Anna Victoria Cabrera Rondon Oikawa

Balanceamento de Carga Dinâmico Orientado a Aprendizado de Máquina para Aplicações Paralelas

Florianópolis

2019

Anna Victoria Cabrera Rondon Oikawa

Balanceamento de Carga Dinâmico Orientado a Aprendizado de Máquina para Aplicações Paralelas

Monografia submetida ao Programa de Graduação em Ciências da Computação para a obtenção do Grau de Bacharel.

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Ciências da Computação

Orientador: Márcio Bastos Castro

Coorientador: Laércio Lima Pilla

Florianópolis

2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Oikawa, Anna Victoria Cabrera Rondon
Balanceamento de Carga Dinâmico Orientado a Aprendizado de Máquina para Aplicações Paralelas / Anna Victoria Cabrera Rondon Oikawa ; orientador, Márcio Bastos Castro, coorientador, Laércio Lima Pilla, 2019.
133 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciências da Computação. 2. Ciências da Computação. 3. Balanceamento de Carga. 4. Aprendizagem de Máquina. 5. Computação de Alto Desempenho. I. Bastos Castro, Márcio. II. Lima Pilla, Laércio. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Balanceamento de Carga Dinâmico Orientado a Aprendizado de Máquina para Aplicações Paralelas

Anna Victoria Cabrera Rondon Oikawa

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Ciências da Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Márcio Bastos Castro
Universidade Federal de Santa Catarina - UFSC

Orientador: Prof. Laércio Lima Pilla
Universidade Federal de Santa Catarina - UFSC

Prof. Frank Augusto Siqueira
Universidade Federal de Santa Catarina - UFSC

Prof. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina - UFSC

Florianópolis, 14 de julho de 2019.

Dedico este trabalho primeiramente a Deus, que é o meu sustento, autor da minha vida e a razão do meu viver. A Ele toda a honra, toda glória e todo o louvor. Dedico também aos meus pais, que ajudaram a formar o meu caráter e a base que eu precisava para enfrentar os desafios da vida.

Agradecimentos

Aos meus pais, por todo amor incondicional, pela força nos momentos difíceis e pelo suporte dado para que eu tivesse sucesso. Vocês são essenciais e a base de tudo o que sou.

Aos meus irmãos, que me ensinaram a importância do esforço e dos estudos. Mais do que irmãos, foram pais para mim durante minha infância.

Toda minha gratidão e amor aos meus tios Josefina e Everaldo, que foram um dos principais pontos de apoio durante minha trajetória no curso. Vocês foram fundamentais para tornar esse sonho possível.

Aos meus amigos, que trouxeram alegria em situaçõse árduas e que contribuíram na minha evolução para eu me tornar uma mulher mais forte.

A todas as pessoas que de alguma forma fizeram parte dessa história. Deixo aqui um agradecimento eterno, pois vocês transmitiram a força e a confiança que eu necessitava quando me mudei para uma nova cidade.

Aos meus professores Márcio e Laércio, responsáveis pela orientação deste trabalho e pelo empenho nas revisões. Agradeço também a todos os professores que contribuíram com a minha formação acadêmica.

À Universidade Federal de Santa Catarina, por me proporcionar um ambiente agradável, motivador e com múltiplas oportunidades.

E agradeço a Deus, que é a maior certeza de minha vida, que me fortalece constantemente e que tem me abençoado desde o meu primeiro fôlego de vida.

“Consagre ao Senhor tudo o que você faz,
e os seus planos serão bem-sucedidos.“
(Bíblia Sagrada, Provérbios 16:3)

Resumo

Aplicações científicas desenvolvidas em grandes centros de pesquisa necessitam de alto poder computacional para que possam obter resultados precisos. Para que seja possível explorar ao máximo todos os recursos computacionais e de memória disponíveis, faz-se o uso de plataformas de Computação de Alto Desempenho (*High Performance Computing* - HPC) em conjunto com interfaces de Computação Paralela. Uma aplicação HPC pode ser decomposta em tarefas paralelas, onde cada uma possui cargas computacionais diferentes de acordo com suas características individuais. Tais diferenças resultam em um desbalanceamento de carga entre os recursos computacionais, causando um impacto negativo no desempenho e escalabilidade da aplicação. Assim, para evitar o desperdício computacional e de energia, são aplicados algoritmos de escalonamento global para que essas tarefas em desequilíbrio sejam redistribuídas e o desempenho da aplicação tenha uma melhoria significativa.

A proposta deste trabalho é a automatização da escolha de algoritmos de escalonamento global para aplicações científicas HPC, fazendo o uso de Aprendizagem de Máquina para auxiliar na tomada de decisão do melhor algoritmo a ser escolhido em tempo de execução. Essa decisão é feita por um *meta-escalonador adaptativo* que foi desenvolvido com a finalidade de analisar dinamicamente uma aplicação nos instantes de seu balanceamento de carga, e invocar o algoritmo de *Load Balancing* (LB) mais apropriado para aquele contexto. Além de não existir um algoritmo ideal para encontrar uma distribuição de tarefas ótima que funcione em todas as situações, não há uma forma de identificar qual algoritmo é o mais indicado para uma dada aplicação executando em uma dada plataforma e em um dado instante do tempo. Portanto, é um problema não trivial que afeta aplicações científicas atualmente e, assim, o foco deste trabalho é o estudo e implementação de uma proposta para a solução deste problema.

Os experimentos realizados comparam o desempenho do meta-escalonador com outros quatro algoritmos de Balanceamento de Carga presentes no *framework* de LB do Charm++, além de comparar com o cenário em que a aplicação não realiza o balanceamento. Os resultados obtidos mostram que a solução proposta foi capaz de atingir um aumento no desempenho de 1.63% se comparado ao segundo melhor algoritmo do contexto analisado, e 11.34% melhor do que o cenário em que não ocorre balanceamento de carga. Adicionalmente, o meta-escalonador cumpriu com o objetivo de se adaptar ao comportamento observado da aplicação em cada instante do balanceamento de carga, realizando a troca do algoritmo de LB em tempo de execução e mostrando-se mais vantajoso do que a abordagem estática de escolha do algoritmo.

Palavras-chaves: *HPC*. balanceamento de carga. aprendizagem de máquina. escalonamento global.

Lista de ilustrações

Lista de tabelas

Listagens

Lista de Algoritmos

Siglas

AMR *Adaptative Mesh Refinement*

API *Application Programming Interface*

CC-NUMA *Cache-Coherent Non-Uniform Memory Access*

Charm++ RTS *Charm++ Runtime System*

CPU *Central Processing Unit*

DSM *Distributed Shared Memory*

E/S Entrada e Saída

GPU *Graphics Processing Unit*

HPC *High Performance Computing*

IA Inteligência Artificial

KNN *K-Nearest Neighbors*

LB *Load Balancing*

MIMD *Multiple Instruction Multiple Data*

MISD *Multiple Instruction Simple Data*

ML *Machine Learning*

MNIST *Modified National Institute of Standards and Technology*

MPI *Message Passing Interface*

NUMA *Non-Uniform Memory Access*

OpenMP *Open Multi-Processing*

PE *Processing Element*

RAM *Random-Access Memory*

RPC *Remote Procedure Call*

SIMD *Single Instruction Multiple Data*

SISD *Single Instruction Simple Data*

SMP *Symmetric Multiprocessing*

SO Sistema Operacional

SPMD *Single Program, Multiple Data*

UMA *Uniform Memory Access*

Sumário

1 Introdução

Um dos principais elementos que contribuem para o avanço da Ciência é o desempenho das aplicações utilizadas em grandes centros de pesquisa. Diversas áreas no contexto de Pesquisas Espaciais, por exemplo, necessitam de simulações numéricas complexas e de alto desempenho para que se possa alcançar resultados precisos e em grande escala (MARCO et al., 2017; MANNEL, 2018).

Assim, para satisfazer essa necessidade de alto poder computacional, plataformas de Computação de Alto Desempenho, em inglês *High Performance Computing* (HPC), são utilizadas em conjunto com linguagens e interfaces de computação paralela. Um dos elementos que levam a complexidade no escalonamento de aplicações é a dependência dessas tarefas, visto que é necessário estimar quando uma delas será iniciada e encerrada para que as tarefas dependentes a ela possam iniciar execução (RODAMILANS, 2009).

Atualmente, há um crescimento constante e significativo na quantidade de núcleos nos processadores, e consequentemente, as plataformas mais recentes acabam por conter centenas de milhares de núcleos. Mas para extrair todo o potencial desse aumento do número de núcleos, é preciso adaptar e otimizar as aplicações que farão uso deles. Por conta disso, o desenvolvimento de programas paralelos possui extrema importância para o uso das plataformas computacionais atuais, visto que o paralelismo de processadores e plataformas tende a aumentar (PILLA; MENESES, 2015; KANG; LEE; LEE, 2015).

Uma vez que os tipos de aplicações que executam em ambientes de larga escala possuem um objetivo complexo, diversos elementos estão envolvidos de acordo com as características únicas de cada aplicação e dos cenários que as mesmas estão sendo executadas. Assim, a solução para atingir esse objetivo é decompor a aplicação principal em tarefas menores, onde cada tarefa realiza um trabalho específico e diferente das outras tarefas, para que depois seja possível fazer a união desses resultados menores para compor o resultado final. Deve-se levar em consideração que nem sempre é possível prever a quantidade de trabalho que cada tarefa receberá antes da aplicação iniciar a sua execução. Uma irregularidade de carga poderá surgir das tarefas que realizam diferentes papéis na aplicação, além da possibilidade da carga mudar dinamicamente ao longo da execução (PILLA, 2014; RASHID; BANICESCU; CARINO, 2008; MENESES; KALE; BRONEVETSKY, 2011).

Em grande parte das situações, o sobrecusto relacionado às diferenças de carga nos processadores é o fator dominante responsável por encerrar suas respectivas execuções em momentos diferentes. Alguns processadores podem permanecer ociosos enquanto outros estão sobrecarregados, e como cada tarefa possui um objetivo diferente, a consequência é que cada uma delas terá uma carga computacional diferente. Assim, pode-se dizer que uma

aplicação é desbalanceada se um número significativo de nodos computacionais dependem que outros terminem algum tipo de trabalho, resultando em desperdício de recursos e de energia (RASHID; BANICESCU; CARINO, 2008; FREITAS; PILLA, 2017).

Aplicações científicas como simulações de partículas têm cargas de trabalho imprevisíveis, isto é, variam de acordo com o fluxo de computação. Para atingir um alto desempenho nessas classes de aplicação, é necessário que a carga de trabalho seja distribuída entre os processadores dinamicamente. Mas os requisitos na escolha do algoritmo de balanceamento de carga variam de acordo com a natureza de cada contexto. Ou seja, requisitos para uma aplicação de simulação de partículas serão diferente daqueles escolhidos para aplicações como Refinamento Adaptativo de Malhas, em inglês *Adaptative Mesh Refinement* (AMR) (CHAUBE; CARINO; BANICESCU, 2007). Adicionalmente, aplicações paralelas, como as de dinâmica molecular, têm o balanceamento de carga como aspecto crucial para um bom desempenho em máquinas paralelas de larga escala (BATHELE; KALE; S., 2009).

Selecionar um algoritmo de escalonamento eficiente entre os disponíveis atualmente para alcançar o balanceamento de carga no contexto de aplicações que são executadas em ambientes imprevisíveis é uma tarefa não trivial. O balanceamento de carga pode ser necessário em diferentes partes da aplicação e cada uma dessas partes pode precisar de outros algoritmos de escalonamento para atingir um desempenho ótimo. Uma maneira eficiente de escalar essas aplicações é realizando uma redistribuição de cargas de trabalho, mas é possível que efeitos indesejados surjam com esse processo (RASHID; BANICESCU; CARINO, 2008; FREITAS; PILLA, 2017). Isso resulta em comportamentos irregulares observados dinamicamente, levando a um desbalanceamento de carga dos recursos computacionais e a um custo maior de comunicação. Dessa forma, o desempenho e escalabilidade da aplicação são comprometidos visto que processadores podem ficar ociosos, o que acarreta um desperdício computacional e de energia.

Os algoritmos de escalonamento global usados em balanceadores de carga gerenciam a distribuição das tarefas decompostas da aplicação entre os recursos disponíveis em um ambiente paralelo. O principal ponto é que encontrar uma distribuição de tarefas ótima é um problema NP-Difícil e, por conta disso, existem algoritmos que são baseados em diferentes heurísticas para serem usados em diferentes aplicações científicas (LEUNG, 2004). Citando como exemplo, um sistema de tempo real distribuído é composto por tarefas que se comunicam, tornando a distribuição dessas T tarefas para P processadores um desafio. É necessário uma heurística específica para encontrar uma alocação ótima, como a estratégia *Simulação de Recozimento*, ou *Simulated Annealing* em inglês (TINDELL; BURNS; WELLINGS, 1992). Assim, deve-se ter em mente que cada algoritmo de escalonamento global tem como foco diferentes objetivos e leva em consideração as características específicas da plataforma computacional alvo.

Para ilustrar esse fato, pode-se ter a situação em que um escalonador tem como pri-

oridade reduzir o consumo de potência de uma plataforma, enquanto outro concentra seu foco em reduzir o tempo de execução das iterações da aplicação (PILLA, 2017). Logo, *não há um algoritmo que seja ideal para todas as situações*, uma vez que cada aplicação científica é executada sob diferentes condições, sejam elas o tipo da plataforma, o objetivo que se quer atingir, as características que compõem a aplicação, ou outros elementos que influenciam sua singularidade. Além disso, mesmo que um algoritmo cumpra com o objetivo para o qual foi desenvolvido, pode ocorrer de outro aspecto ser impactado negativamente.

No caso de uma aplicação que é caracterizada por possuir tarefas com alto dinamismo de carga, pode haver a necessidade da *migração de uma quantidade maior de tarefas* para atingir um equilíbrio. O sobrecusto do processo de migração de tarefas pode acabar prejudicando o desempenho, chegando ao ponto em que a redistribuição de trabalho torne a aplicação mais lenta do que na situação em que a redistribuição não é realizada. Além disso, um algoritmo de escalonamento escolhido estaticamente pode realizar um bom trabalho no início da execução da aplicação, mas posteriormente tornar-se inapropriado. Uma entidade mais inteligente é necessária para selecionar dinamicamente algoritmos de escalonamento ao longo da execução (RASHID; BANICESCU; CARINO, 2008).

Assim, nota-se a necessidade de um meta-escalonador que realize uma análise individual para cada aplicação científica de forma adaptativa. Esse meta-escalonador levará em conta as diferentes características que compõem o contexto onde a aplicação está sendo executada, sejam elas o instante do tempo, aspectos da plataforma paralela da execução, ou características da própria aplicação. É esperado que tratando do problema de escalonamento global no contexto de aplicações científicas, estas terão um aumento no desempenho e acabarão por produzir resultados mais precisos e em maior escala. Isso abre portas para novas áreas que poderão ser exploradas ou obter respostas mais próximas da realidade das aplicações que atualmente não podem explorar os recursos disponíveis que não estão sendo aproveitados.

1.1 Objetivos

Para que o desenvolvimento da solução ao problema exposto seja feito através de uma base bem estruturada, são identificados o objetivo geral e os objetivos específicos nas subseções seguintes.

1.1.1 Objetivo Geral

O principal desafio é gerenciar a execução de uma aplicação científica para explorar ao máximo os recursos de uma plataforma paralela e então, atingir desempenho e escalabilidade (PILLA, 2014). Portanto, uma vez que o problema do escalonamento global foi identificado no contexto de aplicações científicas e seu impacto negativo no avanço da Ciên-

cia, este *Trabalho de Conclusão de Curso* tem como objetivo a implementação de um meta-escalonador adaptativo orientado à carga de trabalho das tarefas que compõem uma dada aplicação científica que executará em plataformas paralelas. O meta-escalonador fará uso de técnicas de *Aprendizagem de Máquina* uma vez que a tomada de decisão ocorrerá de forma adaptativa para cada cenário em aplicações diferentes e por consequência, conjunto de características distintas.

1.1.2 Objetivos Específicos

A fim de tornar possível que as aplicações atuais possam alcançar tempos de execução melhores através do uso do meta-escalonador, é necessário compreender e definir os conceitos envolvidos no escopo deste trabalho. Logo, têm-se como objetivos específicos:

1. Definição das características que afetam o desempenho das aplicações científicas nos aspectos de desbalanceamento de carga e comunicações custosas;
2. Definição de métricas relacionadas ao desempenho de núcleos de processamento e transferência de dados;
3. Definição de métricas para a compreensão e comparação de algoritmos de escalonamento global;
4. Estudo e escolha de um algoritmo de aprendizado de máquina adequado para ser utilizado no meta-escalonador;
5. Implementação do meta-escalonador em um ambiente de programação paralela.

Por fim, o objetivo final deste projeto é a realização da análise dos resultados obtidos das execuções do meta-escalonador. Métricas serão definidas para verificar se, dado um cenário de execução composto por características individuais, a tomada de decisão do melhor algoritmo de escalonamento será feita de fato da melhor forma. Isto é, se o algoritmo escolhido satisfaz as necessidades de cada aplicação que o executará. Assim, pode-se obter comparações dos resultados obtidos do trabalho implementado com o estado da arte, verificando as melhorias que essa nova abordagem poderá trazer.

1.2 Organização do Texto

O Capítulo 2 engloba a fundamentação teórica necessária para a compreensão deste trabalho, a qual servirá de base para o desenvolvimento da parte prática. O Capítulo 3 introduz a proposta para a solução do problema aqui apresentado, juntamente com o fluxo de projeto utilizado para a implementação do meta-escalonador. Já o Capítulo 4 apresenta

a descrição da plataforma experimental, o *benchmark* sintético utilizado e a análise dos resultados obtidos na fase de experimentação. Por fim, o Capítulo 5 contém as conclusões e considerações finais deste trabalho.

2 Fundamentação Teórica

Este capítulo engloba a base de conhecimento necessária para a compreensão e desenvolvimento do presente projeto. Inicialmente, a área de Computação Paralela será introduzida na Seção 2.1, envolvendo conceitos como Arquiteturas Paralelas (Seção 2.1.1) e Ambientes de Programação (Seção 2.1.2). Em seguida, serão apresentados conceitos básicos de Balanceamento de Carga na Seção 2.2 e de Aprendizagem de Máquina na Seção 2.3.

2.1 Computação Paralela

A Computação Paralela faz uso de múltiplos processadores que trabalham juntos ao mesmo tempo para um propósito em comum. Como consequência dessa cooperação, é possível executar bilhares de instruções por segundo, aumentando o desempenho da aplicação sendo executada. Além de atingir uma maior velocidade de execução, computadores paralelos podem fornecer sistemas mais confiáveis do que máquinas de único processador, visto que se uma determinada *Central Processing Unit* (CPU) no sistema paralelo falhar, este ainda pode continuar a operação sem que haja uma falha geral (ZARGHAM, 1996).

As seções seguintes introduzem os conceitos principais de arquiteturas paralelas e de ambientes de programação relevantes para a execução deste trabalho.

2.1.1 Arquiteturas Paralelas

Tarefas com alta carga de processamento, como as encontradas em aplicações científicas, possuem alta demanda por recursos computacionais. Arquiteturas Paralelas têm o propósito de aumentar a capacidade de processamento a fim de executar o trabalho necessário em alto desempenho. Para compreender as diferentes arquiteturas de computadores paralelos, a taxonomia de Flynn (FLYNN, 1972) as classifica com base na existência de um fluxo único ou múltiplo de instruções e de dados.

Na classe *Single Instruction Simple Data* (SISD), o sistema opera sobre apenas uma instrução por ciclo e faz uso de apenas um conjunto de dados, sendo um modelo de programação sequencial. É identificada como uma arquitetura de computador serial pois apenas um fluxo de instrução está sendo processado na CPU durante um ciclo de relógio e apenas um fluxo de dados é usado como entrada (VEERASAMY, 2010).

Para a classe *Multiple Instruction Simple Data* (MISD), é raro encontrar exemplos reais de computadores que a represente. MISD é caracterizada por executar várias instruções em diferentes CPUs usando um único conjunto de dados, onde cada unidade de processamento opera de forma independente. Assim, múltiplos programas serão executados em

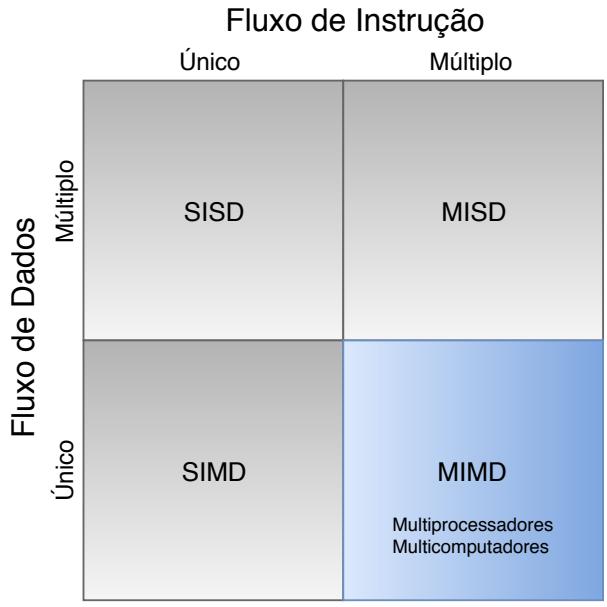
diferentes processadores e utilizando os mesmos valores para a computação (VEERASAMY, 2010).

A classe *Single Instruction Multiple Data* (SIMD) permite que uma mesma instrução possa operar sob diferentes conjuntos de dados de forma simultânea (ZARGHAM, 1996). Um exemplo típico de arquitetura paralela que pertence a esta classe são as *Graphics Processing Unit*s (GPUs).

Por fim, *Multiple Instruction Multiple Data* (MIMD) é a classe que inclui máquinas com mais de uma unidade de processamento, onde cada CPU é capaz de executar múltiplas instruções aplicadas em conjuntos de dados diferentes e simultaneamente. Neste cenário, há a possibilidade de ganho de eficiência através do uso de processamento concorrente, pois se houver uso da operação paralela dos processadores, múltiplos processos são executados de forma concorrente uns aos outros (ZARGHAM, 1996).

A Figura 1 ilustra as classes de arquiteturas de computadores de acordo com a Taxonomia de Flynn. Uma vez identificadas, pode-se separar as arquiteturas paralelas em dois grupos principais: *multiprocessadores* e *multicomputadores*. Esses dois grandes grupos se encaixam na classe de MIMD e são descritos nas subseções seguintes.

Figura 1 – Taxonomia de Flynn para Arquitetura de Computadores.



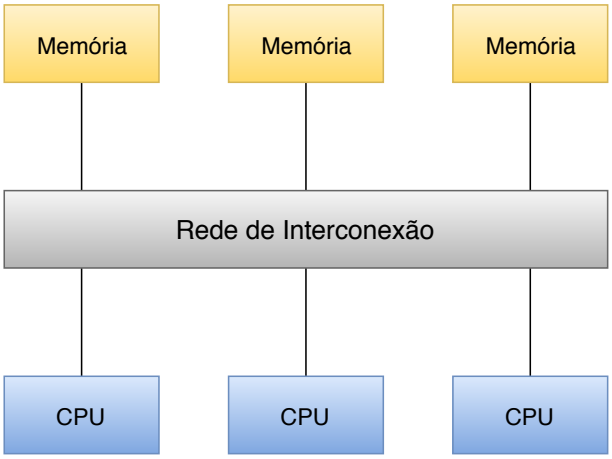
Fonte: produzido pela autora.

2.1.1.1 Multiprocessadores

Sistemas multiprocessados são caracterizados por compartilhar a *Random-Access Memory* (RAM) entre todas as CPUs, fazendo uso de uma rede de interconexão. Uma vez que existe a possibilidade de múltiplos processadores alterarem a memória concorrentemente,

situações de condições de corrida podem acontecer, levando à inconsistência no conteúdo modificado (TANENBAUM, 2007). A Figura 2 ilustra a arquitetura de um sistema multiprocessado, onde há uma rede de interconexão que torna possível o compartilhamento da memória com todas as CPUs do sistema.

Figura 2 – Arquitetura genérica de um multiprocessador.



Fonte: produzido pela autora.

A comunicação entre *threads* de um mesmo processo ocorre através de instruções de acesso à memória do tipo *load* e *store*. Assim, há diferentes tipos de acesso à memória do sistema que diferem na existência de um acesso uniforme à memória ou no uso de uma *cache* (NAVAUX; ROSE; PILLA, 2011).

Em arquiteturas com acesso uniforme à memória, denominadas *Uniform Memory Access* (UMA), a memória é centralizada, e portanto compartilhada, com a característica de possuir a *mesma distância* de todas as CPUs. Esse tipo de sistema também é conhecido por *Symmetric Multiprocessing* (SMP) (GARAVEL; VIHO; ZENDRI, 2001) e seu acesso à memória é considerado uniforme pois a latência independe de qual processador faz a requisição (RAJPUT; KUMAR; PATLE, 2012). Assim, o tipo de rede de interconexão não afeta o tempo uniforme de acesso, podendo ser um barramento ou entrelaçado, desde que mantenha a mesma latência de acesso para todos os processadores (NAVAUX; ROSE; PILLA, 2011).

