento do EPOS na prática, ou seja, instalação, configuração, utilização e programação no ambiente.

4.1. DOMÍNIO DA APLICAÇÃO

De modo geral, este caso de estudo consistiu na implementação de um simulador para o controle de operação de um avião. Resumidamente, para um avião levantar voo, o seu peso precisa ser erguido, incluindo o peso dos passageiros,

cargas, etc. As asas geram a maior parte da elevação e sustentação necessária para mantê-lo no ar.

Para gerar sua elevação e subsequente sustentação, ele deve ser empurrado por meio do ar. O ar resiste ao movimento sob a forma aerodinâmica de arrasto. Os motores da turbina, que estão localizados sob as asas, fornecem o impulso necessário para superar tal resistência gerada pelo ar [54].

As asas possuem uma parte móvel denominada *aileron*, que são partes móveis das bordas de fuga que servem para controlar seu movimento de rolamento sob o eixo longitudinal (i.e. eixo x). Além disso, as asas possuem componentes adicionais, denominados *flaps* e *slats*. Eles possuem basicamente funções para proporcionar força para a asa e estabilizar o voo durante a decolagem e aterrissagem do avião [54].

Para controlar e manobrar o avião, asas menores estão localizadas na sua parte traseira. Sua cauda (*tail*) possui um formato específico para manter o voo estável, por meio da presença de partes fixas, denominadas estabilizadores horizontais e vertical. Os estabilizadores horizontais apresentam partes móveis denominadas elevadores (*elevators*) anexos a eles, movendo o avião para cima ou para baixo (eixo y). O estabilizador vertical, por sua vez, apresenta anexo a si um leme (*rudder*), que é uma parte móvel com a finalidade de controlar a guinada do avião, movendo sua direção para esquerda ou direita (eixo z) [54].

A concepção do simulador para o controle de operação de um avião se deu por meio das partes componentes do avião e de suas respectivas funcionalidades. Para um melhor entendimento da estrutura deste caso de estudo, a Figura 19 ilustra um diagrama de classes UML no qual o cerne do sistema é representado.

De acordo com a Figura 19, o cerne dessa aplicação diz respeito à simulação de voos (classe *Flight*), que são constituídos por aviões (classe *Airplane*), pilotos (classe *Aviator*) e rotas (classe *Route*).

A classe *Route* representa uma rota, cujos principais atributos são a origem e o destino. Ainda, a relação entre piloto e avião é a mais intensa, no sentido de implicar à maioria das chamadas de métodos da aplicação.

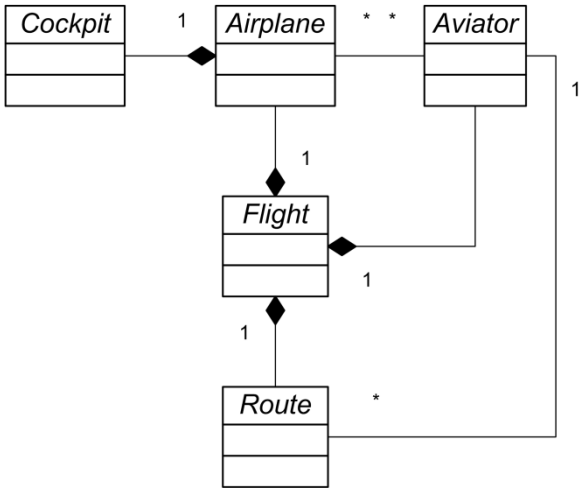

Figura 19. Representação do cerne do controle de operação de um avião

Um avião é composto por um *cockpit* (classe *Cockpit*), que representa uma cabine de pilotagem. Essa, por sua vez, está situada na frente do avião, sendo controlada pelo piloto. A cabine de pilotagem inclui uma série de instrumentos de voo, os quais permitem com que o piloto possa acelerar, levantar voo, guiar e aterrissar o avião durante o percurso preestabelecido em seu plano de voo.

Os instrumentos de voo representados para esse caso de estudo foram: altímetro (classe *Altimeter*), controlador de aceleração do motor (classe *EngineThrottleControl*), controlador de *flaps* (classe *FlapsSwitch*), odômetro (classe *Odometer*), controlador de *slats* (classe *SlatsSwitch*), velocímetro (classe *Speedometer*) e alavanca do estabilizador horizontal (classe *Stick*). Tal estrutura é representada por meio do diagrama de classes UML ilustrado na Figura 20.