Arquiteturas do tipo *Non-Uniform Memory Access* (NUMA), por outro lado, são caracterizadas por possuir diversos processadores, os quais estão localizados em nodos que compõem o sistema. Cada nodo é formado por uma CPU e um bloco de memória RAM (HOLLOWELL et al., 2015). O acesso aos blocos de memória é feito de maneira transparente, pois o espaço de endereçamento físico é único. Todavia, o tempo de acesso aos blocos de memória depende da distância entre a CPU e o bloco de memória onde o dado está localizado, acarretando no aumento do custo e na não uniformidade da latência de acesso (MUDDUKRISHNA; JONSSON; BRORSSON, 2015b). As características arquiteturais dos sistemas do

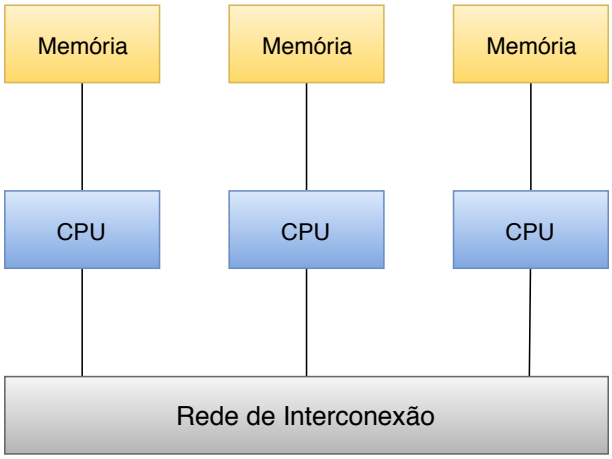
tipo NUMA tornam possível um aumento significativo do número de processadores, ganhando alta escalabilidade se comparado à arquitetura UMA (GARAVEL; VIHO; ZENDRI, 2001).

Tendo em vista que o acesso a um bloco de memória remoto é mais lento do que em uma memória local, é possível adicionar memórias *cache* em cada um dos nodos de CPU. Esse tipo de sistema é chamado de *Cache-Coherent Non-Uniform Memory Access* (CC-NUMA) (TANENBAUM, 2007), pois mantém a coerência entre múltiplas cópias dos mesmos dados (GARAVEL; VIHO; ZENDRI, 2001).

2.1.1.2 Multicomputadores

Em arquiteturas multicomputadas, cada CPU possui a sua própria memória local e privada para si. Um nodo é uma unidade individual de processamento, sendo composto pelo processador com sua respectiva memória local e sua porta de entrada e saída (E/S). Assim, um processador possui acesso direto à sua memória e não às memórias remotas, sendo necessário utilizar um mecanismo de troca de mensagens ao invés de variáveis compartilhadas em memória (ZARGHAM, 1996). A Figura 3 ilustra a arquitetura de um sistema multicomputado, onde cada CPU tem acesso a sua própria memória e há uma rede de interconexão que possiblita a interação entre CPUs.

Figura 3 – Arquitetura genérica de um multicomputador.



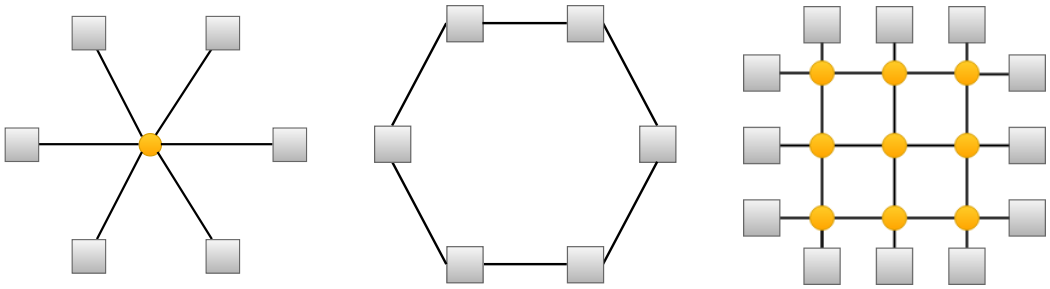
Fonte: produzido pela autora.

O Sistema Operacional (SO) provê uma forma de envio e recebimento de mensagens, a qual será implementada em uma biblioteca de procedimentos e então disponibilizada aos processos usuários. O uso de chamadas de sistema como *send* e *receive* é uma abordagem de comunicação, tendo como variação a característica de serem bloqueantes (chamadas *síncronas*) ou não bloqueantes (chamadas *assíncronas*).

Outra maneira de implantar a comunicação entre processos em diferentes CPUs é a realização de *Remote Procedure Call* (RPC). Diferente do uso de *send* e *receive* que realiza operações de Entrada e Saída (E/S) visível ao programador, esta usa a ideia de invocação remota de procedimentos pelo *cliente* para o *servidor*, passando a impressão de acontecer localmente por conta da abstração fornecida. Por fim, pode-se citar o possível uso de *Distributed Shared Memory* (DSM), onde há a ilusão de que a memória é compartilhada (TANENBAUM, 2007).

A organização dos canais de comunicação entre processadores segue uma topologia de rede, que são conectados às CPUs por uma interface de comunicação (ZARGHAM, 1996). Para sistemas pequenos, pode-se usar um *switch* para interconectar os nodos processadores formando a topologia em *estrela* (Figura 4a), ou até mesmo em *anel* (Figura 4b), caso não seja usado um switch central. Já o uso de *grid* ou malha (*mesh*) facilita a escalabilidade do sistema (TANENBAUM, 2007), permitindo que novos nodos processadores sejam interconectados entre os já existentes (Figura 4c).

Figura 4 – Topologia de Interconexão



(a) Estrela (b) Anel (c) Malha

Fonte: produzido pela autora.

2.1.2 Ambientes de Programação

Ambientes de Programação paralela são ferramentas que fornecem suporte ao desenvolvimento de aplicações de HPC, permitindo fazer uso do *software* para explorar mais dos recursos de *hardware*, como o processador. O uso de tais ferramentas diminui a complexidade da programação da aplicação, pois fornece um ambiente com bibliotecas e linguagens que facilitam a implementação dos algoritmos paralelos.

A fim de alcançar um alto desempenho, suportar paralelismo e concorrência entre processos e extrair ao máximo os benefícios dos recursos computacionais, é necessário utilizar tais ambientes para o desenvolvimento de aplicações científicas de HPC. As seções seguintes introduzem algumas das ferramentas mais importantes para atingir tais objetivos de acordo com diferentes arquiteturas.

2.1.2.1 OpenMP

Open Multi-Processing (OpenMP) é uma *Application Programming Interface* (API) de alta portabilidade que usa multiprocessamento em memória compartilhada e possui o propósito de desenvolver programas paralelos com memória compartilhada. Assim, a API fornece uma alta abstração ao encapsular detalhes de implementação em um conjunto de diretivas de compilador. Estas são responsáveis pela criação de *threads*, sincronização de operações e gerenciamento da memória compartilhada. A comunicação entre *threads* pode ser eficiente uma vez que os programas são compilados para serem *multithreaded*, onde todas as *threads* de um programa em OpenMP compartilham o mesmo espaço de endereçamento de memória (KANG; LEE; LEE, 2015).

Dessa forma, o programador é capaz de expressar unidades lógicas de paralelismo, as quais serão escalonadas em *threads* por um sistema de execução. Esse modelo de programação viabiliza a característica de paralelização incremental e introduz o conceito de tarefas (*tasks*) como paralelismo lógico, utilizando o conceito de *composição* do mesmo (MUDDUKRISHNA; JONSSON; BRORSSON, 2015a). Assim, a computação paralela pode ser facilmente adaptada a um *software* sequencial já existente (BRUNST; MOHR, 2008).

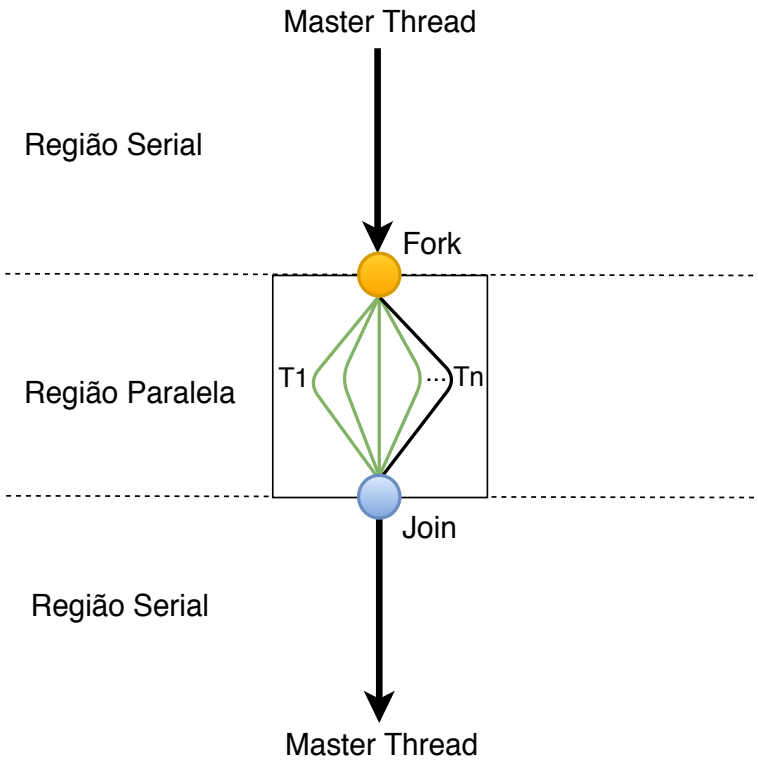
Além das diretivas de compilador, a API provê rotinas de biblioteca e variáveis de ambiente para especificar *como* o programa deve executar em paralelo. O desenvolvedor da aplicação possui uma visão de alto nível do paralelismo, deixando os detalhes de gerenciamento de concorrência para o compilador. Portanto, as notações do OpenMP podem ser adicionadas a um programa sequencial nos trechos que devem ser paralelizados, ou seja, dividindo o trabalho entre *threads* que serão executadas em diferentes CPUs (BALAJI, 2015).

Essa estrutura em bloco que alterna entre seções sequenciais e paralelas seguem o modelo *fork*/*join*. Quando a execução do programa chega em um bloco paralelo, uma única *thread* de controle, denominada *master thread* é dividida em um determinado número de *worker threads*. Após executarem o trabalho existente na seção e alcançar o fim do bloco, as *worker threads* são sincronizadas e unidas novamente em uma *master thread*, que por sua vez, continua a execução do código normalmente (KANG; LEE; LEE, 2015). A Figura 5 ilustra esse processo.

2.1.2.2 MPI

Message Passing Interface (MPI) é um padrão de interface de comunicação que faz uso do modelo de programação de memória distribuída e se caracteriza por ser um sistema baseado em bibliotecas. Além de providenciar características que expressam a comunicação necessária em programas paralelos, possui suporte para sistema de arquivo E/S paralelo e suporte ao modelo de programação MIMD (BALAJI, 2015). Adicionalmente, pode-se dizer que a API é centrada a processos e possui uma visão localizada dos dados da aplicação den-

Figura 5 – Esquema do modelo *fork*/*join*.



Fonte: produzido pela autora.

tro do contexto de um processo. Além disso, é orientada ao usuário, uma vez que o fluxo do programa é explicitamente definido pelo programador (NIKHIL et al., 2015).

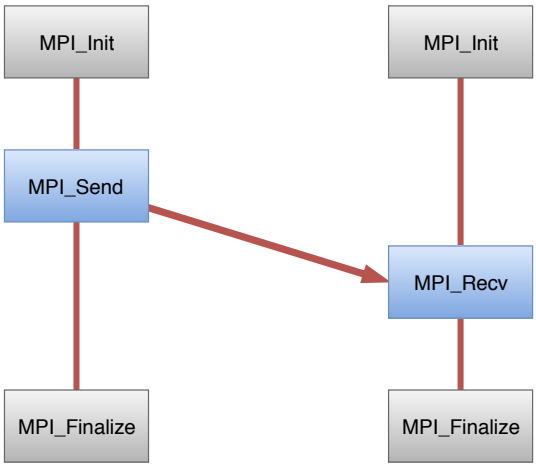
Duas importantes características que MPI fornece são o alto desempenho e escalabilidade para sistemas grandes (BRUNST; MOHR, 2008). Sua alta portabilidade se dá pelas implementações da API que estão disponíveis para múltiplas plataformas, como *laptops*, *desktops*, *clusters*, servidores e supercomputadores. Além disso, diferentes arquiteturas e sistemas operacionais são compatíveis com o modelo, tornando-o uma importante ferramenta para aplicações científicas paralelas (BALAJI, 2015).

Cada processo possui o seu próprio espaço de endereçamento e a comunicação entre processos ocorre através do acesso a seus respectivos espaços de endereçamento. É de responsabilidade do programador particionar a carga de trabalho e mapear as tarefas para cada processo (KANG; LEE; LEE, 2015), que são mapeados um por um para processadores físicos (NIKHIL et al., 2015).

A API inclui modelos de comunicação como o envio *ponto a ponto* e operações *coletivas* ou *globais*, onde a troca de mensagens é restrita a um grupo de processos especificado pelo usuário. A troca de mensagens ponto a ponto ocorre entre dois processos de maneira direta, enquanto a coletiva é caracterizada por um processo se comunicar com um grupo de processos simultaneamente (GROPP et al., 1996).

A Figura 6 ilustra uma comunicação genérica ponto-a-ponto. A rotina *MPI_Init* inicializa o ambiente de execução MPI, sendo invocada inicialmente antes de outras funções. A operação bloqueante *MPI_Send* envia uma mensagem para o segundo processo, sendo que a rotina só retorna quando a mensagem estiver disponível em um *buffer* da aplicação. Após, a operação bloqueante *MPI_Recv* recebe a mensagem e não retorna até que o dado recebido esteja disponível no buffer da aplicação. Por fim, *MPI_Finalize* encerra e execução do ambiente MPI (BARNEY, 2019).

Figura 6 – Comunicação MPI ponto a ponto.



Fonte: produzido pela autora.

Apesar de possuir essas características de comunicação global e fluxo de controle global, o MPI pode não ser a melhor alternativa para aplicações dinâmicas com fluxo de controle dependentes de dados. A razão é que há um suporte limitado para balanceamento de carga e para tratar interações baseadas em envio de mensagens (NIKHIL et al., 2015). Sendo assim, há a necessidade de outro ambiente de programação paralela para o desenvolvimento de aplicações com alto dinamismo e de larga escala. Para ilustrar essa situação, pode-se citar aplicações científicas como a simulação de dinâmica molecular (MURTY; OKUNBOR, 1999), cosmologia (BRYAN; ABEL; NORMAN, 2001) ou química quântica (HANS et al., 1995), que fazem uso de outro ambiente para satisfazer seus comportamentos irregulares. Um exemplo de ambiente adaptado para tais aplicações é o Charm++ (BILGE et al., 2016).

2.1.2.3 Charm++

O desenvolvimento de aplicações paralelas de larga escala traz o aumento de complexidade, visto que atualmente há desafios como heterogeneidade e falhas de sistema. As variações dinâmicas presentes em aplicações científicas tornam necessário a análise de aspectos como sobredecomposição (*overdecomposition*), execução orientada a mensagens as-

síncronas, migrabilidade (*migratability*), adaptatividade (*adaptive*) e introspecção em tempo de execução. Assim, o *framework* de programação paralela Charm++ foi projetado para satisfazer esses aspectos, usando um sistema de execução adaptativo, denominado *Charm++ Runtime System* (Charm++ RTS), de forma a facilitar a implementação de aplicações escaláveis e de alta complexidade (BILGE et al., 2014). Adicionalmente, devido a suas características arquiteturais, o Charm++ abstrai detalhes de funcionamento do desenvolvedor da aplicação, além de oferecer portabilidade para plataformas que contém um ou mais nodos (PILLA, 2014).

A *sobredecomposição* divide a computação em unidades independentes, resultando em unidades de trabalho e de dados que serão mapeadas para cada elemento processador pelo Charm++ RTS, denominado *Processing Element* (PE) (BILGE et al., 2014). Esse mapeamento estático inicial pode ser alterado durante a execução, migrando tarefas e dados a outros PEs caso a aplicação esteja em um estado desbalanceado em termos de carga de trabalho (MENON, 2012).

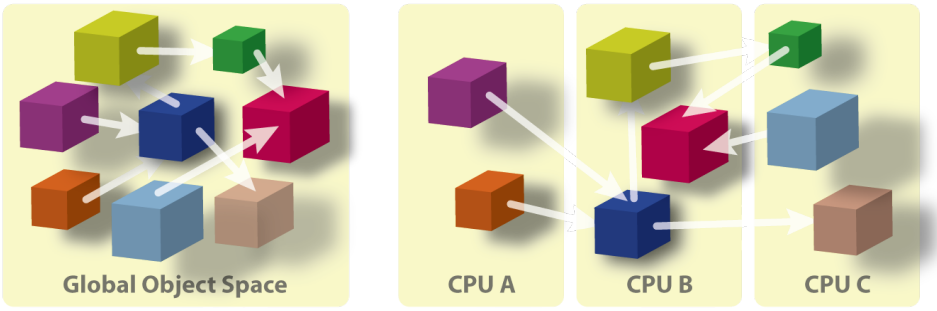
Ao contrário de pré-programar uma sequência de unidades a serem escalonadas, utiliza-se a *execução guiada por mensagens* para selecionar qual das unidades será a próxima a ser executada em um PE. Assim, uma dada unidade só será escalonada para execução se uma mensagem for enviada a ela. Já a *migrabilidade* garante que as unidades de trabalho e de dados não são fixas em um dado PE, pois o Charm++ RTS é capaz de migrá-las a qualquer outro PE do sistema durante a execução, conforme seja necessário (BILGE et al., 2014).

Este ambiente de programação paralela utiliza a linguagem de programação *C++* e decompõe o domínio da aplicação em *chares*, que representam objetos migráveis. A comunicação se dá pela troca de mensagens entre esses objetos (FREITAS, 2017), e assim, sua base é um modelo de execução orientado a mensagens assíncronas, onde cada dado e computação são encapsulados em *chares*, e estes mapeados para um core específico onde será herdado localidade de dados. Os métodos de um *chare* só serão invocados quando uma mensagem for enviada a ele (BAK et al., 2017). A Figura 7 ilustra o mapeamento de *chares* para diferentes CPUs, onde cada seta representa a comunicação realizada entre eles, ou seja, o envio de mensagem entre *chares*.

Em contraste ao MPI, Charm++ é centrado a trabalho e também provê uma visão localizada dos dados da aplicação, mas dentro de um objeto C++. Além disso, é dito ser orientado ao sistema, uma vez que o Charm++ RTS decide qual computação será executada com base na disponibilidade de dados dos *chares*. Isso permite a existência de um fluxo de controle concorrente em uma execução direcionada à disponibilidade de dados e direcionada ao Charm++ RTS (NIKHIL et al., 2015).

O sistema permite que informações do programa sejam coletadas durante a execução do mesmo, e assim, o próprio programa tem conhecimento da carga de trabalho existente para as tarefas e pode realizar estratégias de balanceamento de carga para o au-

Figura 7 – Comunicação entre chares.



Fonte: (LABORATORY, 2017b).

mento do desempenho (FREITAS, 2017). Além disso, mensagens inesperadas são tratadas facilmente por conta da natureza reativa dos *chares*, sendo uma característica de grande importância para aplicações que possuem comportamento irregular (XIANG; LAXMIKANT; RASMUS, 2015).

2.2 Balanceamento de Carga

O Balanceamento de Carga, em inglês *Load Balancing* (LB), é um ponto crítico para aplicações de diferentes propósitos e contextos, como para a eficiência nas operações de redes *peer-to-peer* (KARGER; RUHL, 2004), escalonamento de tarefas em Computação em Nuvem (DHINESH; KRISHNA, 2013), reduzir custo de energia com balanceamento de carga geográfico em *Green Computing* (LIU et al., 2011) e para simulações científicas adaptativas (SCHLOEGEL; KARYPIS; KUMAR, 2000).

Em sistemas de memória distribuída, o balanceamento de carga é o processo de redistribuir o trabalho entre recursos de *hardware* com o objetivo de melhorar o desempenho ao transferir tarefas de recursos sobrecarregados para os recursos subcarregados. Uma aplicação paralela possui a carga desbalanceada quando o trabalho é distribuído de uma maneira irregular, de forma que os processadores com menos tarefas devem esperar o fim da execução dos processadores sobrecarregados (PADUA, 2011).

Juntamente com a chegada da computação em petaescala, há o desafio de escrever programas paralelos que explorem de fato a potência de processamento, além de manter a alta escalabilidade. O desbalanceamento de carga de uma determinada aplicação é um dos fatores que afetam o desempenho e a escalabilidade da mesma (MENON, 2012). Além disso, a maioria das simulações científicas atualmente usam um modelo de programação paralela *Single Program, Multiple Data* (SPMD), o qual realiza a computação simultaneamente em múltiplos processadores. Entretanto, o ponto de sincronização entre eles terá que esperar

pelo fim da computação da CPU mais lenta (PEARCE, 2014).

Além do aumento da complexidade das aplicações, estas têm utilizado estruturas irregulares e técnicas de refinamento adaptativas. Consequentemente, a carga de computação varia *dinamicamente* durante a execução, como acontece em aplicações de simulação de tempo, estruturas dinâmicas e refinamento de malha adaptativo. Para esses tipos de programa, o balanceamento de carga e a frequência que este deve ser realizado são fatores críticos e que determinam a escalabilidade e o desempenho do sistema (BILGE et al., 2014). Portanto, a realização do processo de balanceamento de carga propriamente dito também possui um custo que pode não ser conhecido previamente (MENON et al., 2012) e que deve ser levado em consideração.

As aplicações científicas de HPC, como as de dinâmica molecular, possuem paralelismo de granularidade fina com o intuito de atingir um alto desempenho e escalabilidade, resultando na necessidade de uma alocação *adaptativa* de recursos para que a carga se mantenha balanceada. Entretanto, se torna um risco deixar nas mãos do programador da aplicação a responsabilidade de lidar com a alocação de recursos e com o balanceamento de carga (MENON, 2012). Assim, a distribuição de tarefas para processadores (*particionamento de tarefas*) tem um papel importante no contexto de HPC. A associação do trabalho para as CPUs disponíveis pode ocorrer no início de uma computação (*particionamento estático*) e a redistribuição do mesmo pode ser necessária durante a computação (*particionamento dinâmico*). É esperado minimizar o tempo total da solução ao atribuir tarefas aos processadores igualmente, ou seja, balanceando a carga de trabalho (DEVINE; BOMAN; KARYPIS, 2006).

As políticas do balanceamento de carga *estático* são baseadas em informações sobre o comportamento médio do sistema e as decisões de transferência são independentes do estado atual do sistema (KAMEDA et al., 2000). Essa estratégia funciona bem para aplicações que possuem um comportamento consistente e previsível, ou seja, uma vez balanceadas, permanecem balanceadas (PADUA, 2011). O objetivo de encontrar uma distribuição ótima engloba o desenvolvimento de heurísticas, como por exemplo, prioridades para tarefas maiores. Espera-se particionar as tarefas de tal forma que os processadores estejam igualmente ocupados e com o mínimo de sobrecusto de comunicação (MANEKAR et al., 2012). Apesar da viabilidade de particionar tarefas em várias partes, essas partições podem possuir diferentes tamanhos. Assim, depois de uma distribuição inicial entre os processadores físicos, alguns deles podem se tornar inativos mais cedo do que outros, surgindo a necessidade de um balanceamento de carga dinâmico para a transferência de trabalho a essas CPUs ociosas (KUMAR; GRAMA; VEMPATY, 1994).

No balanceamento de carga *dinâmico*, as políticas envolvem *reagir* ao estado atual do sistema em execução para a tomada de decisões de transferência, sendo considerado mais complexo do que o balanceamento estático (KAMEDA et al., 2000). Logo, faz-se uso de algoritmos para redistribuir o trabalho entre os processos de forma igualitária e durante

a execução da aplicação (PEARCE, 2014). Aplicações que possuem mudanças dinâmicas de comportamento necessitam de rebalanceamento periódico e fazem uso da estratégia dinâmica (PADUA, 2011). É preciso observar se o sobrecusto do Charm++ RTS é maior do que o custo do trabalho realizado ao mover uma tarefa de processadores sobrecarregados para processadores mais leves em carga. Caso a resposta seja positiva, então a eficiência se torna um problema e o objetivo nesse caso é a utilização máxima de todos os processadores, fazendo uso de algoritmos centralizados ou distribuídos (MANEKAR et al., 2012).

Portanto, torna-se claro que o comportamento das tarefas de uma aplicação pode afetar seu desempenho, uma vez que há a possibilidade de cada tarefa possuir cargas diferentes ou ainda ter sua carga ou padrão de comunicação modificados durante a execução do programa. Assim, uma aplicação possui *carga regular* se o tempo de computação das tarefas for similar. Similarmente, a aplicação possui *comunicação regular* se os seus grafos de comunicação são bem definidos, ou seja, o número de mensagens/bytes trocados é o mesmo entre diferentes tarefas. Por fim, uma aplicação com *carga dinâmica* é dada pela existência da variação da carga das tarefas durante a execução do programa. Já a *comunicação dinâmica* é caracterizada quando seu grafo de comunicação sofre modificação no decorrer do tempo (PILLA, 2014).