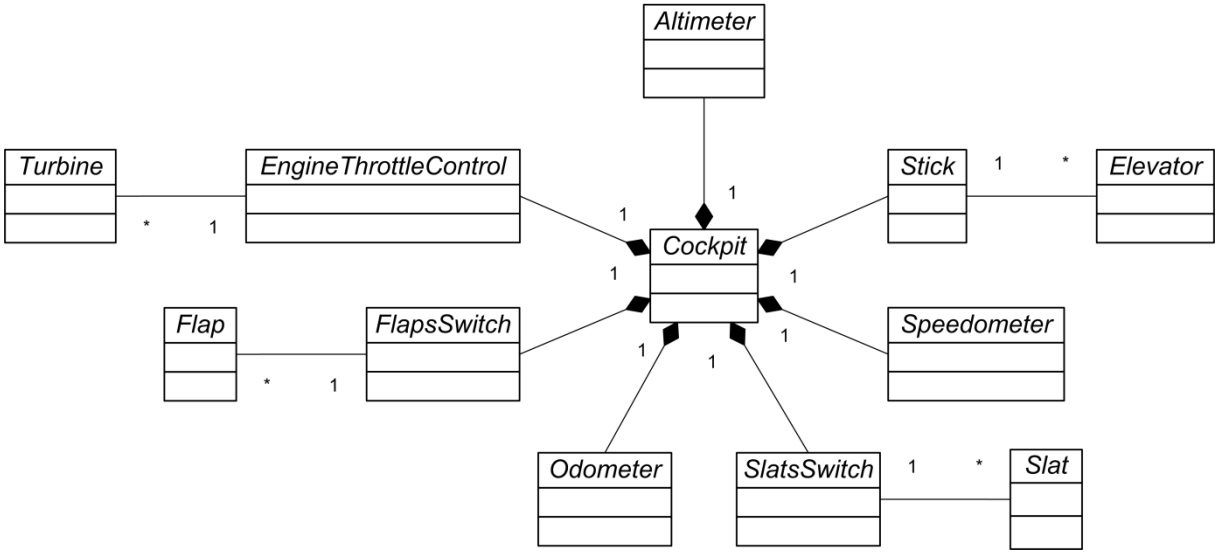

Figura 20. Estrutura do *Cockpit*

A Figura 21 ilustra um diagrama de classes UML que representa o restante do avião (i.e. suas demais partes componentes), formado basicamente pela cauda (classe *Tail*) e asas (classe *Wing*). Tal estrutura é composta pelas seguintes partes: *cockpit*, tanque de combustível (classe *FuelTank*), cauda, trem de pouso (classe *Undercarriage*) e asas.

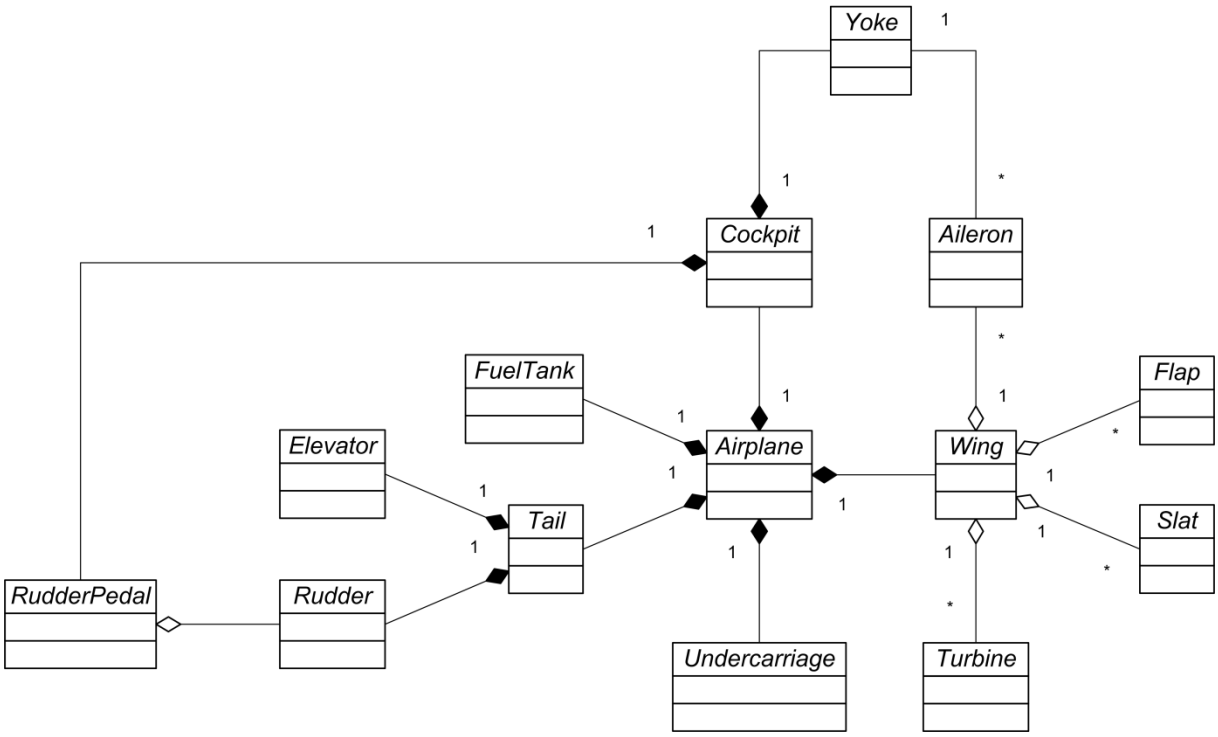

Figura 21. Estrutura da cauda e das asas do avião

A classe *Tail* é composta pelas partes estabilizador horizontal (classe *Elevator*) e leme (classe *Rudder*). Por fim, a classe *Wing* é composta pelos *ailerons* (classe *Aileron*), *flaps*, *slats* e turbinas (classe *Turbine*).

Visando facilitar o entendimento do fluxo de execução principal da aplicação, a Figura 22 ilustra um diagrama de atividades UML que esboça o núcleo de sua execução.

O fluxo de execução principal da aplicação diz respeito ao controle de operação do avião, desde o processo de decolagem, estabilização do voo, até a aterrissagem. Em tais processos, as classes de controle apresentadas estão envolvidas.