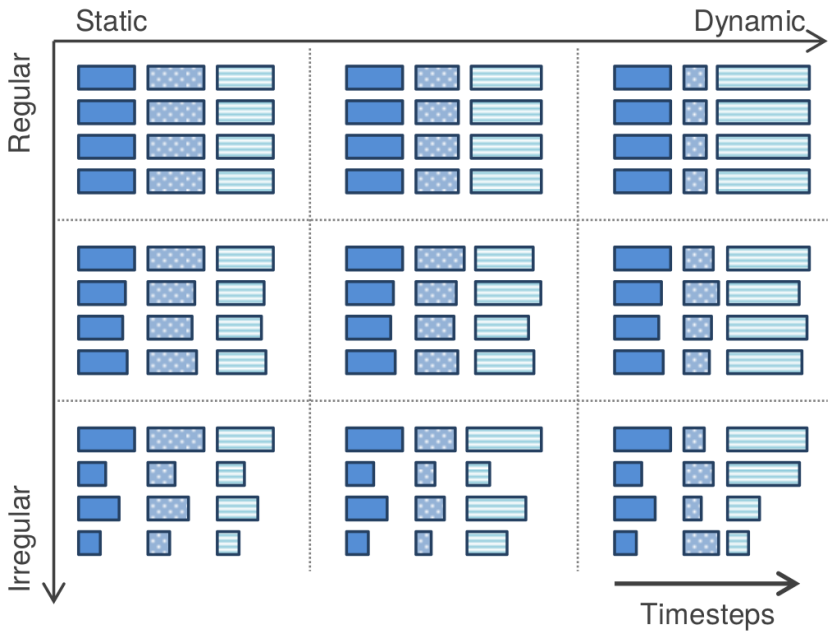
As ilustrações a seguir representam os cenários de regularidade e dinamicidade recém discutidos. A Figura 8 ilustra a disposição das cargas de tarefas de aplicações de acordo com o nível de dinamicidade e regularidade. Assim, uma aplicação estática e regular não tem as cargas de suas tarefas alteradas, enquanto uma aplicação dinâmica e irregular possui alto desbalanceamento de carga para cada tarefa. A Figura 9 mostra a troca de mensagens entre tarefas, sendo que as arestas mais grossas representam um volume de informação maior. Ou seja, uma aplicação que possui um comportamento mais dinâmico e irregular apresenta um padrão de comunicação também dinâmico e irregular.

2.2.1 Algoritmos de Balanceamento de Carga

As aplicações possuem um domínio, propósito e características que as tornam particulares. Logo, suas necessidades de balanceamento de carga podem variar de acordo com seus aspectos computacionais, o que leva à escolha entre múltiplos algoritmos de balanceamento de carga disponíveis atualmente. Os critérios de escolha envolvem identificar o aspecto crítico que deve ser melhorado em um programa, como por exemplo, reduzir a comunicação entre nós de um sistema distribuído ou minimizar a carga máxima em um nodo (PADUA, 2011).

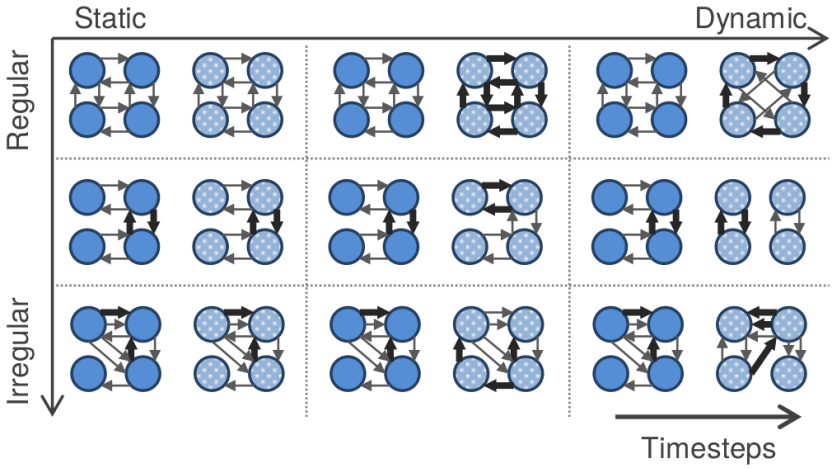
Para ilustrar esse processo, pode-se citar que o fato de uma aplicação com sobrecusto de comunicação usar uma estratégia de balanceamento de carga baseada em particionamento de grafos pode ser melhor do que a que estratégia que equilibra a carga de trabalho. Mas por outro lado, se a aplicação é pesada em termos de computação, então o

Figura 8 – Cenários de desbalanceamento de carga.



Fonte: (PILLA, 2014).

Figura 9 – Cenários de desbalanceamento de comunicação.



Fonte: (PILLA, 2014).

uso de um balanceamento de computação com a heurítisca gulosa é uma alternativa melhor (MENON, 2012). Abaixo serão apresentados alguns dos principais algoritmos usados no balanceamento de carga.

2.2.1.1 Particionamento de Grafos

Pode-se usar um grafo para representar o padrão de comunicação de um programa paralelo ao tratar as unidades de trabalho como nodos e a comunicação como arestas com

pesos. O algoritmo trabalha com base na divisão do grafo em pedaços menores, com o propósito de atribuir cada parte resultante a uma PE (PADUA, 2011).

O objetivo é mitigar o efeito de comunicações custosas ao reduzir o volume de comunicação entre as PEs e ao mesmo tempo evitando migrações de tarefas. Assim, a fase inicial de *coarsening* se dá por reduzir o número de vértices no grafo ao agrupar pares de vértices que possuem redes custosas entre eles. Esse processo é repetido até que a quantidade resultante de vértices seja menor do que um valor fixo, ou até que o último passo do processo não reduza o grafo suficientemente (PILLA, 2014).

Após obter o grafo particionado resultante, os passos seguintes envolvem o processo reverso, pois os nodos combinados são separados. Pode-se fazer uso de um algoritmo de refinamento em cada passo desse procedimento de *uncoarsening*, para que dessa forma, a qualidade de partição seja melhorada ao encontrar trocas de nodos vantajosas. Por fim, o usuário obtém o particionamento para a malha de entrada original.

A flexibilidade e generalidade é uma das vantagens deste algoritmo, visto que um grafo de comunicação pode ser construído com base em qualquer problema e independente da natureza do seu domínio. Por outro lado, o custo pode se tornar elevado, uma vez que se houver uma grande quantidade de partições, o problema se torna computacionalmente intenso (PADUA, 2011).

2.2.1.2 Bisseção Recursiva

Segundo (PADUA, 2011), o algoritmo de Bisseção Recursiva é uma técnica de dividir-e-conquistar, a qual reduz o problema de particionamento a uma série de operações de bisseção. O propósito é dividir o grafo objeto em duas partes iguais, e ao mesmo tempo, diminuir a comunicação entre as partes para permitir a subdivisão de cada metade recursivamente, até que a quantidade desejada de partições seja atingida.

Após a divisão inicial ser realizada, as duas metades divididas são independentes e podem ser computadas em paralelo. Assim, depois de n níveis de bisseção, haverá 2^n operações divididas independentes entre si. Além disso, operações de bisseção geométrica podem ser paralelizadas ao construir um histograma dos nodos a serem divididos.

A técnica de bisseção recursiva é uma forma rápida de computar partições para diversos problemas, além de possuir a vantagem de que o procedimento recursivo introduz paralelismo no processo de particionamento. Entretanto, as características do algoritmo de bisseção a ser usado causam grande impacto no desempenho da resolução ao problema, pois podem afetar a qualidade de particionamento e o custo computacional.

2.2.1.3 Prefixo Paralelo

O padrão de comunicação entre objetos pode não ser o foco principal de uma aplicação, levando a atenção à quantidade de trabalho igualmente dividida entre PEs. (PADUA, 2011) apresenta o algoritmo de Prefixo Paralelo (ou Soma de Prefixo) para contextos onde a carga deve ser distribuída de forma efetiva entre os processadores.

Usa-se um *array* local com pesos de unidades de trabalho em cada processador como entrada para o algoritmo. O funcionamento se dá pela computação da soma das primeiras i unidades para cada uma das n unidades na lista. Após a soma do prefixo ser calculada, pode-se dividir de forma igualitária todo o trabalho entre os processadores através do envio de cada unidade ao número do processador, o qual é dado pelo valor da sua soma de prefixo dividido pelo trabalho total dividido pelo número de processadores.

A vantagem deste algoritmo se caracteriza por sua alta velocidade de execução e baixo custo de implementação. Já sua desvantagem é dada por sua simplicidade, pois não leva em conta a estrutura da comunicação entre os processadores. Além disso, não há a tentativa de minimizar o volume de comunicação da partição resultante, tampouco de minimizar a quantidade de migração necessária.

2.2.2 Balanceamento de Carga em Arquiteturas Paralelas

Como dito anteriormente, o balanceamento de carga se mostra necessário para que a carga de trabalho seja dividida de forma igualitária entre PEs, havendo a possibilidade de aumentar o desempenho de um sistema (KUMAR; GRAMA; VEMPATY, 1994). Isso permite reduzir o tempo de ociosidade dos processadores e diminuir os custos de comunicação ao atribuir trabalhos relacionados para uma mesma CPU (PADUA, 2011). O Charm++ RTS é capaz de realizar um balanceamento de carga dinâmico, sendo de responsabilidade do usuário decidir apenas o período e a estratégia do balanceamento. Assim, é necessário observar o comportamento e características da aplicação em execução para que o usuário possa definir esses aspectos de forma correta (MENON, 2012).

Para que o *framework* de balanceamento de carga do Charm++ seja explorado ao máximo, o problema deve ser decomposto em um grande número de unidades de trabalho e dados. Caso seja detectado uma situação em que a carga do sistema não está balanceada, então as unidades particionadas são remapeadas para outros processadores. Logo, a combinação da sobredecomposição e migrabilidade permite que o Charm++ RTS atue dinamicamente sobre o sistema (MENON, 2016).

Adicionalmente, o *framework* inclui diferentes tipos de estratégias de LB, como a gulosa (*greedy*) e a de refinamento (*refinement*). A abordagem gulosa é caracterizada por *associar* o objeto mais pesado para o processador menos sobrecarregado, ocasionando uma comunicação significativa. Já a estratégia de refinamento migra objetos pesados dos proces-

sadores sobrecarregados para os mais leves, gerando menos comunicação (PEARCE, 2014).

As estratégias de balanceamento são implementadas no Charm++ sob a forma de *balanceadores de carga*. Como exemplo, pode-se citar o *GreedyLB*, o qual realiza o remapeamento das tarefas entre todas as PEs iniciando com as mais pesadas até que a carga se torne uniforme. Entretanto, além de não explorar o paralelismo pelo fato de realizar o processo de forma sequencial, pode acabar tendo um sobrecusto alto de remapeamento uma vez que essa técnica move muitas tarefas entre os processadores (FREITAS, 2017). Essa estratégia centralizada não considera a comunicação e nem a distribuição de tarefas atual, e portanto, realiza um número maior de migrações.

O *RefineLB* é um outro exemplo de balanceador de carga disponível no Charm++, mas este é baseado na técnica de refinamento, onde os objetos mais pesados são migrados de PEs sobrecarregados para PEs mais leves, até que a carga do sistema alcance um valor médio (ACUN; KALE, 2016). Apesar de possuir um sobrecusto alto por realizar o processo de forma sequencial, é uma abordagem um pouco menos agressiva se comparada ao método guloso, pois leva em conta a distribuição de trabalho atual entre as PEs, resultando em um número menor de migrações.

Já o foco do *GreedyCommLB* é a comunicação. Esse algoritmo é uma extensão do GreedyLB, mas faz uso do grafo de comunicação para análise. Assim, a carga de comunicação de uma tarefa é computada com base no volume de dados que a mesma envia para outras tarefas em outros núcleos. O algoritmo seleciona a tarefa de carga mais alta e a mapeia para o núcleo de menor carga total, incluindo carga de comunicação e de processamento (PILLA et al., 2015).

Por fim, é interessante mencionar o *GreedyRefineLB*, o qual faz uso de um algoritmo guloso que associa o objeto mais pesado para o processador de menor carga. A diferença é que esse procedimento somente é realizado quando o benefício final é melhor do que o custo da migração, e caso contrário, o objeto permanece no processador no qual já estava antes. Além disso, é possível escolher um valor percentual que especifica a quantidade de migrações que pode ser tolerada (LABORATORY, 2017a). A Tabela 1 informa alguns dos algoritmos de LB disponíveis no Charm++ e seus diferentes aspectos, os quais são responsáveis por tornar cada abordagem específica para diferentes tipos de aplicações, sejam elas sobrecarregadas em computação ou comunicação.

Além de fornecer essas estratégias de balanceamento, o ambiente de programação possui a característica de se adaptar eficientemente a ambientes dinâmicos por conta do Charm++ RTS. Este permite realizar o balanceamento de carga de forma automática e provê interações orientadas a mensagens (NIKHIL et al., 2015). Portanto, o Charm++ RTS reúne informações sobre a carga das tarefas, carga dos processadores e sobre o padrão de comunicação para serem usadas pelo *framework* de balanceamento de carga. Esses dados coletados auxiliam no momento de migrar os *chares* entre PEs para balancear a carga computacional

Tabela 1 – Comparação dos algoritmos de LB do Charm++

Algoritmo	**Estratégia**	**Heurística**	**Considera comunicação**	**Considera computação**
GreedyLB	Centralizada	Gulosa	Não	Sim
RefineLB	Centralizada	Iterativa	Não	Sim
GreedyCommLB	Centralizada	Gulosa	Sim	Sim
GreedyRefineLB	Centralizada	Gulosa	Não	Sim
HierarchicalLB	Hierárquica	Híbrida: diferentes algoritmos em cada nível	Não	Sim
MetisLB	Centralizada	Algoritmos recursivos com particionamento de grafos	Sim	Sim
ScotchLB	Centralizada	Particionamento de grafos	Sim	Sim
GrapevineLB	Distribuída	Propagação de informação (protocolo *gossip*) e transferência de carga	Sim	Sim

Fonte: produzido pela autora.

e diminuir o sobrecusto de comunicação durante a execução da aplicação (MENON, 2012).

2.3 Aprendizagem de Máquina

A técnica de Aprendizagem de Máquina, em inglês *Machine Learning* (ML), aborda o desenvolvimento de algoritmos que permitem ensinar a um computador como realizar predições diante de uma determinada situação de interesse. Com base em dados empíricos e captura de características, forma-se uma coleção de *amostras de treinamento* ou *exemplos* que ilustram as relações entre as variáveis observadas. Assim, é possível reconhecer padrões importantes após aplicar o procedimento de aprendizagem (LI, 2011).

O objetivo da técnica de ML é otimizar um critério de desempenho usando dados de exemplo ou uma experiência do passado, sendo crucial para situações onde o problema muda com o tempo ou que depende de um ambiente específico. A abordagem correta em situações dinâmicas, como mencionado, é suportar um sistema que tenha a capacidade de se *adaptar* às circunstâncias expostas, ao invés de programar manualmente um programa diferente para cada circunstância especial (ALPAYDIN, 2010).

ML pode ser considerada um tipo de Inteligência Artificial (IA) que fornece ao computador a habilidade de aprender sem que este seja explicitamente programado (HAMID;

SUGUMARAN; JOURNAUX, 2016). Logo, busca-se a construção de sistemas que dependam o mínimo possível de uma intervenção do programador, para que dessa forma seja possível aprender através dos dados e realizar previsões com o máximo de exatidão. Diferente de outros modelos estatísticos que valorizam a inferência, o foco de ML é a precisão da previsão (HALL et al., 2014).

Há diversas técnicas de ML desenvolvidas para diferentes tipos de aplicação, mas não existe uma técnica universal que funcione igualmente bem para todos os cenários e conjuntos de dados. Atualmente, algoritmos de ML estão presentes em programas com propósitos distintos, como Processamento de Linguagem Natural, *Data Mining*, Diagnóstico Médico, Ferramentas de Busca e Detecção de Padrões (YOU; LI, 2013; HULTH, 2004). Portanto, para tratar essa diversidade de contextos de aplicação, faz-se uso de técnicas que diferem entre si no aspecto de fazer uso ou não de dados rotulados (HAMID; SUGUMARAN; JOURNAUX, 2016). Neste trabalho, serão introduzidas duas grandes abordagens: *Aprendizagem Supervisionada* e *Aprendizagem Não Supervisionada*, as quais serão tratadas nas seções seguintes.

2.3.1 Aprendizagem de Máquina Supervisionada

Um dos tipos de técnicas de ML existentes é a *Aprendizagem Supervisionada*, a qual infere uma função usando um dado supervisionado e realiza uma classificação ou trabalhos de regressão sobre o mesmo (LI, 2011). Assim, é possível aprender um mapeamento de uma entrada para uma saída, da qual os valores corretos e confiáveis são fornecidos por um supervisor (ALPAYDIN, 2010).

É considerado um método *supervisionado* por se basear no treinamento de uma amostra de exemplos provenientes de uma fonte de dados, onde já existe uma classificação correta associada a eles (SATHYA; ABRAHAM, 2013). Em outras palavras, conjuntos previamente rotulados treinam novas amostras de dados para que então uma função seja determinada com o propósito de mapear novas instâncias para as classes já definidas (HAMID; SUGUMARAN; JOURNAUX, 2016).

O objetivo de ML Supervisionada é construir um modelo para a distribuição de rótulos de classe que possui como base o uso de características preditoras. O classificador resultante é usado para associar os rótulos de classe às instâncias que estão sendo testadas, onde os valores de suas características preditoras são conhecidas, mas o valor do rótulo de classe não é (HAMID; SUGUMARAN; BALASARASWATHI, 2016).

ML Supervisionada envolve áreas como Predição, Extração de Conhecimento e Tarefas de Compressão, fazendo uso de uma abordagem estatística de estimativa de densidade (HAMID; SUGUMARAN; JOURNAUX, 2016). Alguns dos principais algoritmos utilizados são o de *Regressão*, *Árvores de Decisão*, *Redes Neurais* e *Naive Bayes* (HALL et al., 2014).

Já no estágio de testes, é necessário garantir que as novas predições realizadas te-

nham resultados consistentes. Assim, o algoritmo deve prever a classe para instâncias ainda não vistas até o momento de acordo com a função definida anteriormente (HAMID; SUGUMARAN; JOURNAUX, 2016).

2.3.2 Aprendizagem de Máquina Não Supervisionada

No contexto da *Aprendizagem Não Supervisionada* não há um supervisor e nem saída de dados, apenas dados de entrada. Essa técnica possui como objetivo encontrar as regularidades que pertencem à entrada recebida (ALPAYDIN, 2010). Assim, sua principal diferença com ML Supervisionada é que não há um treinamento próprio que faz uso de exemplos rotulados.

A técnica envolve o agrupamento de dados em múltiplos grupos (*clustering*) para que seja possível maximizar a similaridade dos itens que pertencem a um mesmo grupo e minimizar a similaridade entre os grupos. Ou seja, possui como princípio encontrar em um dado recebido seu padrão escondido, agrupando dados com tipos parecidos (HAMID; SUGUMARAN; JOURNAUX, 2016). Logo, o foco não é prever um rótulo para um item de dado, mas *descobrir grupos escondidos* ou *padrões* nos dados de entrada. Esses grupos não possuem significado por si só, e portanto, uma análise deve ser realizada separadamente com o propósito de derivar um significado de cada grupos (HAMID; SUGUMARAN; BALASARASWATHI, 2016).

Assim, algoritmos de ML não supervisionada segmentam dados em grupos de exemplos (*clusters*) ou grupos de características. A combinação de características em grupos menores e mais representativos é chamada de *extração de característica*, e encontrar o subconjunto mais importante das características de entrada é chamado de *seleção de característica* (HALL et al., 2014).

ML Não Supervisionada trata de áreas como Reconhecimento de Padrões e Detecção de Anomalias (HAMID; SUGUMARAN; JOURNAUX, 2016). Para tais finalidades, utiliza-se algoritmos como *Clustering* e *Fatorização de Matriz Não Negativa* (HALL et al., 2014).

3 Proposta de um Meta-Escalonador Baseado em Aprendizado de Máquina

3.1 Visão Geral

Como visto anteriormente, o balanceamento de carga é um dos principais elementos que afetam o desempenho e a escalabilidade de certas aplicações. Conforme exposto na Seção 2.2, uma aplicação HPC com comportamento dinâmico tem a tendência de sofrer alterações no comportamento de sua carga, mesmo após um balanceamento. Características observadas em instante específico no tempo, como a taxa de comunicação, taxa de computação e desbalanceamento de carga interferem no comportamento da carga naquele momento. Assim, uma aplicação que inicialmente possuía sua carga desbalanceada devido à alta taxa de comunicação, pode apresentar outro tipo de desbalanceamento futuramente, como ter sua carga desbalanceada com relação à computação. Indo mais além, características como o valor médio da carga de cada objeto em determinado instante pode impactar na escolha do melhor algoritmo de LB para aquela aplicação.

Portanto, a desvantagem de escolher um balanceador estaticamente, mesmo que este seja o melhor para uma determinada aplicação em um momento inicial, é que este balanceador pode não ser o melhor durante todos os instantes em que o balanceamento é realizado. Em outras palavras, é muito provável que, no contexto de uma aplicação dinâmica, seja necessário escolher um algoritmo de LB diferente para cada instante do balanceamento, de acordo com as características que a aplicação apresenta naquele momento.

Assim sendo, o foco deste trabalho é desenvolver um meta-escalonador adaptativo que seleciona o melhor algoritmo de LB levando em conta as diferentes características de cada aplicação sendo analisada. O propósito é evitar que decisões de balanceamento de carga sejam tomadas manualmente para cada tipo diferente de aplicação científica de HPC. Logo, o meta-escalonador terá a capacidade de escolher diferentes algoritmos de Balanceamento de Carga em diferentes instantes de uma mesma execução de uma aplicação, de acordo com as características atuais observadas naquele momento. O meta-escalonador leva em consideração a arquitetura em que a aplicação está sendo executada, como número de PEs e outras características que a tornam única.

Uma vez que diferentes aplicações necessitam de diferentes estratégias de balanceamento, será usado o *framework* de LB do Charm++, o qual fornece múltiplos balanceadores de carga apropriados para cenários distintos. Além disso, o *benchmark* sintético *LBTest* (MENESES; KALE; BRONEVETSKY, 2011) disponível no Charm++ será utilizado para simular aplicações com diferentes parâmetros que as descrevem, como topologia de rede, número

de elementos e tempo mínimo e máximo de duração de cada tarefa.

As estratégias de balanceamento escolhidas para serem usadas neste trabalho, e que foram apresentadas na Seção 2.2, são: *GreedyLB*, *GreedyCommLB*, *RefineLB* e *GreedyRefineLB*. A decisão de escolha do GreedyLB, GreedyRefineLB e RefineLB é baseada no fato de serem as estratégias *centralizadas* mais utilizadas no *Charm++*, oferecendo estabilidade durante seu uso, ausência de erros durante a execução e fornecendo representatividade no conjunto de algoritmos. Assim, tem-se uma estratégia gulosa, uma estratégia baseada em refinamento e uma combinação de ambas, tornando o modelo abrangente. Já o GreedyCommLB foi selecionado para representar um algoritmo de LB que possui foco na comunicação, e assim, tornando possível a sua utilização nos casos em que a aplicação em execução tenha a carga de comunicação mais significativa do que sua carga de computação, dado que o mesmo faz uso do grafo de comunicação para as tomadas de decisão.

A Figura 10 mostra uma visão geral da proposta deste trabalho. O principal objetivo da fase estática é obter o meta-escalonador que escolherá o melhor algoritmo de balanceamento de carga para diferentes tipos de aplicações. Assim, é necessário gerar diversos tipos de cenários testes através do *LBTest* para obter as amostras que representam aplicações e suas características que as tornam particulares, como por exemplo, diferentes cargas de computação ou comunicação. Por sua vez, a fase dinâmica tem como meta observar a execução de uma aplicação não vista durante a fase de testes e verificar se a invocação do melhor balanceador de carga é feita, o qual é previsto pelo meta-escalonador desenvolvido. As seções a seguir apresentam os detalhes de cada uma das etapas mostradas na Figura 10.

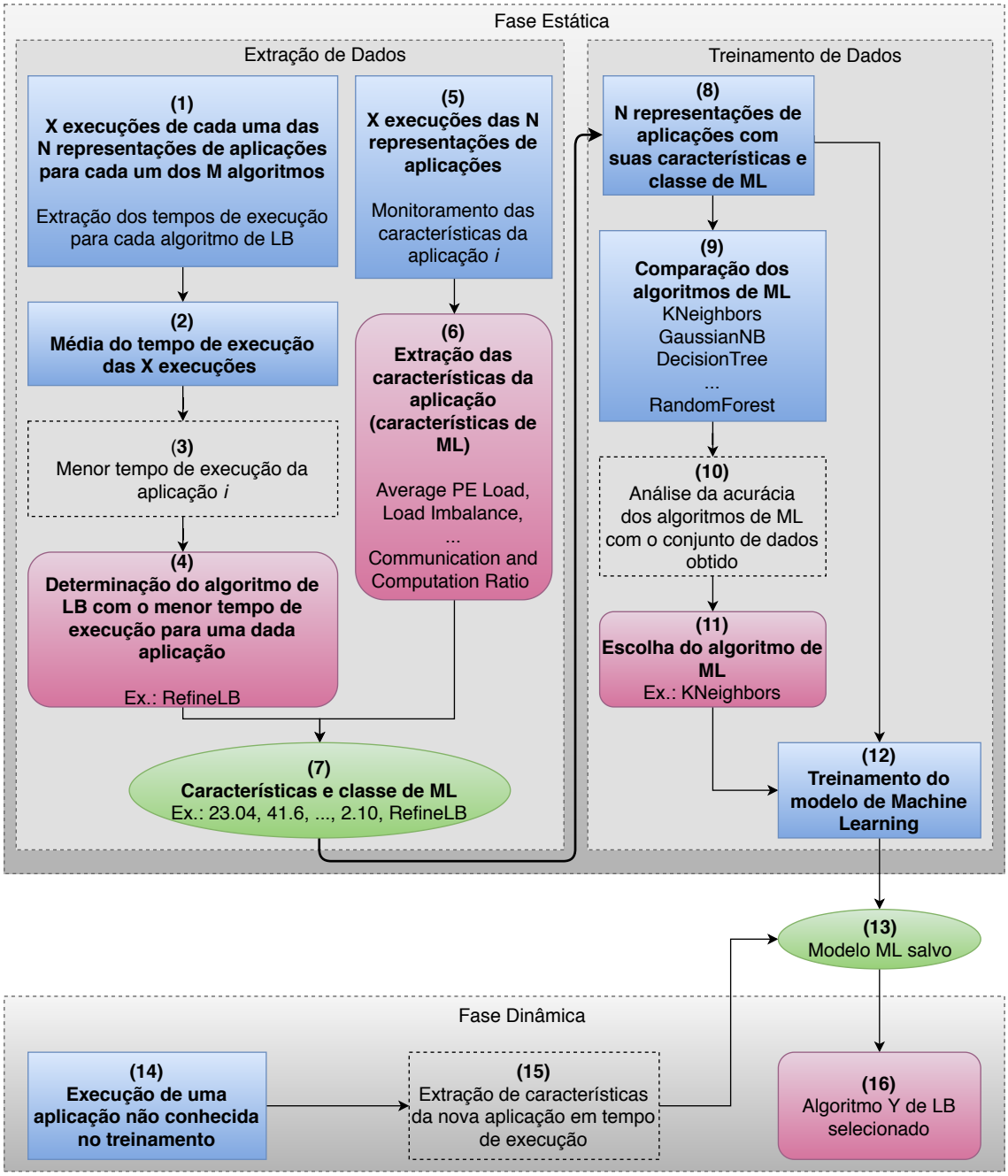
3.2 Extração de Dados

O estágio de extração de dados é a primeira etapa que ocorre durante a fase estática do projeto, tendo como produto final as características de diversas aplicações e o melhor algoritmo de balanceamento de carga para cada uma delas. Em outros termos, o objetivo é obter todas as características e classes de diferentes aplicações para usar no futuro modelo de Aprendizado de Máquina.

O Charm++ RTS fornece informações sobre a carga da aplicação através de um gerenciador de LB que reside em cada processador, monitorando o comportamento da carga daquela CPU específica. Seu papel é coletar dados da carga dos objetos, carga do *background* e estatísticas sobre tempo de ociosidade, e por fim armazena todas essas informações no banco de dados de LB (MENON, 2012). O foco deste trabalho é nas estratégias *centralizadas*, onde há um processador dedicado que coleta informações globais sobre o estado de toda a máquina.

Sabe-se que objetos podem migrar de um processador a outro em tempo de execução, podendo aumentar o desempenho do programa paralelo ao realizar a migração de um

Figura 10 – Funcionamento detalhado do projeto.



Fonte: produzido pela autora.

PE sobrecarregado para outro com menos carga. O *framework* de LB do Charm++ permite coletar a estrutura da comunicação e carga de computação durante a execução, armazenando-os em um banco de dados de balanceamento de carga. Esse banco de dados guarda informações de LB em cada processador, coordenando o gerenciador de LB e o balanceador. Assim, essa estrutura permite que as estratégias de LB usem a informação do banco de dados para a tomada de decisão com relação ao novo mapeamento de objetos a ser realizado (LABORATORY, 2017a).

A geração de diversas aplicações com características particulares é o primeiro passo a ser tomado, conforme indicado na Figura 10. A grande vantagem de usar o *LBTest* é que ele permite a alteração de seus parâmetros para gerar *n* aplicações sintéticas com diferentes tipos de distribuição de trabalho e comunicação, representando o conjunto de treinamento para o algoritmo de aprendizagem de máquina. Além disso, o *benchmark* também será usado para gerar o conjunto de testes que será usado na etapa de validação da proposta. Assim, é possível obter alta representatividade de aplicações com diferentes características e comportamentos, verificando quais elementos são relevantes para a análise da escolha do que deve ser balanceado. O *LBTest* espera como parâmetros:

- Número de elementos da aplicação;
- Número de passos ou iterações que a aplicação deve realizar;
- Frequência de *print*;
- Frequência de LB;
- Duração mínima de cada tarefa - microsegundos;
- Duração máxima de cada tarefa - microsegundos;
- Topologia de rede - *ring, mesh2d, mesh3d*;
- Algoritmo de LB- opcional;
- *Flag* de *debug* - opcional.

Diversos *scripts* foram desenvolvidos em *bash* e em *Python* para gerar essas aplicações sintéticas de forma automatizada. Portanto, o funcionamento a nível de implementação da caixa 1 na Figura 10 é feito por um *script* que realiza X execuções de uma mesma aplicação i para cada um dos M algoritmos de LB. Os resultados dessas execuções são armazenados em um arquivo para análise futura e a média dos X tempos de execução observados é salva. É importante ressaltar que é necessário executar múltiplas vezes uma mesma aplicação com um dado algoritmo de LB para evitar *outliers*, ou seja, um número muito diferente que pode ser obtido por variações no *hardware* ou por outros fatores externos.

Foram realizados testes para analisar quantas execuções de cada cenário seriam necessárias, observando a alteração dos valores gerados de acordo com o aumento do número de execuções. Sabe-se que quanto mais execuções são feitas, mais lento o processo se torna, e uma vez que há uma grande quantidade de aplicações que passarão por esse procedimento, é inviável testar muitas variações desses valores neste trabalho. Dado que o tempo é limitado, um estudo mais aprofundado da quantidade de execuções deve ser realizado em trabalhos futuros e, portanto, a quantidade de *cinco execuções* se mostrou com resultados estáveis e com um tempo viável para realizar esse processo.

3.2.1 Seleção do Melhor LB para uma Determinada Aplicação

O fluxo do projeto tem o seu início na representação de diversas aplicações com características diferentes. Este passo é essencial para que o meta-escalonador tenha sucesso nas escolhas futuras, pois em uma situação real, a decisão do melhor algoritmo de LB só será bem sucedida se durante o treinamento houver uma instância com características similares às características da aplicação sendo analisada. Sendo assim, deve-se gerar instâncias com grande variedade de parâmetros para ilustrar cada cenário diferente de aplicações reais que podem vir a ocorrer.

Além disso, é importante ressaltar que uma aplicação que apresenta um conjunto de informações sobre si mesma no instante *X* da execução pode ter informações diferentes no instante *Y* da mesma execução. No contexto do Charm++, isso significa que os valores que informam o estado atual dos processadores, objetos e comunicação armazenadas no banco de dados de LB no instante *X*, podem ser diferentes ao consultar o banco de dados em um instante futuro *Y*. Portanto, é necessário obter a representação de diferentes momentos da aplicação sendo observada.

Conforme dito anteriormente, a abordagem escolhida para solucionar essa questão é a execução de cada aplicação sintética cinco vezes para cada um dos algoritmos de balanceamento de carga. Assim, é possível calcular a média dos tempos de execução para um determinado cenário, e com essas informações obtidas, o processo de determinar o algoritmo mais rápido para um contexto em particular se torna simples e os dados se tornam mais estáveis e confiáveis.

Um exemplo prático pode ser visto na Tabela 2. Os resultados obtidos representam o cenário de uma aplicação que possui as seguintes características: 8500 elementos, mesh2d como topologia de rede e tarefas cuja duração varia de 1000 a 2000 ms. Sua execução ocorre durante 601 passos e o balanceamento de carga é realizado a cada 580 iterações, ou seja, neste caso espera-se a iteração 580 para observar as características naquele instante e invocar o algoritmo de LB. Na prática, os comandos executados para este exemplo são vistos na Listagem 3.1.

Ao analisar a Tabela 2, nota-se que ao comparar os valores da média do tempo de execução obtido para cada algoritmo de LB, há um algoritmo mais rápido e portanto, específico para a aplicação sintética que foi executada. Neste caso, pode-se dizer que o *GreedyCommLB* é o melhor algoritmo para realizar o balanceamento de carga no cenário de uma aplicação com características similares à que estava sendo executada. Futuramente, no momento em que o meta-escalonador extrair informações de uma aplicação real em tempo de execução e essas informações forem similares às informações do caso sendo analisado nesse exemplo, ele saberá determinar que o algoritmo a ser escolhido naquele instante para aquelas características será o *GreedyCommLB*.

```
./charmrun +p20 ./lb_test 8500 601 1 580 1000 2000 mesh2d +balancer
   GreedyLB +LBDebug 1 +pemap 0-19

./charmrun +p20 ./lb_test 8500 601 1 580 1000 2000 mesh2d +balancer
   GreedyCommLB +LBDebug 1 +pemap 0-19

./charmrun +p20 ./lb_test 8500 601 1 580 1000 2000 mesh2d +balancer
   RefineLB +LBDebug 1 +pemap 0-19

./charmrun +p20 ./lb_test 8500 601 1 580 1000 2000 mesh2d +balancer
   GreedyRefineLB +LBDebug 1 +pemap 0-19
```

Listagem 3.1 – Exemplo de comandos para geração de aplicações sintéticas no *LBTest*.

Tabela 2 – Comparação dos tempos de execução dos algoritmos de LB.

Execução	Tempo de execução dos algoritmos (segundos)			
	GreedyLB	GreedyCommLB	RefineLB	GreedyRefineLB
1	108.435121	108.499599	108.470317	115.424120
2	108.487886	94.124221	108.463129	115.476028
3	108.504923	108.353334	108.507224	97.631944
4	108.619229	108.469404	100.027127	115.576170
5	108.529570	101.767865	108.448754	115.561473
Média:	108.5153458	104.2428846	106.7833102	111.933947

Fonte: produzido pela autora.

Por fim, após obter diversas representações de aplicações sintéticas, executá-las X vezes para cada um dos algoritmos de balanceamento de carga selecionados, calcular as médias dos tempos de execução para cada algoritmo e determinar qual deles foi o mais rápido para cada aplicação gerada, pode-se prosseguir ao próximo passo do projeto. Até este momento, as caixas *1*, *2*, *3* e *4* da Figura 10 foram discutidas.

3.2.2 Extração das Características para o Modelo de Aprendizado de Máquina

Neste ponto, tem-se as instâncias de aplicações geradas pelo *benchmark*, seus tempos de execução e o melhor algoritmo de LB para cada uma delas. O passo seguinte é a realização da *extração de características* que serão usadas futuramente no modelo de ML, como mostra a caixa *5* da Figura 10. Esta etapa observa as propriedades pertinentes presentes nas instâncias, definindo quais dados são importantes para caracterizar uma aplicação e tornando possível utilizar as informações obtidas na tomada de decisão do algoritmo de LB.

Assim, ocorrerá a extração das características de cada instância criada no passo anterior. Foi desenvolvido um *Perfilador*, cuja responsabilidade é observar uma aplicação em tempo de execução e coletar as informações relevantes que a representam, repassando es-

ses dados para o meta-escalonador que, por sua vez, alimenta o modelo treinado de ML para obter a resposta da predição. Similarmente ao processo anterior, os *scripts* desenvolvidos realizam esse procedimento *X* vezes para que se possa obter a média de cada uma das características, evitando valores extrapolados por causas externas.

A base para o sucesso do modelo a ser gerado é a escolha correta das características que serão extraídas. Para isso, é necessário realizar um estudo com um grande número de características potenciais que poderiam ser usadas. Durante esses experimentos, foi possível descartar diversas características candidatas por razões como significado ambíguo a de outras, baixo impacto no resultado, sobrecusto muito grande para cálculo do valor a ser usado, entre outros motivos. Por exemplo, informações como número total de mensagens, número de mensagens enviadas dentro de um mesmo *PE*, número de mensagens enviadas para uma PE externa são similares, e portanto, algumas foram descartadas para evitar o sobre-ajuste do modelo de ML. Neste caso, foi concluído que usar a proporção da taxa de comunicação e computação combinada com a porcentagem da quantidade média de mensagens enviadas para PEs externas, acaba por ser suficiente para representar essa informação de comunicação, sem usar características em excesso.

Um outro exemplo de característica candidata que foi descartada é o número de núcleos da arquitetura onde a aplicação será executada. A razão para desconsiderá-la é que o número de núcleos será sempre o mesmo para uma dada arquitetura. Em outras palavras, se essa informação fosse usada, isso resultaria em um conjunto de dados cuja característica *número de núcleos* possui sempre o mesmo valor para todas as instâncias possíveis. Neste caso, o número de núcleos será informado apenas uma vez e de forma manual, no instante em que o *LBTest* é invocado com seus devidos parâmetros.

Essa análise de características candidatas é de extrema importância para o sucesso do meta-escalonador sendo desenvolvido, e portanto demanda tempo para analisar as diversas combinações e importância das informações coletadas. Afinal, pode ser o caso de uma característica potencial ser útil, mas o sobrecusto para calculá-la é grande demais a ponto de tornar o meta-escalonador mais lento do que deveria e degradar o desempenho da aplicação ao fazer o balanceamento de carga. Dessa forma, sete características foram escolhidas para representar uma aplicação, como mostra a Tabela 3:

Uma vez definidas as características que serão coletadas, o Perfilador desenvolvido é responsável por extraí-las de cada aplicação em execução. Esse processo será realizado para cada uma das instâncias geradas na etapa anterior, para que dessa forma, não só seja possível saber o melhor algoritmo para uma aplicação *X*, mas também quais são as características internas que a definem.

É importante ressaltar a razão de não ter realizado esse processo juntamente com a geração das aplicações na etapa anterior. É intuitivo pensar que, se já estão sendo criadas representações de aplicações para observar o melhor algoritmo de LB para elas, poderia ser

Tabela 3 – Características selecionadas.

característica	**Descrição**
AvgPELoad	Média da carga dos processadores
AvgOverloadedPEs	Média de processadores sobrecarregados em porcentagem
LoadImbalance	O quanto a aplicação está desbalanceada (MaxPeLoad/AvgPeLoad)
AvgIdleTime	Tempo médio que os processadores estão ociosos
AvgObjLoad	Carga média de todos os objetos
CommCompRatio	Taxa de comunicação/computação
MsgOutsidePePercent	Porcentagem da média de mensagens enviadas para outros processadores

Fonte: produzido pela autora.

possível observar as características dessas aplicações concorrentemente. O motivo de realizar essas etapas separadamente é que o tempo de coleta das características pode afetar no tempo de execução do balanceamento de carga, que é o principal ponto observado na etapa anterior. Consequentemente, pode-se acabar por determinar que um algoritmo de LB Z é o melhor para a aplicação i erroneamente, pois um algoritmo de LB Y é na verdade o melhor mas teve seu tempo de execução alterado por conta do Perfilador. Assim, as caixas *1*, *2*, *3* e *4* da Figura 10 devem ser executadas separadamente para que não haja interferência alguma no tempo de execução dos algoritmos de LB para cada aplicação sendo analisada.

Citando como exemplo a aplicação da subseção anterior, sabe-se que já foi criada não só a sua representação (8500 elementos, mesh2d, duração de 1000 a 2000), mas também foi definido o algoritmo de balanceamento de carga mais rápido para ela. Agora é necessário observar a execução desta mesma aplicação para coletar as características previamente definidas, e dessa forma, as duas informações poderão ser unificadas e transformadas em dados com significado relevante para o modelo de ML. Na prática, a Listagem 3.2 mostra o comando que é executado três vezes e então é calculado a média de cada característica obtida:

```
./charmrun +p20 ./lb_test 8500 601 1 580 1000 2000 mesh2d +balancer
   Profiler +LBDebug 1 +pemap 0-19
```

Listagem 3.2 – Exemplo de obtenção de características.

Nota-se que os parâmetros da aplicação são exatamente os mesmos do exemplo da etapa anterior cujo objetivo era encontrar o melhor algoritmo para essa instância. Agora que já se sabe o melhor algoritmo de LB para ela, o comando acima é responsável por executar a aplicação com o Perfilador, observar e coletar suas características. Como resultado, as características que definem essa aplicação em particular se tornarão conhecidas e serão

o conjunto de informações usado para representar essa instância. A Tabela 4 mostra o caso real das três execuções do Perfilador e a média de cada uma das características extraídas.

Tabela 4 – Coleta de características

Execução	Avg Pe Load	Avg Overloaded PEs (%)	Load Imbalance	Avg Idle Time	Avg Obje Load	Comm Comp Ratio	Msg Outside PE (%)
1	101.5291	40.00	1.0298	3.0313	0.2385	0.2026	5.8824
2	101.4635	40.00	1.0290	2.9503	0.2383	0.2027	5.8824
3	101.5005	40.00	1.0288	2.9296	0.2384	0.2026	5.8824
Média:	101.4977	40.00	1.0292	2.9704	0.2384	0.2026	5.8824

Fonte: produzido pela autora.

Uma vez que as características foram coletadas, podemos unificar as informações da Tabela 2 e 4 como ocorre na caixa *7* da Figura 10. Isso é realizado por um *script* de processamento de dados, cujo objetivo final é gerar um arquivo *.csv* onde as sete primeiras colunas representam cada uma das características e a coluna final representa a classe de ML, ou seja, aquilo que se quer prever: o algoritmo de LB. Na prática, tem-se até o momento:

```
Aplicação Exemplo: 8500 elementos, mesh2d, 1000 a 2000 duração das
   tarefas, 601 iterações, 580 de frequência de LB

Arquivo .csv: 101.4977, 40.00, 1.0292, 2.9704, 0.2384, 0.2026, 6.8824,
   GreedyCommLB
```

Listagem 3.3 – Exemplo de características e classe de uma aplicação.

Portanto, é visível que os dados estão prontos para a etapa de treinamento de dados. Sabe-se que para cada aplicação *i* criada, há um algoritmo de LB que teve um melhor desempenho para ela. Além disso, sabe-se que para cada uma dessas aplicações, é possível representá-las através de um conjunto de características que foram observadas e extraídas durante suas respectivas execuções. Isso é útil futuramente, pois para uma aplicação real sendo executada, seu conjunto de características coletadas pode ser similar a um dos conjuntos de características que foi extraído na fase de treinamento. Logo, o meta-escalonador poderá tomar a decisão do melhor algoritmo de LB para aquela aplicação real não conhecida na etapa de treinamento.

3.3 Treinamento do Modelo de ML

As etapas de extração de dados realizadas anteriormente tiveram como principal objetivo gerar *N* representações de aplicações com suas características e respectivas classes de

ML (algoritmos de LB) definidas. Assim, dando continuidade ao fluxo do projeto e prosseguindo para a caixa *9* da Figura 10, pode-se dar início ao processo de escolha de um algoritmo de ML e treinamento do modelo.

Foi utilizado a biblioteca *scikit-learn* disponível para a linguagem de programação Python. O principal motivo dessa escolha é o suporte existente para diversos algoritmos de aprendizagem de máquina supervisionada e não supervisionada. Assim, a biblioteca explora problemas de classificação, regressão e *clustering*, fornecendo algoritmos como *SVM*, *Random Forests*, *k-means*, entre outros (PEDREGOSA et al., 2011). Adicionalmente, Python tem se tornando uma das linguagens mais populares no ramo da computação científica por conta da sua natureza interativa, facilidade na visualização e análise de dados e suporte a bibliotecas científicas, como *NumPy*, *SciPy* e *Cython* (WALT; CHRIS; VAROQUAUX, 2011).

Além da facilidade de uso, uma das vantagens da biblioteca *scikit-learn* é que a mesma tem como objetivo maximizar a eficiência computacional. A Tabela 5 mostra a comparação do tempo computacional entre alguns algoritmos implementados nas bibliotecas de ML mais utilizadas em Python. Alguns dos seus algoritmos são escritos em *Cython* para atingir um desempenho melhor, o que é prioridade no contexto deste trabalho. Por fim, a biblioteca também é eficiente para realizar a comparação de diferentes tipos de algoritmos por meio de métricas e gráficos, facilitando a análise de dados (PEDREGOSA et al., 2011).

Tabela 5 – Tempo em segundos de diversas bibliotecas de ML disponíveis em Python.

	scikit-learn	mlpy	pybrain	pymvpa	mdp	shogun
Support Vector Classification	**5.2**	9.47	17.5	11.52	40.48	5.63
Lasso (LARS)	**1.17**	105.3	-	37.35	-	-
Elastic Net	**0.52**	73.7	-	1.44	-	-
k-Nearest Neighbors	0.57	1.41	-	**0.56**	0.58	1.36
PCA (9 components)	**0.18**	-	-	8.93	0.47	0.33
k-Means (9 clusters)	1.34	0.79	*	-	35.75	**0.68**
License	BSD	GPL	BSD	BSD	BSD	GPL

-: Não implementado
*: Não converge em 1 hora

Fonte: (PEDREGOSA et al., 2011).

Assim, foi desenvolvido um *script* com o objetivo de realizar experimentos entre diferentes algoritmos de ML para identificar o melhor no contexto dos dados coletados. A entrada do *script* é o arquivo *.csv* previamente gerado, possuindo o conjunto de características e a classe de cada uma das *n* aplicações geradas. A partir deste momento, é necessário efetuar um *pré-processamento* dos dados, tendo como objetivo aumentar a qualidade do conteúdo obtido na etapa anterior e garantir uma maior confiabilidade ao fornecer as amostras para o algoritmo de ML. Há características que podem assumir valores extremamente variá-

```
# normalizing features
label_encoder = preprocessing.LabelEncoder()
label_encoder.fit(classes)
classes = label_encoder.transform(classes)
features = preprocessing.normalize(features)
```

Listagem 3.4 – Pré processamento de dados: normalização da características e transformação das classes.

veis, e portanto, esses números passam pelo processo de *normalização* e as classes (algoritmos de LB) são transformadas de texto para valores numéricos. A Listagem 3.4 mostra esse processo.

Foi gerado um total de 314 instâncias de aplicações diferentes. Esse processo levou semanas para ser concluído, levando em consideração que os *scripts* foram executados em uma máquina remota por 12 horas diárias ininterruptas. Uma das razões para tal demora é consequência da alta variedade de parâmetros escolhidos para representar diversas aplicações, podendo ter uma alta carga de trabalho para muitos objetos em alguns cenários, e até um número de iterações grande associado a esses parâmetros. Adicionalmente, cada aplicação deve ser executada cinco vezes para cada um dos quatro algoritmos de LB para obter o tempo de execução, e posteriormente executadas três vezes para ter suas características extraídas. O cuidado e paciência na fase de geração de dados é de extrema importância para garantir alta representatividade e corretude do futuro modelo de ML.

Esse conjunto de dados é dividido de forma aleatória, sendo que 70% é destinado à etapa de treinamento e cerca de 30% destinado à etapa de validação dos dados treinados. Isso permite treinar o modelo com uma quantidade de dados significativa e validar a acurácia de cada um dos algoritmos de classificação, e portanto, uma decisão sobre a escolha do classificador a ser utilizado poderá ser tomada com base nos dados observados durante esse processo.

3.3.1 Comparação dos Algoritmos de ML

A abordagem a ser utilizada neste trabalho é a Aprendizagem de Máquina Supervisionada, conforme apresentada na Seção 2.3. Ou seja, o modelo será treinado sobre um conjunto de dados predefinidos para que o mesmo possa tomar decisões futuras ao receber novos dados. Mais especificamente, serão investigados alguns *classificadores*, os quais podem ser considerados uma subcategoria da aprendizagem supervisionada. A razão para tal escolha se dá pelo encaixe perfeito dos tipos de dados obtidos, ou seja, toma-se uma entrada e um rótulo é atribuído a ela. No caso desse projeto, as entradas são as características (únicas de uma aplicação) e o rótulo é o melhor algoritmo de LB para ela.

Assim, dado que já existe uma classificação correta associada aos dados previamente

rotulados, é possível construir um modelo no qual o classificador resultante é utilizado para associar os rótulos às novas instâncias sendo testadas. Foram escolhidos sete algoritmos disponíveis na biblioteca *sk-learn* para serem analisados no contexto deste trabalho:

- **K-Nearest Neighbors Classifier (KNN)** - a classificação é computada através da similaridade de um dado com os seus vizinhos mais próximos;
- **Support Vector Machines** - classificador binário não probabilístico: encontra uma *linha de separação* (hiperplano) entre dados de duas classes, buscando maximizar a distância entre os pontos mais próximos em relação a cada uma das classes;
- **Decision Tree Classifier** - árvore de decisão onde cada nodo representa uma característica, cada ramo é uma regra de decisão e cada nodo folha é a classe;
- **Random Forest Classifier** - cria uma floresta de forma aleatória, sendo uma combinação (*ensemble*) de árvores de decisão;
- **Ada Boost Classifier** - constrói um classificador forte a partir da combinação de diversos classificadores de desempenho menor com o objetivo de aumentar a acurácia;
- **Gaussian Naive Bayes** - classificador probabilístico baseado no *Teorema de Bayes*. Calcula a probabilidade de uma instância pertencer a cada uma das classes e classifica com base na classe de maior probabilidade;
- **Linear Discriminant Analysis** - classificador com limite de decisão linear, fazendo uso da regra de *Bayes*, e ajustando uma densidade *Gaussiana* para cada classe;
- **Quadratic Discriminant Analysis** - classificador com limite de decisão quadrático, fazendo uso da regra de *Bayes*, e ajustando uma densidade *Gaussiana* para cada classe.

Há diversas formas visuais de analisar os resultados obtidos, como por exemplo, o relatório de classificação resultante da comparação dos algoritmos. O relatório mostra a precisão, *recall* e o *F1 Score*, tornando-se útil para um melhor entendimento do comportamento do classificador e para a comparação com outros classificadores (YELLOWBRICK, 2016). As métricas são definidas a seguir:

- **Acurácia**: proporção do número de predições corretas sobre o número total de predições. Ou seja, qual a fração de predições que o modelo acertou?
- **Precisão**: habilidade do classificador em não determinar uma classe *X* quando na verdade espera-se uma classe *Y*. Ou seja, para todas as instâncias classificadas como uma classe *X*, qual o percentual de acertos?