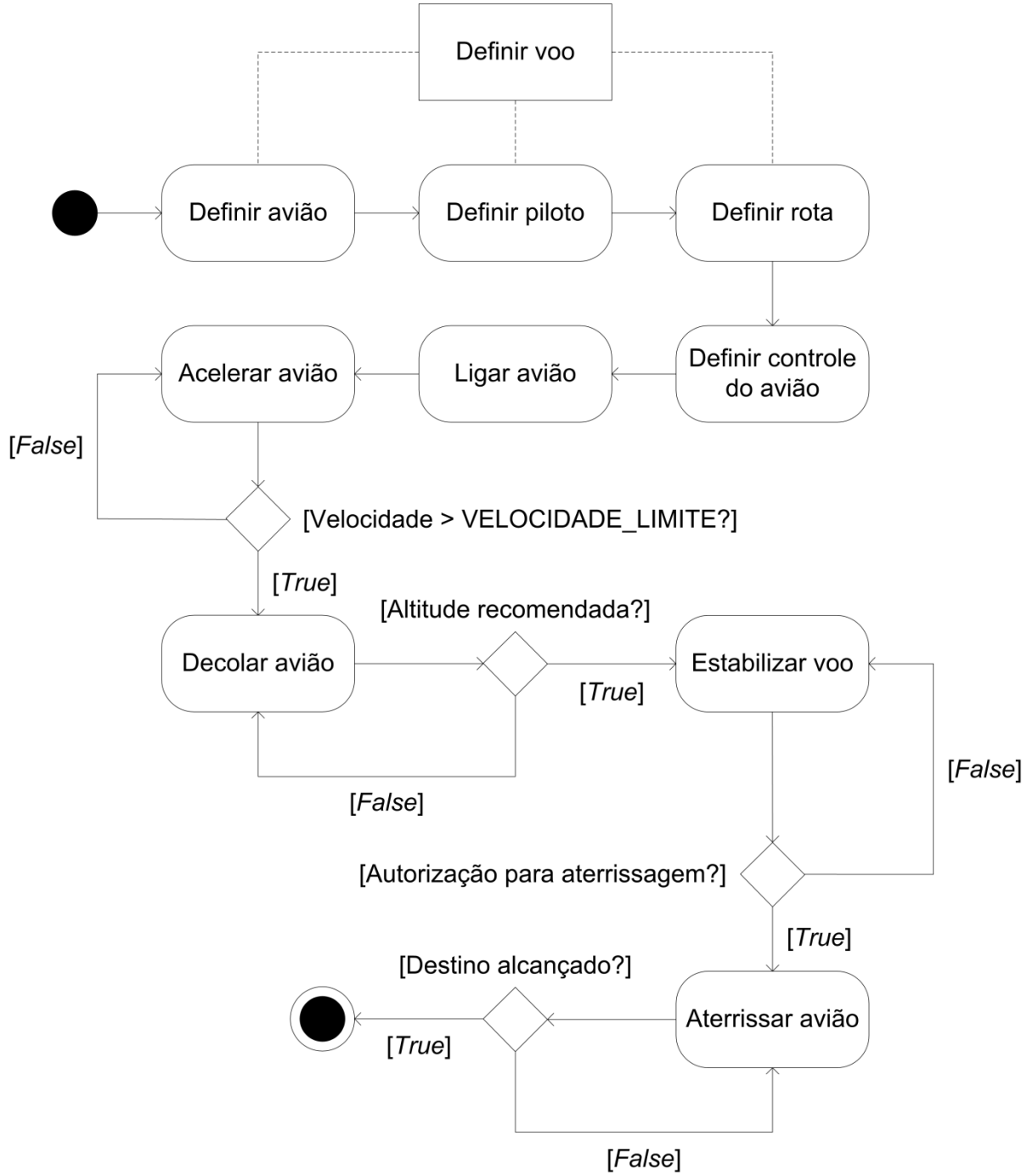

Figura 22. Fluxo de execução principal do controlador de operação de avião

4.2. IMPLEMENTAÇÃO NO POO EM *MULTICORE* COM *THREADS* NO LINUX

No primeiro estudo implementou-se o simulador para o controle de operação de um avião no POO, distribuindo-se a aplicação em ambiente *multicore*, por meio de *threads* no SO Linux.

A compreensão do domínio da aplicação e a extração de requisitos do simulador, feita no primeiro estudo no POO, deram-se naturalmente, devido ao

conhecimento do autor e do seu domínio do paradigma e da linguagem de programação C++.

De modo a exemplificar a implementação da aplicação (i.e. programação distribuída em *multicore*), a concepção do seu cerne (Figura 19) se deu por meio dos seus componentes e de suas funcionalidades, conforme ilustra o Código-fonte 1.

```
1  void Main::initialize() {
2
3   // Create airplanes
4   Airplane* boeing737 = new Boeing737();
5   boeing737->setInitialPosition(0, 0);
6
7   Airplane* boeing747 = new Boeing747();
8   boeing747->setInitialPosition(50, 0);
9
10  // Create aviators
11  Aviator* joao = new Aviator("Joao", new AggressiveBehavior());
12  joao->setCPU(0); // Core 1
13
14  Aviator* jose = new Aviator("Jose", new PassiveBehavior());
15  jose->setCPU(1); // Core 2
16
17  // Create routes
18  Route* curitibaToManaus = new Route(1000000000);
19  Route* curitibaToRio = new Route(2000000000);
20
21  // Create flights
22  Flight* flight1A = new Flight(curitibaToManaus, boeing737,
joao);
23  flights.push_back(flight1A);
24
25  Flight* flight2B = new Flight(curitibaToRio, boeing747, jose);
26  flights.push_back(flight2B);
27
28  ...
29
30 }
```

Código-fonte 1. Método *initialize()* do simulador

O método *initialize()* é responsável pela criação dos aviões (*airplane*), pilotos (*aviator*), das rotas (*route*) e dos voos (*flight*). Ainda, cabe destacar o método *setCPU* (linhas 12 e 15), cujo parâmetro é a CPU que define o núcleo no qual a *thread* será executada. Conforme seção 2.3, a distribuição da aplicação, em ambiente *multicore*, se dá por meio de *threads*.