- **Recall**: habilidade do classificador de encontrar todas as instâncias corretas. Ou seja, para todas as instâncias de uma classe *X*, qual o percentual de classificações corretas daquela classe?

- **F1 Score**: média harmônica com pesos da precisão e do *recall*, de tal forma que o melhor *score* possível é 1.0 e o pior é 0.0. Seu valor é mais baixo do que medidas de acurácia pois usa o *recall* e precisão na sua computação. Assim, só deve ser usado para comparar classificadores, e não para uso global.

Complementando o relatório de classificação, é feito o uso da matriz de confusão com a finalidade de comparar cada uma das previsões realizadas com a classe real esperada. Cada linha da matriz representa as previsões que resultaram na classe correspondente da mesma linha, e cada coluna representa a classe de fato esperada. A diagonal principal indica quantas previsões realizadas acertaram a classe original, enquanto as demais diagonais mostram os erros de classificação. O objetivo da matriz é entender quais classes são mais facilmente confundidas em comparação com as outras.

Para cada um dos algoritmos de ML escolhidos para análise, houve o treinamento do modelo com diversas alterações de parâmetros para se obter a maior acurácia possível. Por exemplo, no caso de uma árvore decisão, foi visto que sua profundidade máxima tem influência na qualidade das previsões. Da mesma forma, foi necessário realizar diversos experimentos com cada um dos sete classificadores, alterando seus parâmetros como o número de estimadores, número de vizinhos, métricas, entre outros. Isso é importante pois um determinado classificador pode ter seu desempenho totalmente degradado por conta da escolha de parâmetros incorretos.

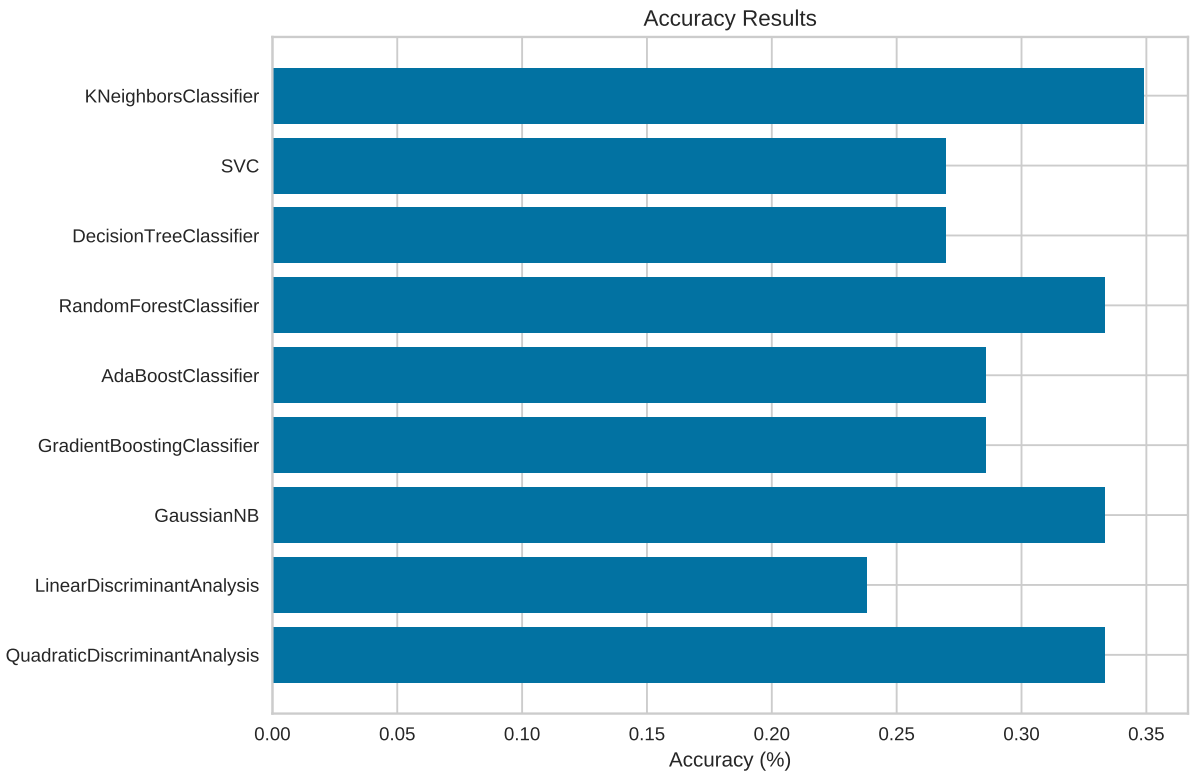
Após a realização desses experimentos, observou-se que os algoritmos com os melhores resultados foram o *KNN*, seguido do *GaussianNB* e *QuadraticDiscriminantAnalysis*. As acurácias obtidas foram de 34.92%, 33.33% e 33.33%, respectivamente. Esses valores correspondem aos acertos do meta-escalonador, ou seja, a porcentagem das vezes em que o algoritmo de LB de melhor desempenho foi de fato escolhido.

Foi necessário um estudo aprofundado para compreender as possíveis razões de obter acurácias de valores abaixo do esperado. Além desse estudo para descobrir a causa, também foram testadas outras alternativas para o aumento dessa métrica, como a mudança de parâmetros e geração de novos dados. A Figura 11 mostra as acurácias obtidas por todos os classificadores analisados.

3.3.2 Análise da Acurácia do Classificador

A resposta encontrada para a situação de que a acurácia não obteve um valor mais elevado é a falta de representatividade de dados no treinamento para um mesmo contexto de

Figura 11 – Comparação dos algoritmos de ML candidatos.



uma aplicação. Ou seja, dada uma aplicação *i* que possui um conjunto de *k* parâmetros, só existe uma única representação dessa aplicação no conjunto de treinamento. Uma vez que não há outra aplicação *j* com características similares a de *i*, no momento da validação do modelo treinado o classificador usará uma instância de teste que também é única e não terá sua representação previamente treinada. Logo, o classificador tentará prever uma aplicação que não foi vista antes no momento do treinamento, uma vez que os dados previamente gerados foram criados para representar *n* aplicações únicas.

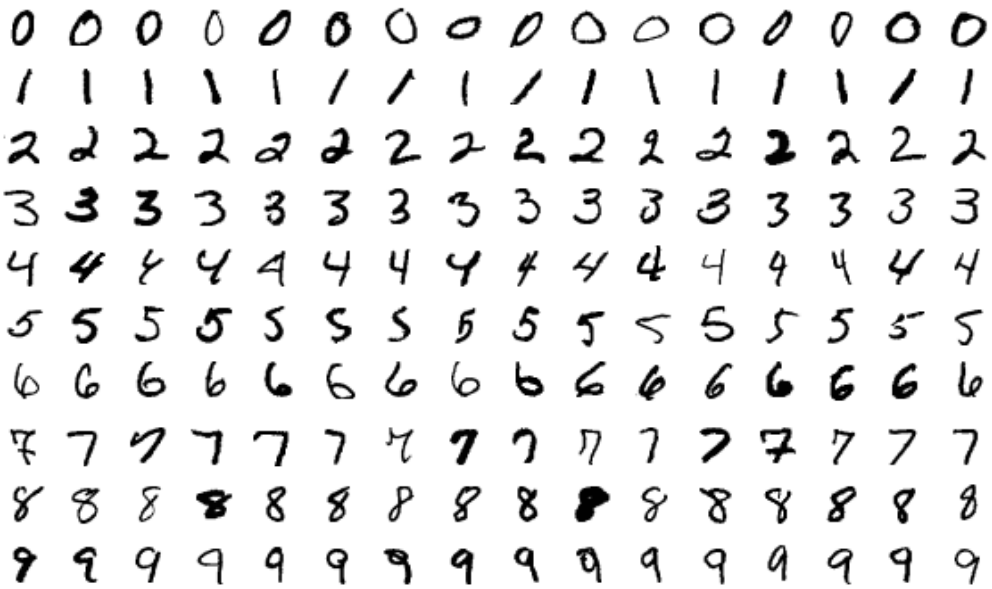
Para solucionar essa situação, seria necessário gerar um número significativo de aplicações com parâmetros similares a cada uma das 314 instâncias previamente geradas. Por exemplo, se é desejado representar uma aplicação *i*, deve-se ter mais cinco outras aplicações com parâmetros semelhantes a de *i* para que a mesma esteja sendo bem representada e que no momento da validação o modelo saiba prever corretamente. Seguindo essa lógica, seria necessário ter 1570 instâncias para melhorar a acurácia da etapa de treinamento.

Um exemplo para ilustrar esse cenário é o problema de classificação de imagens de dígitos, popularmente tratado com o banco de dados *Modified National Institute of Standards and Technology Database (MNIST)* (LECUN; CORTES; BURGES, 2019). O MNIST é um grande banco de dados populado com imagens de dígitos de zero a nove escritos a mão, frequentemente usado para treinar diversos sistemas de processamento de imagens (WIKIPEDIA, 2019).

Se o modelo fosse populado com apenas uma imagem para cada dígito existente, ou seja, um total de dez imagens com os dígitos de 0 até 9, o modelo obteria uma acurácia extremamente baixa. Isso ocorre porque o modelo seria treinado, por exemplo, com imagens dos dígitos 0, 1, 4, 5, 9, mas quando ocorrer a validação para classificar os dígitos 2, 3, 6, 7, 8 não vistos na fase treinamento, o modelo toma uma decisão errada. Essa consequência acontece porque não houve mais representações com imagens diferentes para cada um dos dígitos, ou seja, apesar da estrutura do classificador estar correta, as previsões serão erradas por falta de dados.

A Figura 12 representa um conjunto de amostras dos dígitos do problema apresentado. Note que para cada dígito, há 16 representações do mesmo. Se apenas a primeira *coluna* fosse usada para treinar um modelo qualquer, sendo que da linha 0 a 5 é o conjunto de dados e a linha 6 até 9 é o conjunto de testes, a acurácia seria praticamente nula. É mais simples de visualizar ao analisar a figura, pois fica claro que o número 6, por exemplo, usado para validar, não foi visto no treinamento cujos dígitos usados são de 0 até 4. Entretanto, se o modelo usar *todos* os dígitos da primeira coluna durante o treinamento, e escolher arbitrariamente qualquer dígito das outras colunas, é muito provável que o classificador tome a decisão correta, visto que há uma representação similar daquele dígito já usada no modelo treinado.

Figura 12 – Amostra de imagens do conjunto de dados do MNIST.



Fonte: (WIKIPEDIA, 2019).

Da mesma forma do exemplo acima, apesar do modelo desenvolvido neste projeto ter atingido uma acurácia com um valor não tão alto quanto o desejado durante a fase de validação do treinamento, o classificador ainda pode obter um bom desempenho diante de

situações reais. Em outras palavras, se uma aplicação real tiver características similares a alguma das aplicações geradas para treinar o modelo, o classificador muito provavelmente tomará a decisão correta. Isso é provado no Capítulo 4, onde será discutido os resultados dos experimentos do meta-escalonador desenvolvido diante de aplicações não conhecidas durante o treinamento.

O banco de dados do MNIST é populado com 60,000 exemplos para o conjunto de **treinamento** e 10,000 exemplos para o conjunto de **testes**. É visível que a acurácia e precisão do classificador é extremamente alta, pois há milhares de instâncias que reforçam a representação de um único digito. Assim sendo, o foco deste trabalho não é obter o modelo de ML perfeito, mas sim introduzir o uso de ML para a solução do problema do escalonamento global. Como a escolha das características que caracterizam uma aplicação é um processo não trivial, pois demanda a análise de diversos elementos e uma longa observação do comportamento e dos resultados, fica claro que este problema deve ser investigado para obter uma melhor acurácia no futuro. Dado que há um tempo limitado para este projeto e a geração das 314 instâncias levou semanas, a tentativa de gerar mais dados não é viável. Portanto, é necessário realizar um estudo específico sobre esses pontos aqui ressaltados para que essa solução seja aprimorada e para que os resultados sejam ainda melhores.

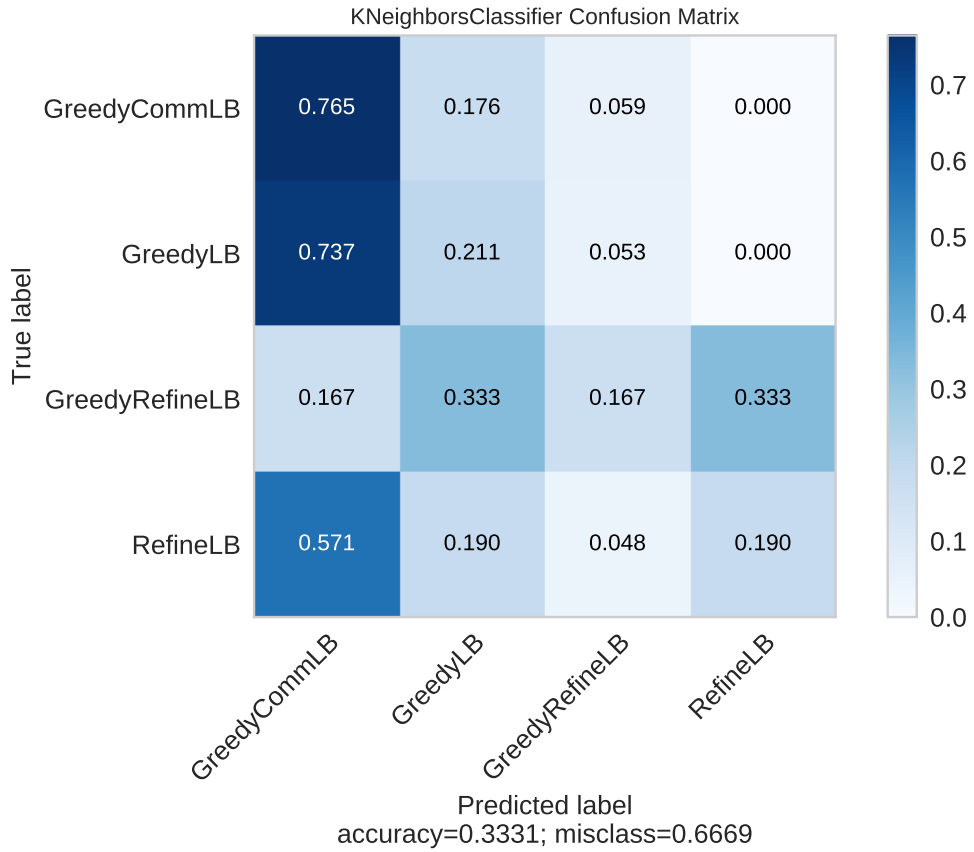
3.3.3 Escolha do Algoritmo de ML

Como explicado anteriormente, os classificadores que resultaram nas melhores acurácias foram o KNN, GaussianNB e QuadraticDiscriminantAnalysis. As Figuras 13 e 14 mostram a matriz de confusão normalizada dos dois primeiros, a fim de comparação. Com base nas matrizes, nota-se que o KNN teve um resultado melhor com uma maior acurácia e menor erro de predição. Seu desempenho para o GreedyLB, GreedyRefineLB e RefineLB foi superior se comparado ao resultado do segundo classificador, e além disso, é compreensível o maior número de predições feitas para o GreedyCommLB, visto que é a classe com mais representações no conjunto de dados.

Usar somente o valor da acurácia para inferir alguma conclusão no modelo sendo analisado pode não ser a abordagem correta. A situação em que há um número desigual de observações para cada classe ou quando há múltiplas classes para realizar a classificação são cenários em que há a necessidade de analisar outros tipos de métricas. A grande vantagem do uso da matriz de confusão, além de resumir o desempenho do algoritmo de classificação, é informação visual do que o modelo acertou e quais erros ele cometeu. Ou seja, é possível não só saber proporção de erros que o classificador cometeu, mas também os tipos de erros ocorridos (BROWNLEE, 2016).

Outro elemento visual utilizado para a análise dos algoritmos foi o relatório de classificação, o qual fornece métricas específicas utilizadas para compreender o modelo gerado. Ele complementa a matriz de confusão, pois apresenta informações úteis que podem ser

Figura 13 – Matriz de confusão do KNN.



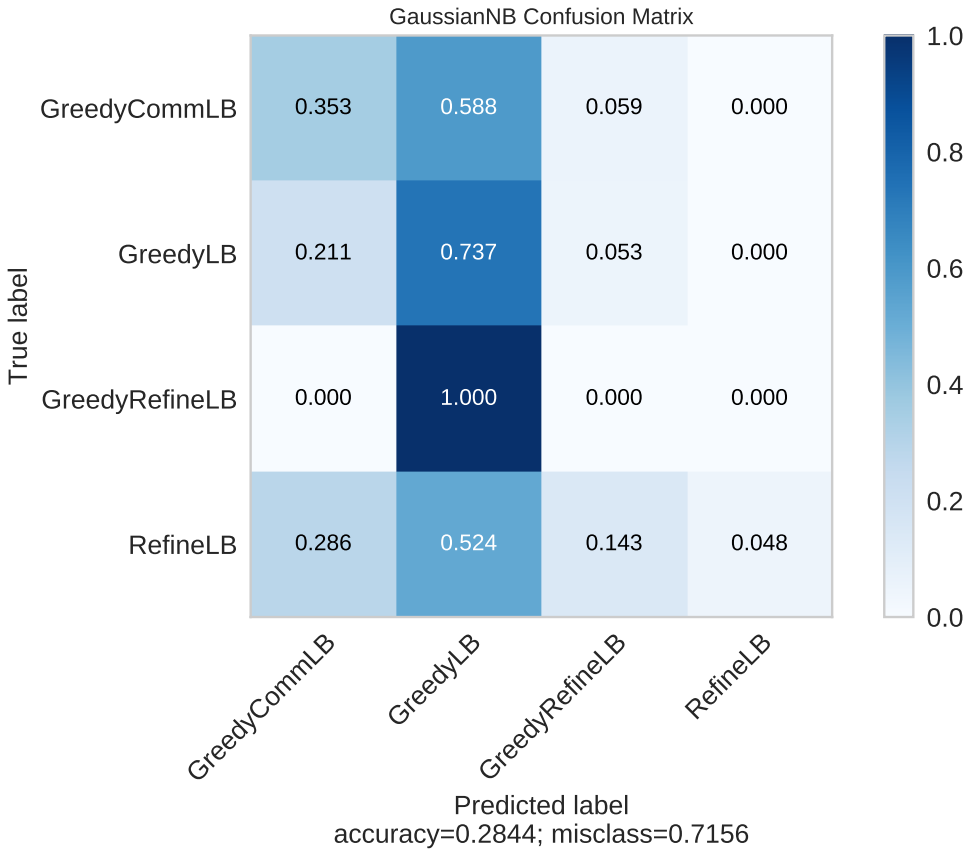
combinadas durante a comparação de classificadores. A Figura 15 representa o relatório de classificação do KNN e a Figura 16 representa o relatório do GaussianNB.

O relatório de classificação comprova o melhor desempenho do KNN e corrobora com a importância de utilizar diferentes ferramentas visuais para a etapa de análise de dados. A acurácia do KNN é levemente maior, mas com as informações disponibilizadas no relatório de classificação pode-se notar uma maior uniformidade nas predições, principalmente para a classe do GreedyRefineLB. Essa classe é a que possui menos instâncias que a representa, ou seja, são 15 instâncias comparadas com 33 instâncias do GreedyCommLB. Mesmo com essa característica, o KNN foi capaz de obter precisão significativa, se comparado ao GaussianNB.

Assim, após uma intensa análise entre os classificadores candidatos, o algoritmo de ML escolhido para este projeto foi o KNN. As métricas usadas para essa decisão foram a acurácia, matriz de confusão e relatório de classificação, uma vez que essas informações combinadas auxiliam em uma escolha mais confiável do modelo a ser utilizado. Como mencionado anteriormente, apesar da acurácia fornecer uma noção geral do desempenho do classificador, ela é melhor usada se o número de observações para cada classe é equivalente.

A Figura 17 ilustra o funcionamento do KNN. O classificador verifica a distância do dado sendo testado com a distância dos seus vizinhos já classificados na etapa de treina-

Figura 14 – Matriz de confusão do GaussianNB.



mento. O grupo de dados (classes) que possui a menor distância entre o ponto de treinamento e o ponto de teste é selecionado. Para explorar todo o potencial algoritmo, é necessário escolher o fator k corretamente, de modo que o modelo seja preciso e que o desempenho do tempo necessário para a predição não cause um impacto negativo.

O fator k é o número de instâncias que serão levadas em consideração para determinar a afinidade das classes. Para a escolha desse número é necessário realizar testes com as instâncias obtidas, uma vez que não há um valor ótimo e genérico que pode ser usado para qualquer conjunto de dados. Portanto, após a realização de experimentos com diferentes parâmetros para o algoritmo, foi concluído que para o cenário deste trabalho, o fator escolhido é $k = 3$.

Antes de prosseguir para a etapa dinâmica, é necessário treinar o modelo com os dados gerados e *salvá-lo* para que o meta-escalonador use o mesmo modelo treinado previamente. Isso é possível através do uso do módulo *pickle*, o qual implementa protocolos binários para serialização e desserialização de um objeto em Python (DOCUMENTATION, 2019). Logo, um objeto Python pode ser transformado em uma *stream* de bytes e o contrário também é verdadeiro, tornando possível armazenar o modelo treinado e recuperá-lo posteriormente no momento da decisão.

Figura 15 – Relatório de classificação do KNN.

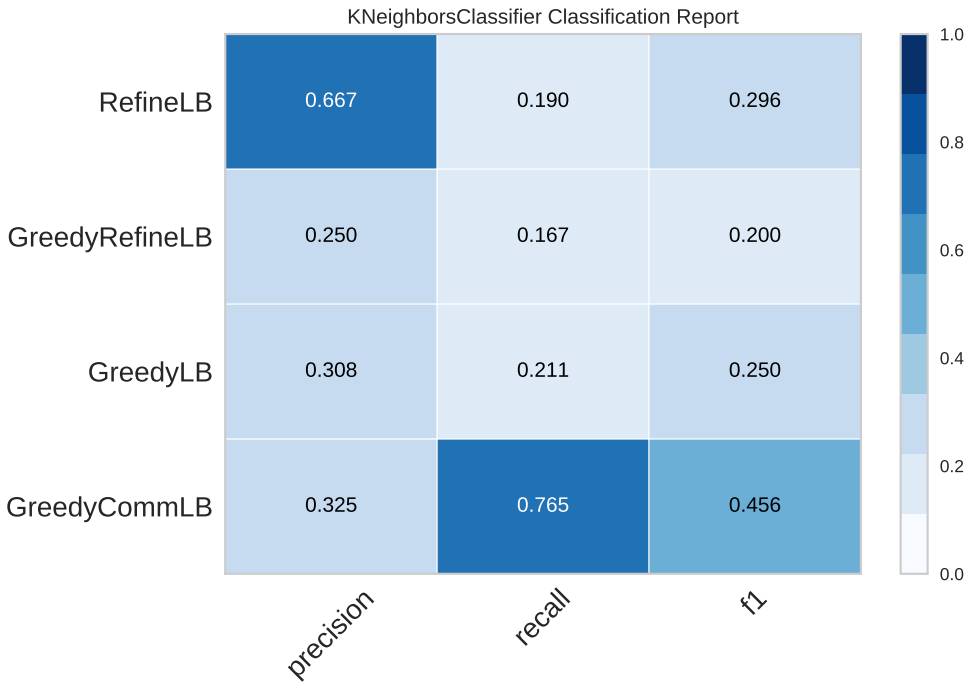
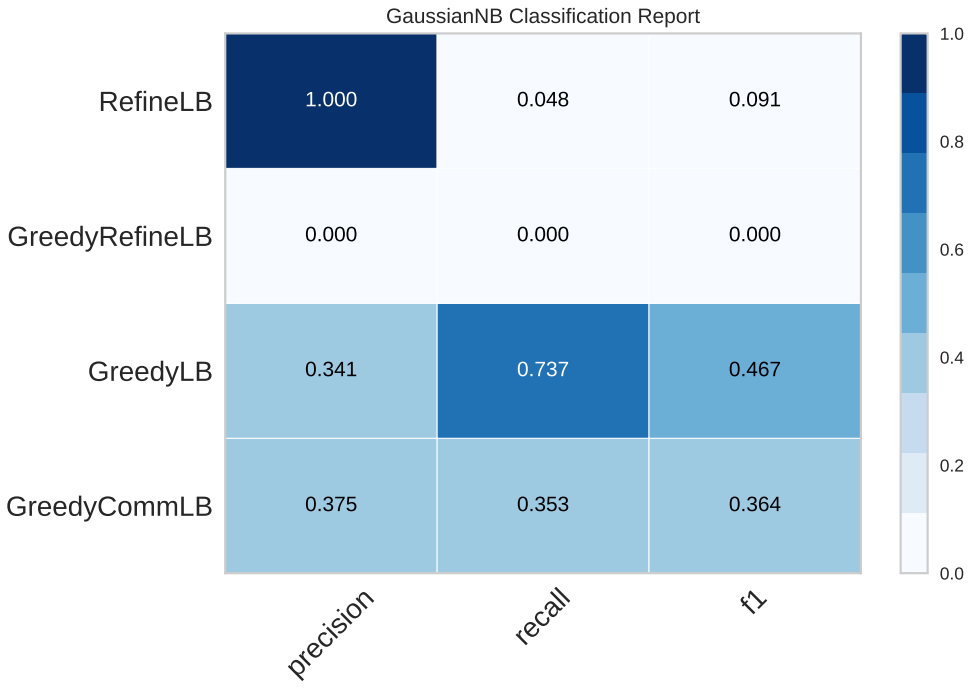


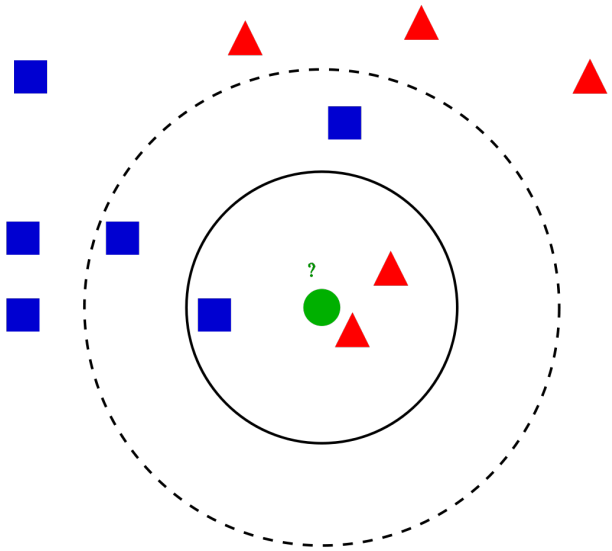
Figura 16 – Relatório de classificação do GaussianNB.