Em se tratando de *thread*, o método *run()* mostra a execução de uma *thread* quando o piloto inicia o controle do avião, conforme ilustra o Código-fonte 2.

```
1  // Runs the thread
2  void Aviator::run() {
3
4   Thread::run();
5   int currentDistance = 0;
6   int count = 0;
7
8   while (currentDistance <= route->getDistanceInMeters()) {
9    this->pilot();
10   currentDistance = cockpit->getOdometer()-
>getCumulativeDistanceInMeters();
11   count++;
12  }
13
14  ...
15
16 }
```

Código-fonte 2. Método *run()*

O controle (i.e. a pilotagem propriamente dita), realizado pelo piloto, executado por meio do método *pilot()* (linha 9), dispara o comando de partes componentes do *Cockpit* (Figura 20), conforme ilustra o Código-fonte 3.

```
1  void Aviator::pilot() {
2
3   if (cockpit->getLeftEngineThrottleControl()->getPercentage() <
behavior->getMaxAccelerationPercent()) {
4    this->accelerateLeftEngine(behavior->getAccelerationPercent());
5   }
6
7   if (cockpit->getRightEngineThrottleControl()->getPercentage() <
behavior->getMaxAccelerationPercent()) {
8    this->accelerateRightEngine(behavior-
>getAccelerationPercent());
9   }
10
11  if ((this->cockpit->getAltimeter()->getAltitudeInFeets() <
airplane->getRecommendedAltitudeInFeets()) &&
12   (this->cockpit->getSpeedometer()->getSpeed() >= behavior-
>getSpeedToTakeOff())) {
13   this->takeOff(behavior->getTakeOffAngle());
14  }
15
16  if (this->cockpit->getAltimeter()->getAltitudeInFeets() >=
airplane->getRecommendedAltitudeInFeets()) {
17   this->normalFlight();
18  }
19
20  ...
21
```

```
22 }
```

Código-fonte 3. Método *pilot()*

O método *pilot()* tem por objetivo controlar as partes do *cockpit*, exemplificadas por meio de alguns dos componentes ilustrados no Código-fonte 3, tais como os controladores de aceleração dos motores (esquerdo e direito), controlador de altitude e controlador de velocidade.

Por fim, tal implementação servirá também para comparativos com as futuras implementações no PON (tanto em ambiente monoprocessado, quanto em ambiente multiprocessado).

4.3. IMPLEMENTAÇÃO NO PON MONOPROCESSADO NO LINUX

No segundo estudo implementou-se parcialmente o simulador no PON, pois, o intuito nesta etapa foi obter conhecimento da programação no paradigma. Para tal implementação, utilizou-se o *Framework* PON (versão original para ambiente monoprocessado) no SO Linux.

Visando exemplificar a concepção de alguns dos componentes e funcionalidades do simulador (Figura 19), como no POO, o Código-fonte 4 ilustra a representação de parte do método *initialize()* (Código-fonte 1) do simulador que representa o cerne do controle de operação de um avião no PON.

```
1  void Main::initFactBase() {
2
3   // Create routes
4   curitibaToManaus = new Route(1000000000);
5
6   // Create airplanes
7   boeing737 = new Airplane();
8
9   // Create aviators
10   joao = new Aviator("Joao", new AggressiveBehavior());
11
12   // Create flights
13   flight1A = new Flight(curitibaToManaus, boeing737, joao);
14
15  ...
16
17 }
```

Código-fonte 4. Método *initFactBase()* do simulador no PON

Conforme o método *initialize()* (Código-fonte 1), o método *initFactBase()* é responsável pela criação das rotas, dos aviões, pilotos e dos voos, só que agora no PON.

O controle de operação do avião (i.e. a pilotagem, realizada pelo piloto) é executado por meio do método *initRules()*. Da mesma forma que o método *pilot()* (Código-fonte 3) no POO, o método *initRules()* (no PON) dispara o comando de partes componentes do *cockpit* (Figura 20), conforme ilustra o Código-fonte 5.

```
1  void Main::initRules() {
2
3   RuleObject* ruleAviatorCanOperate = elementsFactory-
>createRuleObject(
4   "ruleAviatorCanOperate", scheduler, Condition::SINGLE);
5   ruleAviatorCanOperate->addPremise(elementsFactory-
>createPremise(
6   flight1A->airplane->atCurrentY,
7   flight1A->route->distanceInMeters,
8   Premise::SMALLEROREQUAL,
9   false));
10
11   RuleObject* ruleAviatorAccelerateLeftEngine = elementsFactory-
>createRuleObject(
12   "ruleAviatorAccelerateLeftEngine", scheduler,
Condition::CONJUNCTION);
13   ruleAviatorAccelerateLeftEngine-
>addMasterRule(ruleAviatorCanOperate);
14   ruleAviatorAccelerateLeftEngine->addPremise(elementsFactory-
>createPremise(
15   flight1A->airplane->cockpit->leftEngineThrottleControl-
>atPercentage,
16   flight1A->aviator->behavior->maxAccelerationPercent,
17   Premise::SMALLERTHAN,
18   false));
19   ruleAviatorAccelerateLeftEngine-
>addInstigation(elementsFactory->createInstigation(flight1A-
>aviator->mtAccelerateLeftEngine));
20
21   RuleObject* ruleAviatorAccelerateRightEngine = elementsFactory-
>createRuleObject(
22   "ruleAviatorAccelerateRightEngine", scheduler,
Condition::CONJUNCTION);
23   ruleAviatorAccelerateRightEngine-
>addMasterRule(ruleAviatorCanOperate);
24   ruleAviatorAccelerateRightEngine->addPremise(elementsFactory-
>createPremise(
25   flight1A->airplane->cockpit->rightEngineThrottleControl-
>atPercentage,
26   flight1A->aviator->behavior->maxAccelerationPercent,
27   Premise::SMALLERTHAN,
28   false));
```

```
29   ruleAviatorAccelerateRightEngine-
>addInstigation(elementsFactory->createInstigation(flight1A-
>aviator->mtAccelerateRightEngine));
30
31   RuleObject* ruleAviatorTakeOffAirplane = elementsFactory-
>createRuleObject(
32   "ruleAviatorTakeOffAirplane", scheduler,
Condition::CONJUNCTION);
33   ruleAviatorTakeOffAirplane-
>addMasterRule(ruleAviatorCanOperate);
34   ruleAviatorTakeOffAirplane->addPremise(elementsFactory-
>createPremise(
35   flight1A->airplane->cockpit->altimeter-
>atCurrentAltitudeInFeets,
36   flight1A->airplane->recommendedAltitudeInFeets,
37   Premise::SMALLERTHAN,
38   false));
39   ruleAviatorTakeOffAirplane->addPremise(elementsFactory-
>createPremise(
40   flight1A->airplane->cockpit->speedometer->atCurrentSpeed,
41   flight1A->aviator->behavior->speedToTakeOff,
42   Premise::GREATEROREQUAL,
43   false));
44   ruleAviatorTakeOffAirplane->addInstigation(elementsFactory-
>createInstigation(flight1A->aviator->mtTakeOffAirplane));
45
46   RuleObject*
ruleAirplaneAtRecommendedAltitudeAviatorStabilizeFlight =
elementsFactory->createRuleObject(
47   "ruleAviatorTakeOffAirplane", scheduler,
Condition::CONJUNCTION);
48   ruleAviatorStabilizeFlight-
>addMasterRule(ruleAviatorCanOperate);
49   ruleAviatorStabilizeFlight->addPremise(elementsFactory-
>createPremise(
50   flight1A->airplane->cockpit->altimeter-
>atCurrentAltitudeInFeets,
51   flight1A->airplane->atRecommendedAltitudeInFeets,
52   Premise::GREATEROREQUAL,
53   false));
54   ruleAviatorStabilizeFlight->addInstigation(elementsFactory-
>createInstigation(flight1A->aviator->mtStabilizeFlight));
55
56   ruleAirplaneAtRecommendedAltitudeAviatorStabilizeFlight->end();
57   ruleAviatorTakeOffAirplane->end();
58   ruleAviatorAccelerateRightEngine->end();
59   ruleAviatorAccelerateLeftEngine->end();
60   ruleAviatorCanOperate->end();
61
62  ...
63
64 }
```

Código-fonte 5. Método *initRules()* no PON

O método *initRules()* cria as regras que controlam as partes do *cockpit*, exemplificadas por meio do Código-fonte 5, tais como os controladores de aceleração dos motores esquerdo e direito (*ruleAviatorAccelerateLeftEngine* e *ruleAviatorAccelerateRightEngine*, respectivamente) e controlador de altitude (*ruleAirplaneAtRecommendedAltitudeAviatorStabilizeFlight*).

Desta forma, tal implementação, como a anterior, servirá também para comparativos com as futuras implementações no PON em ambiente multiprocessado.

4.4. IMPLEMENTAÇÃO NO POO COM *THREADS* NO EPOS

No terceiro estudo implementou-se parcialmente o simulador no POO com *threads* no EPOS. O intuito nesta etapa foi obter conhecimento prático do EPOS. Implementou-se (na verdade, ajustou-se a implementação do primeiro estudo no POO) para o EPOS apenas o cerne da aplicação. Utilizou-se novamente o SO Linux, uma vez que o EPOS pode ser nele emulado.

Com o intuito de exemplificar alguma implementação da aplicação no EPOS, foram concebidos dois métodos conforme ilustra o Código-fonte 6. Ainda, a concepção desses métodos (i.e. da aplicação como um todo) precisou ser compilada para o EPOS, depois de desenvolvida[8].

```
1   int createAviatorJoaoAndPrintSummary() {
2    Airplane* Boeing737 = new Airplane(50000, "Boeing 737");
3    Boeing737->setInitialPosition(0, 0);
4
5    Aviator* joao = new Aviator("Joao");
6    joao->setAirplane(Boeing737);
7
8    cout << "O >> " << joao->getName() << " esta pilotando um >> "
9    << joao->getAirplane()->getName() << "\n";
10
11   return 0;
12  }
13
14  int createAviatorJoseAndPrintSummary() {
15   Airplane* Boeing747 = new Airplane(62000, "Boeing 747");
16   Boeing747->setInitialPosition(30, 0);
17
18   Aviator* jose = new Aviator("Jose");
19   jose->setAirplane(Boeing747);
```