3.4 Balanceamento de Carga em Tempo de Execução

A última etapa do desenvolvimento do projeto é a fase dinâmica, a qual está preparada para receber o meta-escalonador gerado na fase estática e que já se encontra treinado para receber novas instâncias de aplicações reais. Essas aplicações também passam por uma etapa de *extração de características*, para que assim como foi realizado com as amostras da

Figura 17 – Funcionamento do KNN.



Fonte: (WIKIMEDIA, 2016).

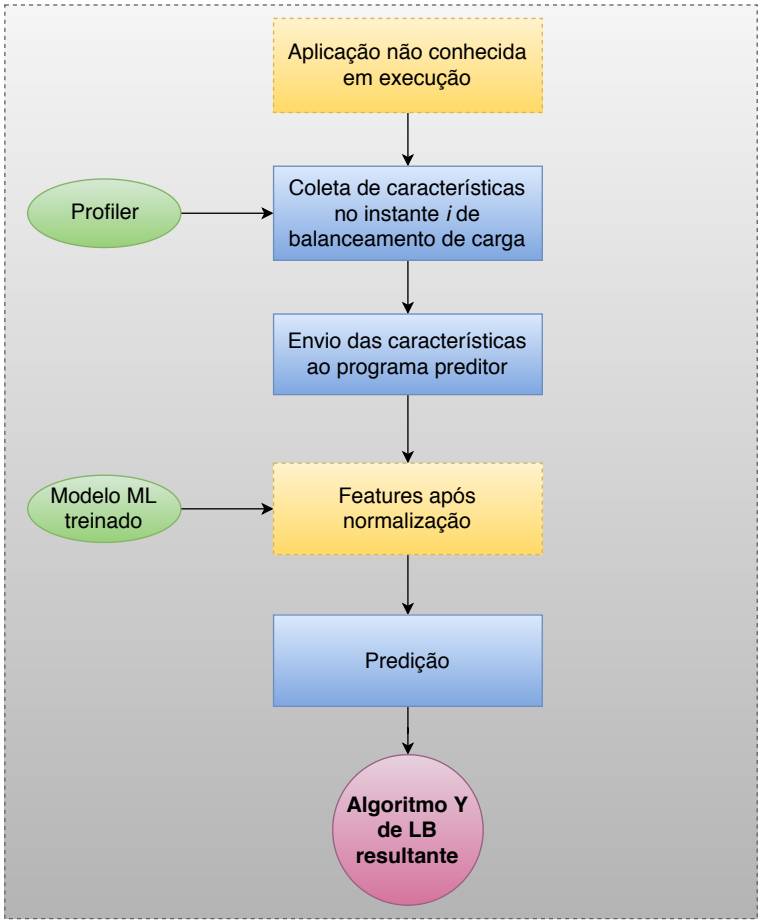
fase estática, seja capturado o conteúdo e o comportamento relevante que as caracterizam. Com base nos dados obtidos, o meta-escalonador decidirá qual é o algoritmo de LB mais apropriado para a aplicação sendo analisada em tempo de execução, fazendo uso do algoritmo KNN escolhido previamente, o qual foi treinado durante a fase estática.

A grande vantagem do trabalho aqui proposto é a possibilidade de efetuar a escolha do algoritmo de LB de forma dinâmica, o que é interessante para aplicações que possuem carga e comportamento que variam durante a execução. Para esses comportamentos irregulares, o algoritmo que havia sido escolhido inicialmente pode deixar de ser o mais apropriado a partir de um determinado instante, tornando necessário a mudança do algoritmo de LB. Portanto, a aplicação deve passar pelo meta-escalonador novamente para que este analise seu atual estado e selecione outro algoritmo mais adequado, caso necessário. A Figura 18 ilustra o processo realizado.

As caixas *13*, *14*, *15* e *16* da Figura 10 são abordadas neste estágio. Inicialmente, é escolhida uma nova aplicação desconhecida durante a fase de treinamento, ou seja, o modelo de ML não sabe nada sobre ela, pois a mesma não pertencia ao conjunto de dados do treinamento. Essa aplicação será observada pelo meta-escalonador em *tempo de execução* através do Perfilador desenvolvido, tendo como objetivo extrair suas características para uso posterior.

No instante em que o Perfilador retorna com a coleta das características da aplicação, o meta-escalonador toma conhecimento sobre ela. Em outras palavras, os dados extraídos são responsáveis por caracterizar aquela instância em particular, e podem ser considerados como a *identidade* da aplicação. Em seguida, essas características podem ser repassadas di-

Figura 18 – Processo de predição em tempo de execução.



Fonte: produzido pela autora.

namicamente ao modelo que foi treinado na fase estática, ou seja, na prática ocorre uma chamada de sistema ao modelo que foi serializado e salvo pelo programa implementado previamente.

Por fim, o classificador analisa as características recebidas e usa o modelo treinado para identificar qual é o algoritmo mais apropriado para aquela aplicação, de acordo com suas características coletadas naquele instante específico no tempo. O meta-escalonador por sua vez, recebe o nome do algoritmo escolhido e invoca o balanceador de carga no código do Charm++ através do seu respectivo método *work*.

Todos os balanceadores de carga existentes no *framework* do Charm++ implementam uma função *work()*. O corpo dessa função é responsável por realizar o balanceamento propriamente dito, ou seja, onde residem todos os comandos das migrações a serem feitas de acordo com a heurística do algoritmo (gulosa, de refinamento, híbrida). Assim, após obter qual é o melhor algoritmo para a aplicação sendo analisada, o meta-escalonador repassa o trabalho para o balanceador de carga que será de fato usado, invocando seu método *work*. Esse processo se repete de acordo com a frequência de LB definida, permitindo a troca de al-

goritmos diversas vezes durante a mesma execução de uma aplicação. O Algoritmo 1 mostra esse processo.

Algoritmo 1: Invocação do algoritmo de LB resultante da previsão.

```
features <– profiler.extract_features();
selected_balancer <– predict_balancer(features);
if selected_balancer == GreedyLB then
    greedyLB–>work()
else if selected_balancer == GreedyCommLB then
    greedyCommLB–>work()
else if selected_balancer == GreedyRefineLB then
    greedyRefineLB–>work()
else
    refineLB–>work()
```

Por fim, é necessário realizar a validação e análise dos resultados obtidos de diferentes execuções do meta-escalonador implementado. Isso envolve a realização de experimentos e testes do meta-escalonador em uma plataforma paralela real e com novas aplicações, analisando os resultados dessas execuções. O próximo capítulo apresenta o ambiente de testes, aborda os experimentos realizados e introduz a análise dos resultados obtidos.

4 Resultados Experimentais

A etapa final é a de validação do algoritmo escolhido para uma dada aplicação executada em uma dada plataforma a fim de verificar que a escolha foi de fato a mais apropriada, de acordo com o monitoramento do sistema realizado pelo meta-escalonador. Ou seja, espera-se que o algoritmo selecionado para a aplicação real recém executada traga um desempenho igual ou melhor ao desempenho do melhor LB disponível no Charm++ RTS para a aplicação em questão. Indo mais além, espera-se que o meta-escalonador tenha desempenho sempre melhor do que a situação em que a aplicação não realiza nenhum tipo de balanceamento de carga.

Para verificar a corretude do meta-escalonador desenvolvido, foi criado um conjunto de 12 instâncias de aplicações sintéticas com o objetivo de validar a proposta introduzida. Essas amostras não foram utilizadas durante a fase de treinamento, e portanto, o meta-escalonador não tem conhecimento sobre elas.

A Seção 4.1 apresenta o ambiente usado para a realização dos experimentos, a Seção 4.2 apresenta o *benchmark* do Charm++ utilizado, enquanto a Seção 4.3 apresenta a análise dos resultados obtidos durante a experimentação.

4.1 Descrição da Plataforma Experimental

A fase de geração de dados e a fase de experimentos foram realizadas na máquina *Tesla* do *Laboratório de Pesquisa em Sistemas Distribuídos* da *Universidade Federal de Santa Catarina*. Abaixo estão listadas as características técnicas referentes à *Tesla:*

- Processador Intel(R) Xeon(R) CPU ES-2640 v4 @ 2.40GHz
- 10 núcleos físicos com *hyper-threading* ativado
- Cache L1: 32 KB de dados, 32 KB de instruções por núcleo
- Cache L2: 256 KB por núcleo
- Cache L3: 25 MB acessíveis por todos os núcleos das CPUs
- 2 Nodos NUMA (20 núcleos físicos + hyperthreading)
- 128 GB de RAM
- NVIDIA Tesla K40c

Hyper-threading é uma forma de duplicar virtualmente os núcleos nas CPUs, sendo útil para tornar os processadores mais eficientes ao distribuir a carga de trabalho entre os núcleos. Entretanto, seu uso não será abordado no contexto deste trabalho, pois o intuito é utilizar somente os núcleos físicos a fim de simplificar a plataforma. Além disso, é importante ressaltar a necessidade de realizar o *bind* das *threads* para núcleos especificados explicitamente, uma vez que migrações indesejadas podem ser feitas pelo SO. Logo, esse mapeamento possui como finalidade evitar que as mesmas migrem para outros processadores durante a execução da aplicação.

Assim, dado que há um total de 20 núcleos físicos e 2 nodos, o objetivo é usar 10 núcleos físicos de cada processador. Isso é possível mapeando manualmente os *chares* para o *id* de cada processador destino, como mostra o comando do *LBTest* abaixo, o qual mapeia o trabalho para os processadores físicos cujo *id* varia de 0 a 19. A Listagem 4.1 mostra esse processo.

```
./charmrun +p20 ./lb_test n_elements steps print_freq lb_freq min_dur
   max_dur topology +balancer balancer +LBDebug 1 +pemap 0-19
```

Listagem 4.1 – Exemplo do mapeamento manual para processadores físicos.

4.2 Benchmark Sintético: LBTest

O *LBTest* é um *benchmark* sintético implementado no *Charm++*, e como o próprio nome indica, é destinado a testar balanceadores de carga. A grande vantagem de seu uso é a flexibilidade na alteração de seus parâmetros para simular uma grande variedade de distribuição de comunicação e computação. Adicionalmente, o *benchmark* permite o mapeamento manual dos processadores físicos conforme exposto na Seção 4.1, alinhando-se com as necessidades deste trabalho.

Um total de 314 instâncias foi gerado na fase de treinamento do modelo e 12 instâncias foram criadas para a realização dos experimentos. As instâncias de treinamento tiveram a seguinte variação de parâmetros:

1. **Número de elementos**: 500 a 9500 (intervalo de 1000)
2. **Iterações**: 101, 601, 1001
3. **Frequência de LB**: 80, 580, 980 (número de iterações)
4. **Topologia**: mesh2d, mesh3d, ring
5. **Frequência de print**: 1
6. **Duração mínima**: 1, 1000, 2000

7. **Duração máxima**: 1000, 2000, 3000, 4000, tal que *duração máxima* > *duração mínima*

8. **Algoritmo de LB**: GreedyLB, GreedyCommLB, GreedyRefineLB, RefineLB

9. **LBDebug**: sempre ativado

O objetivo dos parâmetros apresentados na lista acima é gerar uma grande variedade de perfis de aplicações com diferentes cargas de trabalho, comunicação, topologia, entre outros. Por exemplo, a extração das características responsáveis por caracterizar uma aplicação ocorre no momento do balanceamento de carga (iteração 80, 580 ou 980), e portanto, a coleta realizada nesse instante representa um estado específico x de uma aplicação. Ou seja, uma mesma aplicação i terá diferentes estados (características) dependendo de quando ocorrer a coleta.

Por esse motivo, os valores da *frequência de LB* foram escolhidos de forma a obter diferentes estados que uma aplicação pode assumir. Em um estágio inicial (iteração 80), é provável que o comportamento da aplicação não apresente um desbalanceamento extremo. Já em um estágio intermediário (iteração 580), pode-se representar um cenário onde há um desbalanceamento considerável. Por fim, em um estágio mais avançado (iteração 980), há uma maior probabilidade de obter uma situação com um desbalanceamento de carga maior. Assim, esses valores definidos permitem coletar diferentes cenários de carga que uma aplicação pode assumir em um determinado instante, representando uma alta variedade de comportamentos possíveis durante a execução de uma aplicação.

A frequência de impressão dos dados coletados é sempre a cada iteração, permitindo acompanhar e ter a ciência do que ocorre durante todo o tempo de vida da aplicação. O tempo mínimo e máximo das tarefas são responsáveis por gerar mais desbalanceamento e portanto, criando um cenário satisfatório para comparar e testar os balanceadores de carga, incluindo o meta-escalonador. Por fim, a *flag LBDebug* permanece sempre ativa, pois é necessário acompanhar todos os acontecimentos ocorridos durante esse processo.

4.3 Análise dos Resultados

A realização dos experimentos para validação do meta-escalonador proposto se dá pela observação do tempo total de execução de novas aplicações sintéticas. Para isso é necessário utilizar uma aplicação não vista durante a fase de treinamento e executá-la em seis cenários diferentes: no contexto em em que não ocorre balanceamento de carga, no contexto do meta-escalonador e no contexto de cada um dos quatro algoritmos de balanceamento de carga selecionados previamente. Especificamente no caso do meta-escalonador, é observado também se o preditor seleciona diferentes algoritmos de LB para cada instante do balanceamento, caso a carga da aplicação tenha sofrido alteração.

Foi desenvolvido um *script* responsável por automatizar o processo dos experimentos, como mostra a Listagem 4.2. Assim, para cada tipo de topologia de rede (*ring*, *mesh2d*, *mesh3d*), quatro aplicações diferentes foram criadas e executadas *n* vezes, garantindo que uma anomalia no tempo de execução não afete o tempo de execução que representa aquele caso. Após, é feito o cálculo da média do tempo de execução de cada um desses cenários, permitindo a análise futura desses resultados obtidos.

Assim, é possível compreender os resultados observando o tempo total de execução de uma dada aplicação no cenário em que a mesma não realiza o balanceamento de carga em nenhum momento, comparando-o com o tempo de execução dos cenários em que a mesma aplicação realiza o balanceamento de carga utilizando o meta-escalonador ou algum dos outros algoritmos de LB.

```
File created: 6600_601_100_mesh3d_1000_5000_MetaSchedulerLB
File created: 6600_601_100_mesh3d_1000_5000_GreedyLB
File created: 6600_601_100_mesh3d_1000_5000_GreedyCommLB
File created: 6600_601_100_mesh3d_1000_5000_RefineLBLB
File created: 6600_601_100_mesh3d_1000_5000_GreedyRefineLB
File created: 6600_601_100_mesh3d_1000_5000
```

Listagem 4.2 – Exemplo de saída obtida do *script* de experimentos para uma aplicação *i*.

No caso do exemplo acima, o meta-escalonador teve o **melhor resultado**. Seu tempo de execução total foi cerca de 6 segundos mais rápido do que a aplicação sem balanceamento e cerca de 7 segundos mais rápido do que o melhor tempo do cenário em que usa algum algoritmo de LB escolhido estaticamente. Em outras palavras, para uma aplicação com 6600 elementos, 601 iterações, onde o balanceamento de carga é realizado a cada 100 iterações, com a topologia *mesh3d* e tarefas com durações variando entre 1000 e 5000 *ms*, o meta-escalonador cumpriu seu objetivo de escolher o melhor algoritmo de LB de acordo com as características coletadas no instante de balanceamento. Isso corrobora com a questão apresentada previamente de que escolher um algoritmo de LB estaticamente e utilizá-lo durante todo o tempo de vida da aplicação pode não ser a melhor escolha.

Esse exemplo também é apresentado na Tabela 6, tornando possível visualizar um cenário no qual o uso dos demais algoritmos de balanceamento de carga não compensa o sobrecusto gerado pelo balanceamento. Ou seja, o tempo de execução final da aplicação *sem realizar* o balanceamento de carga foi *menor* do que fazendo uso de qualquer um dos algoritmos. Por outro lado, o meta-escalonador apresentou um resultado satisfatório, uma vez que cumpriu seu objetivo de selecionar dinamicamente o melhor algoritmo de LB de acordo com as características que a aplicação possuía em determinado instante, sem causar tanto impacto negativo no seu desempenho.

Como consequência, foi possível obter o melhor tempo fazendo uso do meta-escalonador

Tabela 6 – Exemplo do experimento realizado.

Algoritmo	Tempo de execução (s)
Meta-escalonador	349.8568832
GreedyLB	357.8009922
GreedyCommLB	358.6406762
RefineLB	369.3020228
GreedyRefineLB	357.219494
Sem LB	355.9751616

Fonte: produzido pela autora.

desenvolvido por conta da sua característica de analisar em tempo real os dados que compõem uma aplicação em particular e tomar a decisão de acordo com as observações realizadas. Neste exemplo, pode-se provar que a aplicação executada possui um comportamento de carga dinâmico, pois a cada coleta de características realizada, o meta-escalonador escolhe um algoritmo de LB *diferente*, de acordo com as necessidades atuais.

A Listagem 4.3 mostra a saída gerada pelo *script* de experimentos a cada instante em que o balanceamento de carga é realizado, juntamente com o algoritmo de LB que foi escolhido para aquele cenário. Assim, nota-se que para o caso da aplicação apresentada, durante os seis momentos em que ocorre o balanceamento de carga o meta-escalonador escolheu três algoritmos diferentes de acordo com as informações observadas em cada instante. Isso comprova que a estratégia proposta é capaz de se adaptar à mudança de comportamento da aplicação, escolhendo a abordagem mais apropriada para o instante atual sendo analisado.

```
Selected balancer: ['GreedyLB']
Selected balancer: ['GreedyCommLB']
Selected balancer: ['GreedyRefineLB']
Selected balancer: ['GreedyRefineLB']
Selected balancer: ['GreedyCommLB']
Selected balancer: ['GreedyRefineLB']
```

Listagem 4.3 – Algoritmos escolhidos pelo meta-escalonador em cada instante do balanceamento de carga.

Os gráficos abaixo mostram os resultados dos experimentos realizados com aplicações não vistas na fase de treinamento, explorando as três topologias de redes e diferentes variações de parâmetros. São realizadas cinco execuções para cada um dos experimentos e a média de seus respectivos tempos de execução é calculada. No caso da topologia *ring*, a Figura 19 aborda duas situações em que o meta-escalonador atingiu o melhor desempenho em termos de tempo de execução, se comparado com as demais estratégias observadas.

O gráfico *ring-1* da Figura 19 mostra que a estratégia implementada levou cerca de

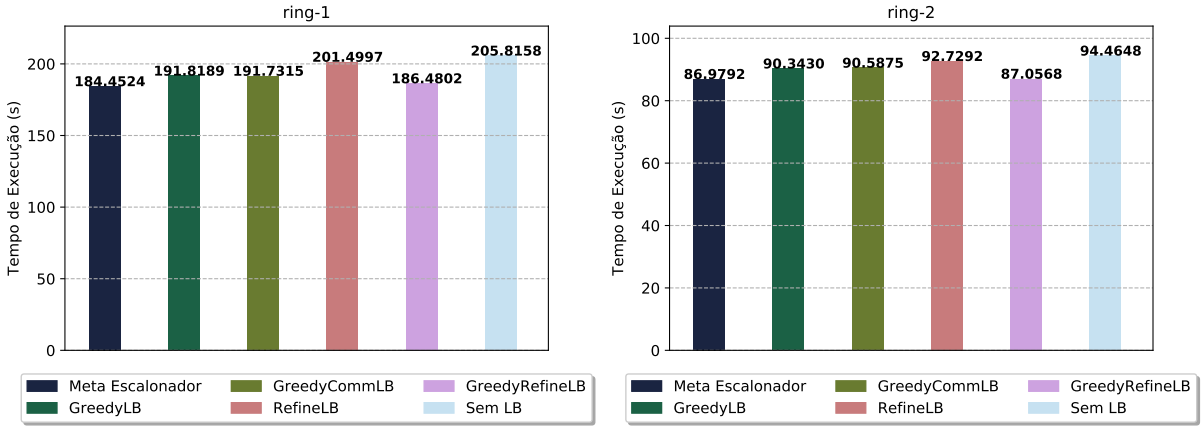


Figura 19 – Experimentos 1 e 2 para a topologia *ring*.

2 segundos a menos do que o algoritmo de LB mais rápido (RefineLB) entre os observados, constatando o melhor desempenho nesse experimento. Além disso, o meta-escalonador foi 21.4 segundos mais rápido do que o tempo da situação em que não ocorre balanceamento de carga, mostrando um ganho significativo de desempenho. Já no gráfico *ring-2*, o meta-escalonador foi cerca de 0.08 segundos mais rápido do que o melhor algoritmo (GreedyRefine) e 7.5 segundos mais rápido do que a estratégia sem balanceamento de carga.

Duas outras situações observadas para a topologia *ring* são apresentadas na Figura 20. O primeiro gráfico mostra a situação em que o meta-escalonador fica com o segundo melhor tempo, com cerca de 0.39 segundos a mais do que o GreedyRefineLB. Entretanto, ainda teve um desempenho satisfatório quando comparado ao cenário sem balanceamento, sendo 9.9 segundos mais rápido. O segundo gráfico mostra que mesmo para aplicações com ciclo de vida menores, o meta-escalonador foi capaz de obter o melhor tempo por conta da representação de aplicações com menos iterações na fase de treinamento. Assim, uma vez que o modelo foi treinado para receber aplicações com diferentes valores de iterações, também é possível escolher o melhor algoritmo para o tipo de situação em que o ciclo de vida da aplicação é menor.

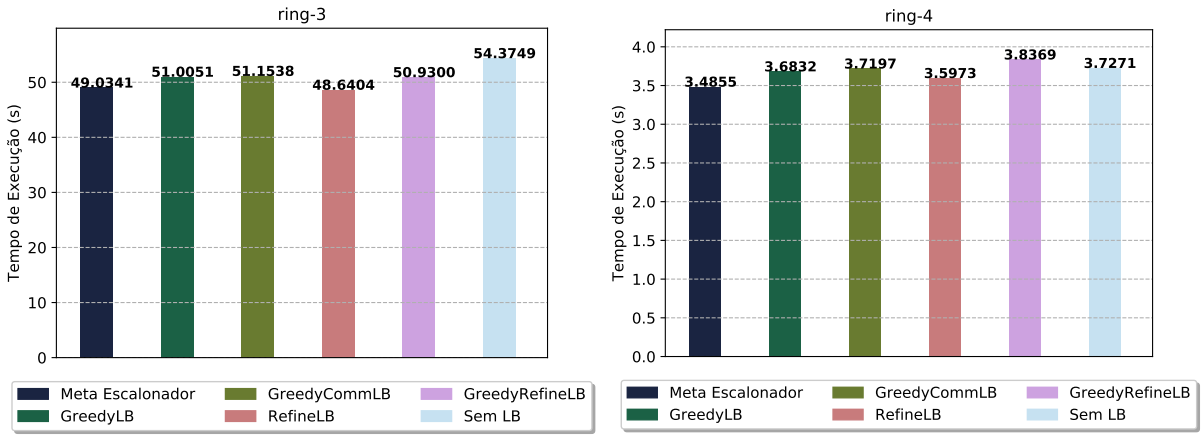


Figura 20 – Experimentos 3 e 4 para a topologia *ring*.

Seguindo para a topologia *mesh2d*, a Figura 21 ilustra dois cenários distintos. O primeiro gráfico indica que o meta-escalonador obteve o melhor tempo de execução, levemente mais rápido do que o tempo do melhor algoritmo de LB (RefineLB) e do que o tempo sem balanceamento de carga. O segundo cenário mostra que a aplicação observada possui como característica um ciclo de vida maior, ou seja, seu tempo de execução é o maior entre os já observados. Neste contexto, o meta-escalonador teve um desempenho muito similar ao do melhor algoritmo e muito melhor do que a situação sem balanceamento. Sendo assim, é apenas 0.57 segundos mais lento do que o GreedyCommLB e 92.47 mais rápido do que o cenário em que a aplicação não realiza balanceamento de carga.