[8] A documentação da API do EPOS é mantida no site: https://epos.lisha.ufsc.br/EPOS+User+Guide.

```
20
21    cout << "O >> " << jose->getName() << " esta pilotando um >> "
22    << jose->getAirplane()->getName() << "\n";
23
24    return 0;
25   }
26
27   int main () {
28    Thread * firstAviatorCreation = new
Thread(&createAviatorJoaoAndPrintSummary);
29    Thread * secondAviatorCreation = new
Thread(&createAviatorJoseAndPrintSummary);
30    firstAviatorCreation->join();
31    secondAviatorCreation->join();
32
33    ...
34
35   }
```

Código-fonte 6. Exemplo de código no POO com *threads* no EPOS

O método *createAviatorJoaoAndPrintSummary()* é responsável por criar um avião, um piloto, atribuir esse piloto ao avião criado e, posteriormente, imprimir na tela essa relação, piloto e avião. O método *createAviatorJoseAndPrintSummary()* tem a mesma função, entretanto para outro avião e piloto. E, o método *main()*, atribui cada um dos métodos supracitados a uma *thread* diferente e coloca-os em execução. Enfim, conforme dito no início dessa seção, tal implementação serviu para conhecimento prático do EPOS.

4.5. CONCLUSÕES DO CAPÍTULO

Esse capítulo mostrou ensaios por meio de experimentos, que tiveram por finalidade a busca do conhecimento de técnicas relacionadas com os objetivos desta pesquisa.

Pretende-se, a partir desses experimentos, avaliar a proposta desta tese, por meio da criação do método para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore*, estendendo o *Framework* PON.

Em relação ao primeiro estudo, implementação no POO em *multicore* com *threads* no Linux, obteve-se efetivo conhecimento do domínio da aplicação, o que melhorou sua compreensão e ajudou no processo de extração de requisitos. Além disso, obteve-se entendimento da programação distribuída em *multicore* por meio de *threads* no Linux.

Em relação ao segundo estudo, implementação no PON monoprocessado no Linux, obteve-se entendimento da programação no PON. Entretanto, apesar do estado da arte do PON estar bem fundamentado, em relação ao estado da técnica, seu entendimento e utilização não são triviais.

Em relação ao terceiro estudo, implementação no POO com *threads* no EPOS, emulado no Linux por meio do QEMU, apesar da implementação realizada inicialmente no POO (em *multicore* com *threads*), a adaptação para o EPOS não se deu naturalmente, devido ao fato do EPOS também ter sido concebido por meio de um *framework* (mesmo que em linguagem de programação C++). Tal *framework* não usa, por exemplo, a estrutura *vector*, apresentando uma implementação própria. O mesmo ocorre com o uso de *threads* e outras bibliotecas para distribuição em ambiente *multicore*. Entretanto, isso não inviabilizou seu uso e, desta forma, pôde-se instalar, configurar e programar no ambiente emulado no Linux.

5. CONCLUSÕES

O ato de concluir uma pesquisa (ou parte de uma pesquisa) pode ser visto sob a ótica de um balanço (orçamentário). Fazendo tal analogia a esse trabalho de tese, com vistas à qualificação, pode-se começar essa avaliação respondendo à primeira pergunta (do problema de base), que foi: como desenvolver *software* que possa fazer uso das infraestruturas de alta capacidade de processamento oferecida pelos processadores *multicore*?

Desde o surgimento do PON, algumas de suas características, como por exemplo, o baixo nível de acoplamento entre as partes das entidades, motivou essa pesquisa, propondo um estudo mais aprofundado para sua aplicação no desenvolvimento de *software* em ambiente *multicore*. Contudo, durante tais estudos, verificou-se que, devido ao fato do PON ser implementado como um *framework* específico para ambientes monoprocessados, isso não seria viável. Assim, do problema de base apresentado, derivou-se o seguinte problema decorrente: Não há um modelo nem técnica para tratar explicitamente e usufruir da programação paralela usando o PON.

Com isso, além do objetivo (geral) dessa pesquisa, que foi propor um método para distribuição dinâmica da carga de trabalho dos *softwares* PON em *multicore* (resolução do problema de base), para que o método pudesse ser viabilizado, foi preciso analisar o estado da técnica do PON. Percebeu-se certa carência na implementação do *framework*, visto que ele não aproveita os benefícios do paradigma, como é o caso da possibilidade de processamento paralelo das entidades que compõem um *software* PON. A aplicação do método proposto no *framework* vigente é impraticável. Nesse sentido, serão necessárias mudanças estruturais no *framework* para que se possa fazer uso efetivo dos múltiplos núcleos de processamento e então viabilizar o método proposto.

Diante do exposto, alguns objetivos específicos se fizeram presentes durante as pesquisas supracitadas. A análise do PON, bem como o aprofundamento do conhecimento sobre a tecnologia *multicore*, ambos com vistas à concepção de *software* paralelo, se mostraram importantes descobertas de modo que instigaram à proposição de certas mudanças no *Framework* PON.

Em relação ao método proposto, pode-se verificar que o *software* PON estará sob o supervisionamento periódico de um monitor de desempenho aplicado

aos processadores. Desta forma, quando algum núcleo apresentar gargalo, novas análises e agrupamentos poderão se apresentar como soluções melhores para a distribuição do *software* em questão. A análise e alocação do *software* PON ocorre dinamicamente, de forma “inteligente”, por meio de uma estratégia para balanceamento de carga e essa se apresenta como uma solução adequada (e satisfatória) para a distribuição das regras entre os *clusters* [50].

Pretende-se, posteriormente a defesa da qualificação, conceber, por meio da implementação de uma possível extensão do *Framework* PON, um conjunto de técnicas para poder avaliar o método proposto (por meio de experimentos), distribuindo a carga de trabalho dos *softwares* PON em *multicore*.

Pode-se notar que os objetivos foram parcialmente alcançados, uma vez que o método foi definido e, a princípio, mostra-se viável. Entretanto, para fazer a avaliação do método, necessita-se da implementação do *Framework* PON para a efetiva execução de *softwares* PON em ambientes paralelos.

Portanto, baseado no método proposto, a criação de um ambiente destinado à distribuição da carga de trabalho dos *softwares* PON em *multicore* mostra-se viável. Desta forma, a continuidade desse trabalho de tese é a efetiva criação dessa extensão e sua avaliação por meio de experimentações.

REFERÊNCIAS

1. PILLA, M.; SANTOS, R.; CAVALHEIRO, G. **Introdução à programação em arquiteturas multicore**. In: Anais da IX Escola Regional de Alto Desempenho. Página(s): 73-102. Porto Alegre, 2009.

2. RUSSINOVICH, M.; SOLOMON, D. **Windows Internals:** Covering Windows Server 2008 and Windows Vista. New York: Microsoft Press, 2008.

3. LOVE, R. **Linux Kernel Development**. Indianapolis: Sams, 2004.

4. BANASZEWSKI, R. **Paradigma Orientado a Notificações:** Avanços e Comparações. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2009.

5. SIMÃO, J.; STADZISZ, P. **Inference Based on Notifications:** A Holonic Metamodel Applied to Control Issues. IEEE Transactions on Systems, Man and Cybernetics, 39 (1): 238-250, 2009.

6. SIMÃO, J. et al. **Notification Oriented Paradigm (NOP) and Imperative Paradigm:** A Comparative Study. Journal of Software Engineering and Applications, 5: 402-416, 2012.

7. HUGHES, C. **Parallel and Distributed Programming Using C++**. Boston: Addison-Wesley, 2003.

8. SIMÃO, J. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2001.

9. SIMÃO, J. **A Contribution to the Development of a HMS (Holonic Manufacturing System) Simulation Tool and Proposition of a Meta-Model for Holonic Control**. Tese (Doutorado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2005.

10. FAISON, T. **Event-Based Programming:** Taking Events to the Limit. New York: Apress, 2006.

11. SIMÃO, J.; STADZISZ, P. **Paradigma Orientado a Notificações (PON):** Uma Técnica de Composição e Execução de Software Orientado a Notificações. Patente submetida ao INPI (Instituto Nacional de Propriedade Industrial). Nº Provisório: 015080004262: Brasil, 2008.

12. SIMÃO, J. et al. **Notification Oriented and Object Oriented Paradigms comparison via Sale System**. Journal of Software Engineering and Applications, 5: 695-710, 2012.

13. SIMÃO, J. et al. **A Game Comparative Study:** Object-Oriented Paradigm and Notification-Oriented Paradigm. Journal of Software Engineering and Applications, 5: 722-736, 2012.

14. SIMÃO, J. et al. **Comparações entre duas materializações do Paradigma Orientado a Notificações (PON):** Framework PON Prototipal versus Framework PON Primário. In: IV Congreso Internacional de Computación y Telecomunicaciones: Lima, 2012.

15. RONSZCKA, A. **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões**. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2012.

16. RONSZCKA, A. et al. **Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo**. In: III Congreso Internacional de Computación y Telecomunicaciones: Lima, 2011.

17. VALENÇA, G. **Contribuição para a Materialização do Paradigma Orientado a Notificações(PON)**. Dissertação (Mestrado em Computação Aplicada): Universidade Tecnológica Federal do Paraná, 2012.

18. WEBER, L. et al. **Viabilidade de Controle Orientado a Notificações (CON) em ambiente concorrente baseado em threads**. In: XV Seminário de Iniciação Científica e Tecnológica: Cornélio Procópio, 2010.

19. SIMÃO, J. et al. **Holonic Manufacturing Execution Systems for Customised and Agile Production:** Manufacturing Plant Simulation. In: V Congresso Brasileiro de Engenharia de Fabricação: Belo Horizonte, 2008.

20. SIMÃO, J.; STADZISZ, P. **An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems**. In: Advances in Logic, Artificial Intelligence and Robotics. Page(s): 234-241. Amsterdam, 2002.

21. SIMÃO, J.; STADZISZ, P.; MOREL, G. **Manufacturing Execution System for Customized Production**. Journal of Material Processing Technology: Amsterdam, 2006.

22. ANDREWS, G. **Foundations of Multithreaded, Parallel, and Distributed Programming**. Boston: Addison-Wesley, 2000.

23. TANENBAUM, A.; WOODHULL, A. **Operating Systems Design and Implementation**. New Jersey: Prentice Hall, 2006.

24. MAZIERO, C. **Sistemas Operacionais**. Disponível em: http://dainf.ct.utfpr.edu.br/~maziero/doku.php/so:livro_de_sistemas_operacionais, acesso em 10-nov-2012.

25. WANG, S.; WANG, L. **Thread-Associative Memory for Multicore and Multithreaded Computing**. In: Proceedings of the International Symposium on Low Power Electronics and Design. Page(s): 139-142. Tegernsee, 2006.