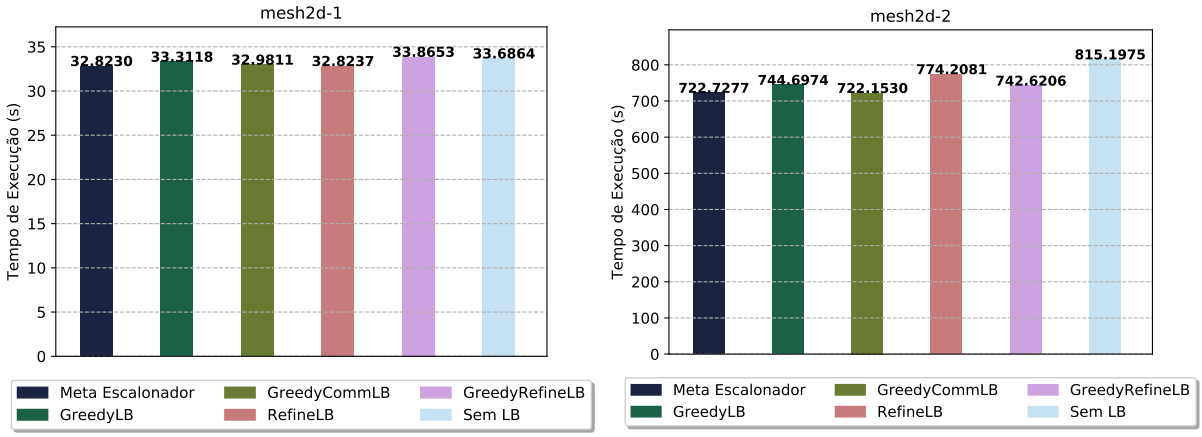


Figura 21 – Experimentos 1 e 2 para a topologia *mesh2d*.

É importante ressaltar que, nas situações em que o meta-escalonador obtém um tempo de execução levemente maior ou muito similar ao tempo da estratégia de melhor desempenho, ainda há a vantagem da automatização do processo de balanceamento. Em outras palavras, não há a necessidade do conhecimento prévio da aplicação para decidir estaticamente e manualmente qual algoritmo de LB será utilizado. Seria necessário conhecer com antecedência as características de uma instância, realizar uma análise dessas informações e então determinar qual algoritmo seria o mais apropriado para a mesma. Porém, uma vez que o meta-escalonador proposto realiza a coleta e análise dessas informações em tempo de execução, seu uso se torna vantajoso em todos os cenários aqui apresentados.

Indo mais além, se por alguma causa incomum uma aplicação tem seu comportamento alterado durante a execução, o algoritmo que havia sido escolhido previamente pode não ser mais apropriado. Ou seja, mesmo que a aplicação não tenha um comportamento dinâmico como característica, ela pode se comportar de outra forma por razões externas, como falhas no *hardware*, por exemplo. Portanto, a escolha manual que funcionava para a maioria dos casos de uma dada aplicação que teve o desempenho levemente melhor do que o meta-escalonador nos experimentos realizados, pode deixar de ser adequada se houver alteração de seu comportamento por diversos fatores externos. No caso da estratégia proposta neste trabalho, a solução acaba cobrindo esse tipo de anomalia, dado que se o com-

portamento da aplicação mudar por causas terceiras, então o preditor realizará a troca do algoritmo de LB em tempo de execução.

Prosseguindo ao caso da Figura 22, os dois gráficos apresentados indicam a situação em que o meta-escalonador foi capaz de atingir o melhor desempenho no contexto das aplicações executadas. No primeiro experimento da figura, nota-se que o ganho foi levemente melhor dado que a aplicação possui um ciclo de vida menor, e portanto, o balanceamento de carga ocorre menos vezes. Já no segundo experimento, o ganho foi maior pois durante o ciclo de vida da aplicação, o balanceador é invocado mais vezes e o meta-escalonador cumpriu com seu objetivo de escolher o algoritmo de LB mais apropriado para cada instante, resultando no melhor desempenho entre as estratégias observadas. Neste último caso, foi possível atingir um tempo de execução aproximadamente 28.26 segundos menor do que a estratégia sem balanceamento de carga.

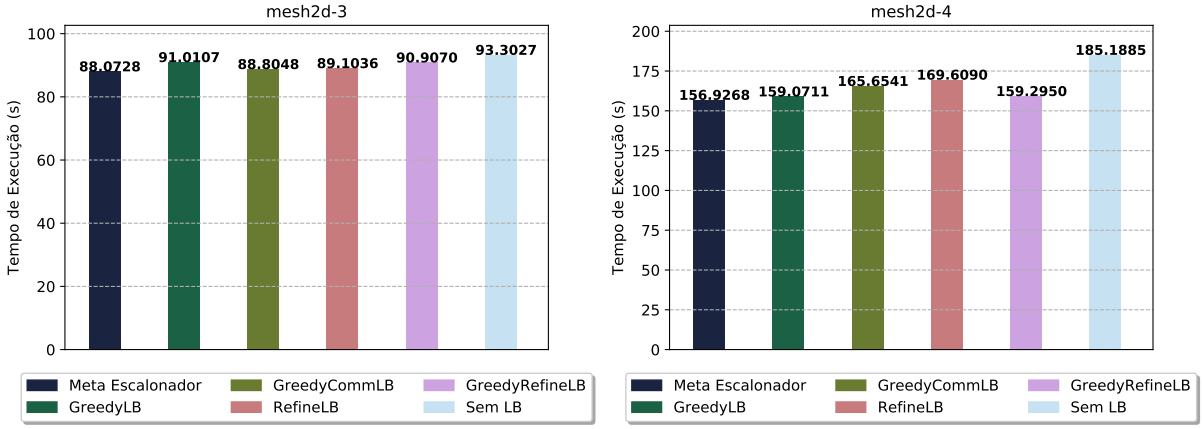


Figura 22 – Experimentos 3 e 4 para a topologia *mesh2d*.

A última topologia analisada é a *mesh3d*, como mostra a Figura 23. No caso do primeiro gráfico, o meta-escalonador levou um tempo aproximadamente 1 segundo mais lento do que o algoritmo primeiro colocado (GreedyCommLB). Por outro lado, a estratégia proposta alcançou um tempo de aproximadamente 19.39 segundos mais rápido do que o cenário sem balanceamento de carga.

O segundo gráfico apresenta uma situação em que um ótimo desempenho foi atingido. O meta-escalonador foi capaz de obter um tempo 6.94 segundos mais rápido do que o segundo colocado. Além disso, a melhora do desempenho ao compará-lo com a estratégia sem balanceamento de carga foi de aproximadamente 90.96 segundos, garantindo um ganho significantemente elevado no contexto observado.

Por fim, a Figura 24 ilustra os dois últimos experimentos realizados, os quais o meta-escalonador foi capaz de obter o melhor desempenho. O primeiro gráfico mostra que seu tempo de execução foi levemente melhor do que a segunda melhor estratégia de LB e significantemente melhor do que a estratégia sem balanceamento. O segundo gráfico ilustra o exemplo mencionado no início desta Seção, onde todas as estratégias de LB acabam tendo o

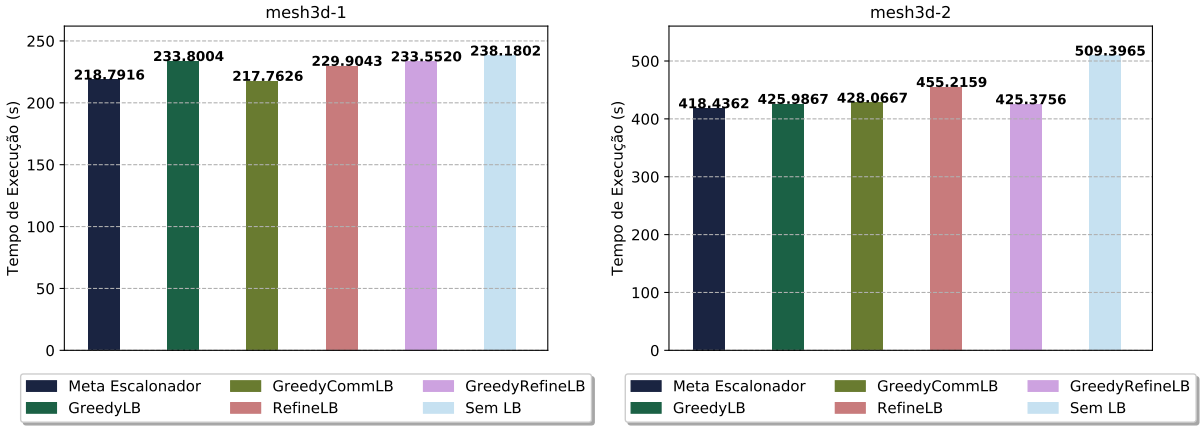


Figura 23 – Experimentos 1 e 2 para a topologia *mesh3d*.

desempenho *pior* do que a situação em que a aplicação não realiza balanceamento de carga. Entretanto, o meta-escalonador foi capaz de superar esse sobrecusto gerado pelos outros algoritmos de LB, visto que o mesmo realiza a troca de estratégia em tempo de execução.

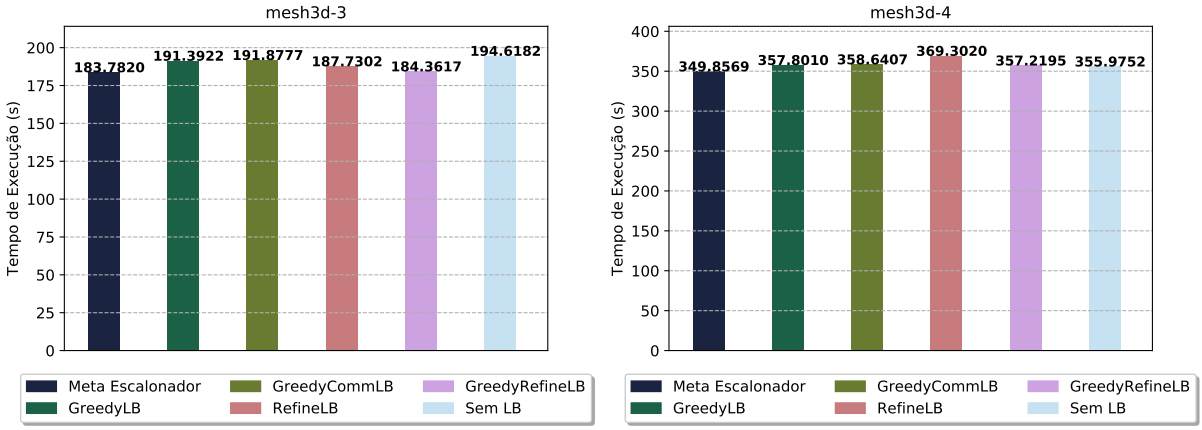


Figura 24 – Experimentos 3 e 4 para a topologia *mesh3d*.

4.3.1 Conclusão

Foi possível observar que o meta-escalonador garantiu um bom desempenho em todos os experimentos realizados. Isso pode ser dividido em diferentes métricas: o número de vezes em que ele foi melhor do que a situação em que não se usa balanceamento de carga, o número de vezes que ele foi mais rápido do que o melhor algoritmo de LB entre os observados, o número de vezes que ele foi o mais lento do que a melhor estratégia e, nessa última situação, o quanto o mesmo foi mais lento.

Combinando os dados apresentados nos gráficos e para facilitar a conclusão obtida através dos experimentos realizados, a Tabela 7 resume essas informações. Assim, para cada um dos exemplos discutidos, o meta-escalonador tem seu tempo de execução comparado com o melhor algoritmo de LB para aquele cenário e com a situação em que a aplicação

não realiza balanceamento de carga. Os valores positivos representam o quanto o meta-escalonador foi mais rápido do que as outras abordagens, e os valores negativos representam o quanto o mesmo foi mais lento do que a melhor estratégia da situação específica.

Tabela 7 – Resultado do desempenho geral do meta-escalonador.

Experimento	Ganho de Desempenho com LB (s)	Ganho de Desempenho sem LB (s)
ring-1	2.0278	21.3634
ring-2	0.0776	7.4856
ring-3	-0.3937	5.3408
ring-4	0.1118	0.2416
mesh2d-1	0.0007	0.8634 0
mesh2d-2	-0,5747	92.4698
mesh2d-3	0.7320	5.2299
mesh2d-4	2.1443	28.2617
mesh3d-1	-1.0290	19.3386
mesh3d-2	6.9394	90.9603
mesh3d-3	0.5797	10.8362
mesh3d-4	7.3626	6.1183

Portanto, através da análise da tabela acima, nota-se que o meta-escalonador *sempre* teve um desempenho melhor do que o caso em que não há balanceamento de carga. Vale lembrar que o experimento *mesh3d-4* representa a situação em que todos os quatro algoritmos de LB observados atingiram um desempenho *pior* do que o cenário sem balanceamento, consequência do sobrecusto gerado pela quantidade de migrações e heurística que cada algoritmo possui. Mesmo nesse caso, o meta-escalonador escolheu corretamente cada estratégia de forma que o sobrecusto fosse menor, e portanto, obtendo o melhor tempo de execução.

Nas demais situações, o meta-escalonador teve o melhor desempenho em *nove* dos doze experimentos realizados e ficou em segundo lugar nos outros *três* casos. Entretanto, além da diferença do tempo de execução ser baixa nas situações em que o seu desempenho não foi o melhor, o meta-escalonador ainda tem a vantagem de automatizar o processo da escolha de um algoritmo de LB de forma orientada à carga de trabalho da aplicação, sem a necessidade de conhecimento prévio sobre a mesma. Isso adiciona um ponto positivo se comparado à abordagem estática, onde seria necessário analisar manualmente a aplicação para compreender todas as suas características e comportamento, e só então escolher um algoritmo de balanceamento para ela.

Outro fator importante que deve ser mencionado, é a questão do sobrecusto ao invocar dinamicamente o modelo de ML treinado. Esse tempo foi medido com a finalidade de verificar se o desempenho do meta-escalonador pode melhorar, caso outra forma de chamar o modelo for utilizada. Assim, observou-se que esse sobrecusto é de aproximadamente 0.69 segundos, o que pode alterar o cenário dos resultados recém apresentados. No caso dos ex-

perimentos *ring-3* e *mesh2d-2*, por exemplo, o meta-escalonador deixaria de ser o segundo colocado e passaria a ser o de melhor desempenho, tornando-se 0.2963 e 0.1153 segundos mais rápido nesses dois casos. Já para o experimento *mesh3d-1*, seu atraso deixaria de ser 1.029 segundos para apenas 0.339 segundos. Essa questão será abordada na Seção 5.1 em mais detalhes.

Por fim, é importante reforçar a justificativa da questão apresentada na Seção 3.3, a qual aborda o valor não tão elevado da acurácia do modelo, mas o mesmo acaba por ter bons resultados. Isso ocorre porque apesar das aplicações que foram utilizadas nessa fase de experimentação serem aplicações sintéticas *não vistas durante a etapa de treinamento*, são aplicações que apresentam características similares àquelas já introduzidas. Ou seja, o modelo não as conhece, porém percebe que o comportamento analisado é parecido com aquelas que ele foi treinado para receber. Dessa forma, o preditor toma conhecimento de que as características das novas aplicações possuem similaridade com as aplicações já conhecidas, tomando então as decisões corretas de acordo com os rótulos atribuídos anteriormente durante a classificação das instâncias no treinamento.

5 Conclusão

Foi visto que ayplicações paralelas podem apresentar uma certa irregularidade de carga proveniente das tarefas decompostas para paralelização, podendo também ter essa carga modificada de forma imprevisível ao longo da execução do programa. Isso resulta em uma aplicação que possui desbalanceamento de carga e no impacto negativo do desempenho da mesma. Para certas aplicações de HPC, foi visto que o balanceamento de carga é um fator crucial para a obtenção do desempenho necessário em máquinas paralelas de larga escala.

O principal problema apresentado por este trabalho é o processo não trivial e desafiador de identificar qual algoritmo de escalonamento global é o mais indicado para uma dada aplicação executada em uma dada plataforma e em um dado instante do tempo. Assim, confirma-se a necessidade de uma entidade responsável por selecionar algoritmos de escalonamento de forma dinâmica e adaptativa e específica para cada aplicação em tempo de execução.

Portanto, o meta-escalonador aqui proposto trata do problema de escalonamento global no contexto de aplicações científicas de HPC, a fim de obter um aumento significativo no desempenho e na produção de resultados mais precisos e em maior escala. Adicionalmente, a estratégia implementada facilita o processo de decisão do algoritmo de LB a ser usado para uma dada aplicação, uma vez que essa escolha é feita pela máquina de forma dinâmica e sem necessidade de conhecimento prévio sobre a aplicação a ser executada.

Os experimentos realizados demonstraram a vantagem de utilizar o meta-escalonador aqui desenvolvido, uma vez que houve um impacto positivo nos resultados diante de aplicações não conhecidas por ele e que foram executadas em uma plataforma real. Adicionalmente, foi comprovado que a estratégia desenvolvida realiza de fato a troca do algoritmo de LB dinamicamente, como foi proposto no início deste trabalho.

Por fim, foi visto que há cenários onde a situação que não ocorre balanceamento de carga pode ser mais rápida do que escolhendo algum algoritmo do *framework* do Charm++ de forma estática, visto que as características de uma aplicação HPC podem mudar ao longo do ciclo de vida da mesma. Assim, o sobrecusto do processo de migração de tarefas pode acabar prejudicando o desempenho a ponto de que a redistribuição de trabalho não tenha um bom resultado. É importante ressaltar que, mesmo nesse cenário, o meta-escalonador foi capaz de obter o melhor desempenho, corroborando com a questão apresentada previamente de que a troca dinâmica do algoritmo de LB pode melhorar o desempenho final da aplicação.

5.1 Trabalhos Futuros

Um dos aspectos que podem melhorar ainda mais os resultados obtidos é a questão do sobrecusto gerado no instante em que o meta-escalonador invoca o modelo de ML treinado para realizar a predição. Atualmente, isso é realizado através de uma chamada de sistema, mas se essa abordagem for incorporada diretamente de alguma forma no código do *Charm++*, há a possibilidade de obter um desempenho melhor. A razão para essa possível melhora é que o tempo resultante do meta-escalonador inclui a invocação externa do modelo de ML previamente treinado. Assim, se futuramente novas alternativas forem exploradas, como por exemplo, implementar o algoritmo de ML manualmente no código do Charm++, o meta-escalonador pode ter um resultado ainda mais satisfatório do que o apresentado neste trabalho.

Outro ponto a ser considerado é a realização de um estudo aprofundado e específico da etapa de *Aprendizagem de Máquina*. Como discutido na Seção 3.3, o modelo pode se tornar mais preciso ao gerar uma quantidade de dados muito maior do que a obtida aqui, considerando que é um processo que demanda uma alta porção de tempo e vai além do escopo deste trabalho. Assim, a geração de aplicações com alta variedade de parâmetros para obter uma representatividade significativa de instâncias é importante, mas será útil somente se houver uma grande quantidade de dados pare reforçar a representação de uma dada aplicação. Isso seria a solução para o aumento da acurácia do modelo e uma possível melhora na escolha do algoritmo durante a execução das aplicações.

Por fim, ainda na etapa de Aprendizagem de Máquina, um estudo aprofundado das características a serem coletadas pode aumentar a qualidade do modelo gerado. Uma vez que essas informações se tornam a principal fonte usada para definir o que caracteriza uma dada aplicação, quanto mais precisa for sua representação, melhor será o resultado final. Logo, seria necessário uma pesquisa específica no ramo de ML para garantir que cada perfil de aplicação esteja sendo replicado corretamente, além da necessidade de ter um número significativo de dados que represente cada uma dessas aplicações.

Referências

ACUN, B.; KALE, L. V. Mitigating processor variation through dynamic load balancing. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. [S.l.: s.n.], 2016.

ALPAYDIN, E. *Introduction to Machine Learning*. 2nd. ed. [S.l.]: The MIT Press, 2010.

BAK, S. et al. Integrating openmp into the charm++ programming model. In: *Third International Workshop on Extreme Scale Programming Models and Middleware*. [S.l.: s.n.], 2017.

BALAJI, P. *Programming Models for Parallel Computing*. [S.l.]: Cambridge, Massachusetts The MIT Press, 2015.

BARNEY, B. *Message Passing Interface (MPI)*. 2019. <https://computing.llnl.gov/tutorials/mpi/>. [Data de acesso: 08-07-2019].

BATHELE, A.; KALE, L.; S., K. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: *Proceedings of the 23rd international conference on Supercomputing*. [S.l.: s.n.], 2009.

BILGE, A. et al. Parallel programming with migratable objects: Charm++ in practice. In: . [S.l.: s.n.], 2014. (SC).

BILGE, A. et al. Power, reliability, and performance: One system to rule them all. *Computer*, v. 49, n. 10, p. 31, 2016.

BROWNLEE, J. *What is a Confusion Matrix in Machine Learning*. 2016. <https://machinelearningmastery.com/confusion-matrix-machine-learning/>. [Data de acesso: 25-05-2019].

BRUNST, H.; MOHR, B. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampir ng. In: MUELLER, M. S. et al. (Ed.). *OpenMP Shared Memory Parallel Programming*. [S.l.]: Springer Berlin Heidelberg, 2008.

BRYAN, G. L.; ABEL, T.; NORMAN, M. L. Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. [S.l.]: ACM, 2001.

CHAUBE, R.; CARINO, R.; BANICESCU, I. Effectiveness of a dynamic load balancing library for scientific applications. In: *Sixth International Symposium on Parallel and Distributed Computing*. [S.l.: s.n.], 2007.

DEVINE, K.; BOMAN, E.; KARYPIS, G. Partitioning and load balancing for emerging parallel applications and architectures. In: *Parallel Processing for Scientific Computing, M. Heroux, A. Raghavan, and H. Simon, eds, SIAM*. [S.l.: s.n.], 2006.

DHINESH, B. L. D.; KRISHNA, P. V. Honey bee behavior inspired load balancing of tasks in cloud computing environments. *Applied Soft Computing*, v. 13, n. 5, 2013.

DOCUMENTATION, P. *pickle - Python Object Serialization*. 2019. <https://docs.python.org/3/library/pickle.html>. [Data de acesso: 29-04-2019].

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, v. 21, n. 9, p. 948–960, 1972.

FREITAS, V. M. C. T.; PILLA, L. L. Comparando diferentes ordenações de tarefas em balanceamento de carga distribuido. In: *XVII Escola Regional de Alto Desempenho*. [S.l.: s.n.], 2017.

FREITAS, V. M. C. T. D. *Balanceamento de Carga Distribuído: Uma Abordagem Orientada A Pacotes*. Tese (Graduação) — Universidade Federal de Santa Catarina, 2017.

GARAVEL, H.; VIHO, C.; ZENDRI, M. System design of a cc-numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *International Journal on Software Tools for Technology Transfer*, v. 3, 2001.

GROPP, W. et al. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, Elsevier Science Publishers B. V., 1996.

HALL, P. et al. *Paper SAS313-2014 An Overview of Machine Learning with SAS ® Enterprise Miner™*. 2014.

HAMID, Y.; SUGUMARAN, M.; BALASARASWATHI, R. Ids using machine learning - current state of art and future directions. *British Journal of Applied Science & Technology*, v. 15, 2016.

HAMID, Y.; SUGUMARAN, M.; JOURNAUX, L. Machine learning techniques for intrusion detection: A comparative analysis. In: *Unternational Conference on Informatics and Analytics*. [S.l.]: ACM, 2016. (ICIA-16).

HANS, L. et al. Parallel computing in quantum chemistry - message passing and beyond for a general ab initio program system. *Future Generation Computer Systems*, v. 11, p. 445 – 450, 1995. ISSN 0167-739X.

HOLLOWELL, C. et al. The effect of numa tunings on cpu performance. *Journal of Physics: Conference Series*, v. 664, n. 9, 2015.

HULTH, A. *Combining Machine Learning and Natural Language Processing for Automatic Keyword Extraction*. Tese (Doutorado) — Stockholm University, 2004.

KAMEDA, H. et al. A performance comparison of dynamic vs. static load balancing policies in a mainframe-personal computer network model. In: *IEEE Conference on Decision and Control*. [S.l.: s.n.], 2000. v. 2.

KANG, S. J.; LEE, S. Y.; LEE, K. M. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015.

KARGER, D. R.; RUHL, M. Simple efficient load balancing algorithms for peer-to-peer systems. In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. [S.l.]: ACM, 2004. (SPAA '04).

KUMAR, V.; GRAMA, A.; VEMPATY, N. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, v. 22, 1994.

LABORATORY, P. P. *The Charm++ Parallel Programming System Manual*. 2017. <http://charm.cs.illinois.edu/manuals/html/charm++/>. [Data de acesso: 30-04-2019].

LABORATORY, P. P. *Parallel Programming with Migratable Objects*. 2017. <http://charm.cs.illinois.edu/research/charm>. [Data de acesso: 13-05-2018].

LECUN, Y.; CORTES, C.; BURGES, C. *The MNIST Database of handwritten digits*. 2019. <http://yann.lecun.com/exdb/mnist/>. [Data de acesso: 28-04-2019].

LEUNG, J. Y. T. *Handbook of scheduling: algorithms, models, and performance analysis*. [S.l.]: Chapman and Hall/CRC, 2004.

LI, Q. *FAST PARALLEL MACHINE LEARNING ALGORITHMS FOR LARGE DATASETS USING GRAPHIC PROCESSING UNIT*. Tese (Doutorado) — Virginia Commonwealth University, 2011.

LIU, Z. et al. Greening geographical load balancing. In: *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. [S.l.]: ACM, 2011. (SIGMETRICS '11).

MANEKAR, A. S. et al. A pragmatic study and analysis of load balancing techniques in parallel computing. In: *International Journal of Engineering Research and Applications*. [S.l.: s.n.], 2012. v. 2.