26. TULLSEN, D.; EGGERS, S.; LEVY, H. **Simultaneous multithreading:** Maximizing on-chip parallelism. In: 22th International Symposium on Computer Architecture. Page(s): 392–403, 1995.

27. HUGHES, C.; HUGHES, T. **Professional Multicore Programming:** Design and Implementation for C++ Developers. Indianapolis: Wiley, 2008.

28. FLYNN, M. **Very High-speed Computing Systems**. In: Proceedings of the IEEE, 54 (12): 1901–1909, 1966.

29. FLYNN, M. **Some Computer Organizations and Their Effectiveness**. IEEE Transactions on Computers, 21 (9): 948-960, 1972.

30. HENNESSY, J.; PATTERSON, D. **Computer Architecture:** A Quantitative Approach. San Francisco: Morgan Kaufmann, 2006.

31. DE ROSE, C.; NAVAUX, P. **Arquiteturas Paralelas**. Porto Alegre: Sagra-Luzzatto, 2003.

32. SANTOS, T.; SANTOS, R. **Projeto e Implementação de Arquiteturas Superescalares**. In: V Escola Regional de Alto Desempenho. Página(s): 57-82. Porto Alegre, 2005.

33. BORKAR, S. **Microarchitecture and Design Challenges for Gigascale Integration**. In: 37th IEEE/ACM International Symposium on Microarchitecture. Page(s): 3. Washington, 2004.

34. KONGETIRA, P.; AINGARAN, K.; OLUKOTUN, K. **Niagara:** A 32-way multithreaded sparc processor. IEEE Micro, 25 (2): 21–29, 2005.

35. PATT, Y. et al. **One Billion Transistors, One Uniprocessor, One Chip**. IEEE Computer, 30 (9): 51–57, 1997.

36. MARCUELLO, P.; GONZALEZ, A.; TUBELLA, J. **Speculative Multithreaded Processors**. In: 12th International Conference on Supercomputing. Page(s): 77–84. Melbourne, 1998.

37. KRISHNAN, V.; TORRELLAS, J. **A chip-multiprocessor architecture with speculative multithreading**. IEEE Transactions on Computers, 48 (9): 866–880, 1999.

38. DONALSON, D. et al. **DISC:** Dynamic Instruction Stream Computer an evaluation of performance. In: 24th ACM International Symposium on Microarchitecture. Page(s): 163–171. New York, 1991.

39. POSIX. **IEEE Std 1003.1. IEEE Standards Association:** Austin Joint Working Group. Disponível em: http://standards.ieee.org/develop/wg/POSIX.html, acesso em 31-mar-2012.

40. BUTENHOF, D. **Programming with POSIX Threads**. Boston: Addison-Wesley, 1997.

41. FRIGO, M. **Multithreaded Programming in Cilk**. In: Proceedings of the International Workshop on Parallel Symbolic Computation. Page(s): 13–14. New York, 2007.

42. PHEATT, C. **Intel Threading Building Blocks**. Journal of Computing Sciences in Colleges, 23 (4): 298–298, 2008.

43. CHANDRA, R. **Parallel Programming in OpenMP**. San Francisco: Morgan Kaufmann Publishers, 2001.

44. NICKOLLS, J. et al. **Scalable Parallel Programming with CUDA**. Queue, 6 (2): 40–53, 2008.

45. FROHLICH, A. **Application-Oriented Operating Systems**. GMD Forschungszentrum Informationstechnik: Sankt Augustin, 2001.

46. WANNER, L.; FROHLICH, A. **Operating System Support for Wireless Sensor Networks**. Journal of Computer Science, 4 (4): 272-281, 2008.

47. FROHLICH, A.; SCHRODER-PREIKSCHAT, W. **Scenario Adapters:** Efficiently Adapting Components. In: Proceedings of the 4th World Multi-Conference on Systemics, Cybernetics and Informatics: Orlando, 2000.

48. MARCONDES, H. et al. **Operating Systems Portability:** 8 bits and beyond. In: 11th IEEE Conference on Emerging Technologies and Factory Automation. Page(s): 124-130. Cracow, 2006.

49. POLPETA, F.; FROHLICH, A. **Hardware mediators:** a portability artifact for component-based systems. In: Proceedings of the 9th International Conference on Embedded and Ubiquitous Computing. Page(s): 271–280. Aizuwakamatsu, 2004.

50. BELMONTE, D.; SIMÃO, J.; STADZISZ, P. **Proposta de um Método para Distribuição da Carga de Trabalho usando o Paradigma Orientado a Notificações (PON)**. Journal SODEBRAS, 8 (84): 10-17, 2012.

51. EPOS. **Embedded Parallel Operating System**. Disponível em: http://epos.lisha.ufsc.br/tiki-index.php, acesso em 25-jul-2012.

52. PRESSMAN, R. **Software Engineering:** A Practitioner's Approach. New York: McGraw-Hill, 2010.

53. DOWNING, D. **Dictionary of Mathematics Terms**. Hauppauge: Barrons Educational Series, 2009.

54. NASA. **Beginner's Guide to Aerodynamics:** Parts of an Airplane. Disponível em: http://www.grc.nasa.gov/WWW/k-12/airplane/airplane.html, acesso em 23-jul-2012.

55. KOSCIANSKI, A. et al. **FMS Design and Analysis:** Developing a Simulation Environment. In: XV International Conference on CAD/CAM: Robotics and Factories of the Future, 1999.