MANNEL, B. *HPE and NASA Increasingly Transform HPC and Space Exploration with Spaceborne Computer*. [S.l.]: HPC Wire, 2018. <https://www.hpcwire.com/solution_content/hpe/government-academia/hpe-and-nasa-increasingly-transform-hpc-and-space-exploration-with-spaceborne-computer/>. [Data de acesso: 08-07-2019].

MARCO, A. D. et al. High performance computing (hpc) and aerospace research activities at the university of naples federico ii. In: ____. [S.l.: s.n.], 2017. p. 307–318. ISBN 978-981-4759-72-4.

MENESES, E.; KALE, L. V.; BRONEVETSKY, G. Dynamic load balance for optimized message logging in fault tolerant hpc applications. In: *2011 IEEE International Conference on Cluster Computing*. [S.l.: s.n.], 2011. p. 281–289.

MENON, H. *METABALANCER AUTOMATED LOAD BALANCING BASED ON APPLICATION CHARACTERISTICS*. Dissertação (Mestrado) — University of Illinois at Urbana-Champaign, 2012.

MENON, H. *Adaptive load balancing for HPC applications*. Tese (Doutorado) — University of Illinois at Urbana-Champaign, 2016.

MENON, H. et al. Automated load balancing invocation based on application characteristics. In: *IEEE International Conference on Cluster Computing*. [S.l.: s.n.], 2012.

MUDDUKRISHNA, A.; JONSSON, P. A.; BRORSSON, M. Characterizing task-based openmp programs. *PLOS ONE*, Public Library of Science, 2015.

MUDDUKRISHNA, A.; JONSSON, P. A.; BRORSSON, M. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Scientific Programming*, v. 2015, 2015.

MURTY, R.; OKUNBOR, D. Efficient parallel algorithms for molecular dynamics simulations. *Parallel Computing*, v. 25, p. 217–230, 1999.

NAVAUX, P. O. A.; ROSE, C. A. F. D.; PILLA, L. L. Fundamentos das arquiteturas para processamento paralelo e distribuído. In: *XI Escola Regional de Alto Desempenho*. [S.l.: s.n.], 2011.

NIKHIL, J. et al. Charm++ and mpi: Combining the best of both worlds. In: *IEEE 29th International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2015.

PADUA, D. *Encyclopedia of Parallel Computing*. [S.l.]: Springer Publishing Company, Incorporated, 2011. ISBN 9780387097657.

PEARCE, O. T. *LOAD BALANCING SCIENTIFIC APPLICATIONS*. Tese (Doutorado) — Texas A&M University, 2014.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

PILLA, L. *Escalonamento Global Adaptativo para Aplicações Científicas*. 2017.

PILLA, L. L. *Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems*. Tese (Doutorado) — Université de Grenoble, 2014.

PILLA, L. L. et al. Comprehensivebench: a benchmark for the extensive evaluation of global scheduling algorithms. *Journal of Physics: Conference Series*, v. 649, n. 1, 2015.

PILLA, L. L.; MENESES, E. *Programação Paralela em Charm++*. 2015.

RAJPUT, V.; KUMAR, S.; PATLE, V. Performance analysis of uma and numa models. v. 2, 2012.

RASHID, M.; BANICESCU, I.; CARINO, R. L. Investigating a dynamic loop scheduling with reinforcement learning approach to load balancing in scientific applications. In: *International Symposium on Parallel and Distributed Computing*. [S.l.: s.n.], 2008.

RODAMILANS, C. B. *Análise de desempenho de algoritmos de escalonamento de tarefas em grids computacionais usando simuladores*. Dissertação (Mestrado) — Universidade de São Paulo, 2009.

SATHYA, R.; ABRAHAM, A. Comparison of supervised and unsupervised learning algorithms for pattern classification. v. 2, 2013.

SCHLOEGEL, K.; KARYPIS, G.; KUMAR, V. A unified algorithm for load-balancing adaptive scientific simulations. In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. [S.l.]: IEEE Computer Society, 2000. (SC '00).

TANENBAUM, A. S. *Modern operating systems, 3rd Edition*. [S.l.]: Prentice-Hall, 2007.

TINDELL, K. W.; BURNS, A.; WELLINGS, A. J. Allocating hard real-time tasks: An np-hard problem made easy. *Real-Time Systems*, v. 4, p. 145–165, 1992.

VEERASAMY, B. D. Concurrent approach to flynn's mpmd classification through java. v. 10, 2010.

WALT, S. van der; CHRIS, S. C.; VAROQUAUX, G. The numpy array: A structure for efficient numerical computation. *Computing in Science and Eng.*, IEEE Educational Activities Department, v. 13, p. 22–30, 2011.

WIKIMEDIA, C. c. *File:KnnClassification.svg*. [S.l.]: Wikimedia Commons, the free media repository, 2016. <https://commons.wikimedia.org/w/index.php?title=File:KnnClassification.svg&oldid=209850200>. [Data de acesso: 25-05-2019].

WIKIPEDIA. *MNIST Database*. 2019. <https://en.wikipedia.org/wiki/MNIST_database>. [Data de acesso: 29-04-2019].

XIANG, N.; LAXMIKANT, V. K.; RASMUS, T. Scalable asynchronous contact mechanics using charm++. In: *IEEE 29th International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2015.

YELLOWBRICK. *Yellowbrick documentation*. 2016. <https://www.scikit-yb.org/en/latest/api/classifier/classification_report.html>. [Data de acesso: 28-04-2019].

YOU, M.; LI, G. Medical diagnosis by using machine learning techniques. *Data Analytics for Traditional Chinese Medicine Research*, p. 39–79, 2013.

ZARGHAM, M. R. *Computer Architecture: Single and Parallel Systems*. [S.l.]: Prentice-Hall International, 1996.

Apêndices

APÊNDICE A – Código

A.1 Profiler.h

```
#ifndef _PROFILER_H_
#define _PROFILER_H

#include "CentralLB.h"
#include "BaseLB.h"
#include "ckgraph.h"
#include "LBDatabase.h"
#include <iostream>
#include <vector>

class Profiler {
public:
        Profiler(BaseLB::LDStats* stat, bool debugMode) {
                stats = stat;
                debug = debugMode;
        }
        ~Profiler() {}

        std::vector<double> extractFeatures();
        char* statsToString(std::vector<double>);
        void predictBalancer(char*);

        private:
                BaseLB::LDStats* stats;
                bool debug;
};

#endif /*_PROFILER_H_*/
```

A.2 Profiler.C

```
#include "Profiler.h"
#include <cstdio>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>

#define STATS_COUNT 7

using namespace std;

vector<double> Profiler::extractFeatures() {
  BaseLB::ProcStats* proc_stats = stats->procs;
  ProcArray proc_array = ProcArray(stats);
  LBDatabase* lbdatabase = (LBDatabase *)CkLocalBranch(_lbdb);
  std::vector<ProcInfo> procs = proc_array.procs;
  LDCommData* commData;

  /* Helper variables */
  int n_objs = stats->n_objs;
  int pes_count = stats->nprocs();
  int overloaded_pes = 0;
  double max_pe_load = -DBL_MAX;
  double pe_load = 0.0;
  double sum_obj_load = 0.0;
  double lb_gain = 0.0;

  int total_msgs = 0;
  int total_bytes = 0;
  int outside_pe_msgs = 0;
  int outside_pe_bytes = 0;
  int num_neighbors = 0;
  int hops = 0;
  int hop_bytes = 0;

  /* Features */
  double avg_overloaded_pes = 0.0;
  double avg_pe_load = proc_array.getAverageLoad();
  double avg_obj_load = 0.0;
  double load_imbalance = 0.0;
  double avg_idle_time = 0.0;
  double comm_comp_ratio = 0.0;
  double msg_outside_pe_percent = 0.0;


  /* Calculation */
  for(int obj=0; obj < n_objs; obj++) {
```

```
        sum_obj_load += stats->objData[obj].wallTime;
    }
    avg_obj_load = sum_obj_load / n_objs;

    for(int pe = 0; pe < pes_count; pe++) {
      pe_load = procs[pe].getTotalLoad();
      if (pe_load > max_pe_load) {
        max_pe_load = pe_load;
      }
      if (pe_load > avg_pe_load) {
        overloaded_pes++;
      }
      avg_idle_time += proc_stats[pe].idletime;
    }
    avg_idle_time = avg_idle_time/pes_count;

    lbdatabase->getLBDB()->GetCommInfo(total_bytes, total_msgs, outside_pe_msgs, outside_pe_bytes,
          num_neighbors, hops, hop_bytes);
    avg_overloaded_pes = ((float)overloaded_pes/pes_count) * 100;
    load_imbalance = max_pe_load/avg_pe_load;
    comm_comp_ratio = (_lb_args.alpha()*total_msgs + _lb_args.beta()*total_bytes) / (avg_pe_load*pes_count);

    lb_gain = max_pe_load - avg_pe_load;
    msg_outside_pe_percent = ((float) outside_pe_msgs/total_msgs) * 100;

    /* Prints features */
    if (debug) {
      CkPrintf("\nAverage PE load: %lf\n", avg_pe_load);
      CkPrintf("Overloaded PEs: %lf%\n", avg_overloaded_pes);
      CkPrintf("Load imbalance: %lf\n", load_imbalance);
      CkPrintf("Avg idle time: %lf\n", avg_idle_time);
      CkPrintf("Avg obj load: %lf\n", avg_obj_load);
      CkPrintf("Communication computation ratio: %lf\n", comm_comp_ratio);
      CkPrintf("Msg outside PE (%): %lf\n", msg_outside_pe_percent);

      CkPrintf("Expected LB gain: %lf\n", lb_gain);
    }

    CkPrintf("\nfeatures:%lf %lf %lf %lf %lf %lf %lf\n",
    avg_pe_load, avg_overloaded_pes, load_imbalance,
    avg_idle_time, avg_obj_load, comm_comp_ratio, msg_outside_pe_percent);

    vector<double> features {avg_pe_load, avg_overloaded_pes, load_imbalance,
    avg_idle_time, avg_obj_load, comm_comp_ratio, msg_outside_pe_percent};

    return features;
}
```

```
char* Profiler::statsToString(vector<double> features) {
  const char* space = "␣";
  char* result = (char*)malloc(1024);
  result[0] = '\0';

  for(unsigned int i = 0; i < STATS_COUNT; i += 1) {
    const char * value = std::to_string(features[i]).c_str();
    char* res = (char*) malloc(50 + strlen(space) + strlen(value));
    strcpy(res, value);
    strcat(res, space);
    strcat(result, res);
    delete res;
  }

  return result;
}
```

A.3 MetaScheduler.h

```
#ifndef METASCHEDULERLB_H
#define METASCHEDULERLB_H

#include "CentralLB.h"
#include "DistBaseLB.h"
#include "MetaSchedulerLB.decl.h"
#include <vector>

void CreateMetaSchedulerLB();

class MetaSchedulerLB : public CBase_MetaSchedulerLB {
public:
  MetaSchedulerLB(const CkLBOptions &);
  MetaSchedulerLB(CkMigrateMessage *m):CBase_MetaSchedulerLB(m) {}
  ~MetaSchedulerLB() {}

protected:
  virtual bool QueryBalanceNow(int) { return true; };
  virtual void work(LDStats* stats);

private:
  const char *greedyLBString = "GreedyLB";
  const char *refineLBString = "RefineLB";
  const char *greedyRefineLBString = "GreedyRefineLB";
  const char *greedyCommLBString = "GreedyCommLB";

  CentralLB *greedyLB;
  CentralLB *refineLB;
  CentralLB *greedyRefineLB;
  CentralLB *greedyCommLB;
  std::vector<double> features;
};

#endif /* METASCHEDULERLB_H */

/*@{*/
```

A.4 MetaScheduler.ci

```
module MetaSchedulerLB {

extern module CentralLB;
initnode void lbinit(void);

group [migratable] MetaSchedulerLB : CentralLB {
  entry void MetaSchedulerLB(const CkLBOptions &);
};

};
```

A.5 MetaScheduler.C

```
#include "MetaSchedulerLB.h"
#include "Profiler.h"

#define FEATURES_SIZE 7

extern LBAllocFn getLBAllocFn(const char *lbname);

CreateLBFunc_Def(MetaSchedulerLB, "Chooses␣the␣most␣suitable␣LB␣algorithm␣for␣a␣given␣
    application")

MetaSchedulerLB::MetaSchedulerLB(const CkLBOptions &opt): CBase_MetaSchedulerLB(opt) {
  features.resize(FEATURES_SIZE, 0.0);

  lbname = (char*)"MetaScheduler";
  if (CkMyPe() == 0) {
    CkPrintf("[%d]␣MetaSchedulerLB␣created\n",CkMyPe());
  }

  LBAllocFn fn = getLBAllocFn(greedyLBString);
  if (fn == NULL) {
    CkPrintf("LB>␣Invalid␣load␣balancer:␣%s.\n", greedyLBString);
    CmiAbort("");
  }
  BaseLB *glb = fn();
  greedyLB = (CentralLB*)glb;

  fn = getLBAllocFn(refineLBString);
  if (fn == NULL) {
    CkPrintf("LB>␣Invalid␣load␣balancer:␣%s.\n", refineLBString);
    CmiAbort("");
  }
  BaseLB *rlb = fn();
  refineLB = (CentralLB*)rlb;

  fn = getLBAllocFn(greedyRefineLBString);
  if (fn == NULL) {
    CkPrintf("LB>␣Invalid␣load␣balancer:␣%s.\n", greedyRefineLBString);
    CmiAbort("");
  }
  BaseLB *dlb = fn();
  greedyRefineLB = (CentralLB*)dlb;

  fn = getLBAllocFn(greedyCommLBString);
  if (fn == NULL) {
    CkPrintf("LB>␣Invalid␣load␣balancer:␣%s.\n", greedyCommLBString);
    CmiAbort("");
```

```
  }
  BaseLB *gclb = fn();
  greedyCommLB = (CentralLB*)gclb;
}

void MetaSchedulerLB::work(LDStats* stats) {
    bool debug = _lb_args.debug();
    Profiler profiler = Profiler(stats, debug);
    features = profiler.extractFeatures();

    FILE *fp;
    char* stats_string = profiler.statsToString(features);
    int len = strlen(stats_string) + strlen("python␣
        /home/anna.oikawa/final-code/lb-machine-learning/predict_balancer.py␣") + 1;
    char selected_balancer[len];
    char *buf = (char*)malloc(len);

    snprintf(buf, len, "python␣/home/anna.oikawa/final-code/lb-machine-learning/predict_balancer.py␣
        %s", stats_string);
    fp = popen(buf, "r");

    if (fp == NULL) {
        printf("Failed␣to␣run␣command\n" );
        exit(1);
    }

    fgets(selected_balancer, sizeof(selected_balancer)-1, fp);
    CkPrintf("\nSelected␣balancer:␣%s\n\n", selected_balancer);

    if (strcmp(selected_balancer, greedyLBString)) {
        greedyLB->work(stats);
    } else if (strcmp(selected_balancer, refineLBString)) {
        refineLB->work(stats);
    } else if (strcmp(selected_balancer, greedyRefineLBString)) {
        greedyRefineLB->work(stats);
    } else {
        greedyCommLB->work(stats);
    }

    free(buf);
    pclose(fp);
}

#include "MetaSchedulerLB.def.h"
```

A.6 config_dir.sh

```
#!/bin/bash

TOP_DIR=$(pwd)
PARAM_DIR=$TOP_DIR/parameters
FEAT_DIR=$TOP_DIR/features
LBSIM_DIR=$TOP_DIR/lbsim
ALGORITHMS=( GreedyLB GreedyCommLB RefineLB GreedyRefineLB )
TOPOS=( ring mesh2d mesh3d )
MIN_DUR=( 1 1000 2000 3000 )
STEPS=( 101 601 1001 )

if [ "$1" = 1 ]
then
  rm -rf $PARAM_DIR # Writes results from scratch
  rm -rf $FEAT_DIR
  rm -rf $LBSIM_DIR
fi

mkdir $FEAT_DIR
mkdir $PARAM_DIR
mkdir $LBSIM_DIR
cd $PARAM_DIR
for topo in "${TOPOS[@]}"
do
  mkdir $topo
done

for topo in $(ls)
do
  cd $topo
  for min_dur in "${MIN_DUR[@]}"
  do
    mkdir $min_dur
    cd $min_dur
    for ((max_dur=(($min_dur+1000)); max_dur<=4001; max_dur=(($max_dur+1000))))
    do
      mkdir $max_dur
      cd $max_dur
      for steps in "${STEPS[@]}"
      do
        mkdir $steps
        cd $steps
        for algorithm in "${ALGORITHMS[@]}"
        do
          mkdir $algorithm
        done
```

```
            cd ..
        done
        cd ..
    done
    cd ..
  done
  cd ..
done
```

A.7 sim_lbtest.sh

```
#!/bin/bash

# Executes multiple runs of lb_test with different input scenarios
# Redirects the output to a file

# <elements> <steps> <print-freq> <lb-freq> <min-dur us> <max-dur us> <topology> <balancer>

TOP_DIR=$(pwd)
CHARMDIR=~/charm
CHARMC=$CHARMDIR/bin/charmcs
LB_TEST_DIR=$CHARMDIR/tests/charm++/load_balancing/lb_test

LB_SIM=$TOP_DIR/lbsim
OUTPUT_DIR=$LB_SIM/output
AVG_DIR=$LB_SIM/averages
PARAM_DIR=$TOP_DIR/parameters
TOPO_DIR=$PARAM_DIR/topology

if [ "$1" = 1 ]
then
        rm -rf $LB_SIM
fi

mkdir -p $LB_SIM
mkdir -p $OUTPUT_DIR
mkdir -p $AVG_DIR

PRINT_FREQ=1

cd $PARAM_DIR

for topology in $(ls)
do
  cd $topology
  for min_dur in $(ls)
  do
    cd $min_dur
      for max_dur in $(ls)
      do
        cd $max_dur
        for steps in $(ls)
    do
                cd $steps
                lb_freq=$(($steps-21))
                for n_elements in {500..8500..1000}
                do
```

```
                CUR_DIR=$(pwd)
                for balancer in $(ls)
                do
                  cd $LB_TEST_DIR
                    FILE_NAME="$n_elements"_"$steps"_"$lb_freq"_"$topology"_"$min_dur"\
                    _ "$max_dur"_"$balancer"
                    HEADER=( $n_elements $steps $PRINT_FREQ $lb_freq $min_dur $max_dur
                        $topology $balancer )

                    # If file does not exist or is empty
                    if [ ! -f "$OUTPUT_DIR"/"$FILE_NAME" ] || [ $(cat "$OUTPUT_DIR"/"$FILE_NAME"
                        | grep -c STEP."$(($steps-1))") -lt 5 ]; then
                      echo ${HEADER[*]} > "$OUTPUT_DIR"/"$FILE_NAME"
                      # Saves entire output into Output directory
                      for n_run in {1..5}
                      do
                        ./charmrun +p20 ./lb_test $n_elements $steps $PRINT_FREQ $lb_freq $min_dur\
                          $max_dur $topology +balancer $balancer +LBDebug 1 +pemap 0-19\
                              >> "$OUTPUT_DIR"/"$FILE_NAME"
                      done
                    fi

                    if [ ! -f "$AVG_DIR"/"$FILE_NAME" ] || [ $(wc -l <"$AVG_DIR"/"$FILE_NAME") -lt 22
                        ]; then # If file does not exist or is empty
                      # Saves parsed output and average into Averages directory
                      echo ${HEADER[*]} > "$AVG_DIR"/"$FILE_NAME"
                      cat "$OUTPUT_DIR"/"$FILE_NAME" \
                        | grep STEP."$(($steps-1))" \
                            | awk '{print $4, $5, $6}' \
                            | python "$TOP_DIR"/lbtest_avg.py >> "$AVG_DIR"/"$FILE_NAME"
                    fi

                    echo File created: "$FILE_NAME"
                  done
                  cd $CUR_DIR
                done
                cd ..
            done
            cd ..
        done
            cd ..
    done
    cd ..
done
```

A.8 feature_extraction.sh

```
#!/bin/bash

# Executes multiple runs of lb_test with different input scenarios
# Redirects the output to a file

# <elements> <steps> <print-freq> <lb-freq> <min-dur us> <max-dur us> <topology> <balancer>

TOP_DIR=$(pwd)
CHARMDIR=~/charm
CHARMC=$CHARMDIR/bin/charmcs
LB_TEST_DIR=$CHARMDIR/tests/charm++/load_balancing/lb_test

FEAT_DIR=$TOP_DIR/features
OUTPUT_DIR=$FEAT_DIR/output
AVG_DIR=$FEAT_DIR/averages
PARAM_DIR=$TOP_DIR/parameters
TOPO_DIR=$PARAM_DIR/topology

if [ "$1" = 1 ]
then
        rm -rf $FEAT_DIR
fi

mkdir -p $FEAT_DIR
mkdir -p $AVG_DIR
mkdir -p $OUTPUT_DIR

PRINT_FREQ=1
BALANCER=DummyLB

cd $PARAM_DIR

for topology in $(ls)
do
  cd $topology
  for min_dur in $(ls)
  do
    cd $min_dur
    for max_dur in $(ls)
    do
      cd $max_dur
      for steps in $(ls)
      do
        CUR_DIR=$(pwd)
        cd $steps
        lb_freq=$(($steps-21))
```

```
        cd $LB_TEST_DIR
        for n_elements in {500..8500..1000}
        do
          FILE_NAME="$n_elements"_"$steps"_"$lb_freq"_"$topology"_"$min_dur"_"$max_dur"
          HEADER=( $n_elements $steps $PRINT_FREQ $lb_freq $min_dur $max_dur $topology )

          # If file does not exist or is empty
          if [ ! -f "$OUTPUT_DIR"/"$FILE_NAME" ] || [ $(cat "$OUTPUT_DIR"/"$FILE_NAME" | grep -c
              features) -lt 3 ]; then
            echo ${HEADER[*]} > "$OUTPUT_DIR"/"$FILE_NAME"
            # Saves entire output into Output directory
            for n_run in {1..3}
            do
              ./charmrun +p20 ./lb_test $n_elements $steps $PRINT_FREQ $lb_freq $min_dur $max_dur
                  $topology +balancer $BALANCER +pemap 0-19\
              >> "$OUTPUT_DIR"/"$FILE_NAME"
            done
          fi

          if [ ! -f "$AVG_DIR"/"$FILE_NAME" ] || [ $(wc -l <"$AVG_DIR"/"$FILE_NAME") -lt 2 ]; then # If file
              does not exist or is empty
            # Saves parsed output and average into Averages directory
            echo ${HEADER[*]} > "$AVG_DIR"/"$FILE_NAME"
            cat "$OUTPUT_DIR"/"$FILE_NAME" \
              | grep features. \
                      | cut -d ':' -f 2,3,4,5,6,7,8 \
              | python "$TOP_DIR"/feature_avg.py >> "$AVG_DIR"/"$FILE_NAME"
                fi

          echo File created: "$FILE_NAME"
        done
        cd $CUR_DIR
      done
      cd ..
        done
        cd ..
  done
  cd ..
done
```

A.9 test_results.sh

```
#!/bin/bash

TOP_DIR=$(pwd)
SCRIPT_DIR=$TOP_DIR/../lb-test-handling
CHARMDIR=/home/$USER/charm
CHARMC=$CHARMDIR/bin/charmcs
LB_TEST_DIR=$CHARMDIR/tests/charm++/load_balancing/lb_test

OUTPUT_DIR=$TOP_DIR/output
AVG_DIR=$TOP_DIR/averages
RES_DIR=$TOP_DIR/results
BALANCERS=( MetaSchedulerLB GreedyLB GreedyCommLB RefineLB GreedyRefineLB )

if [ "$1" = 1 ]
then
        rm -rf $OUTPUT_DIR
        rm -rf $AVG_DIR
        rm -rf $RES_DIR
fi

mkdir -p $OUTPUT_DIR
mkdir -p $AVG_DIR
mkdir -p $RES_DIR

cd $LB_TEST_DIR

n_elements=6400
steps=3000
lb_freq=300
topology=mesh3d
min_dur=1
max_dur=3000
print_freq=1

FILE="$n_elements"_"$steps"_"$lb_freq"_"$topology"_"$min_dur"_"$max_dur"
touch "$RES_DIR"/"$FILE"

for balancer in "${BALANCERS[@]}"
do
        FILE_NAME="$FILE"_"$balancer"
        if [ ! -f "$OUTPUT_DIR"/"$FILE_NAME" ] || [ $(cat "$OUTPUT_DIR"/"$FILE_NAME" | grep -c
                STEP."$(($steps-2))") -lt 5 ]; then # If file does not exist or is empty
                touch "$OUTPUT_DIR"/"$FILE_NAME"

                for n_run in {1..5}
                do
```

```
                ./charmrun +p20 ./lb_test $n_elements $steps $print_freq $lb_freq $min_dur
                    $max_dur $topology +balancer $balancer +LBDebug 1 +pemap 0-19\
                        >> "$OUTPUT_DIR"/"$FILE_NAME"
            done
        fi

        if [ ! -f "$AVG_DIR"/"$FILE_NAME" ] || [ $(wc -l <"$AVG_DIR"/"$FILE_NAME") -lt 22 ]; then # If file
            does not exist or is empty
        # Saves parsed output and average into Averages directory
        touch "$AVG_DIR"/"$FILE_NAME"
        cat "$OUTPUT_DIR"/"$FILE_NAME" \
                | grep STEP."$(($steps-2))" \
                | awk '{print $4, $5, $6}' \
                | python3 "$SCRIPT_DIR"/lbtest_avg.py >> "$AVG_DIR"/"$FILE_NAME"
        fi
        echo File created: "$FILE_NAME"

        time=$(grep "." "$AVG_DIR"/"$FILE_NAME" | tail -1)
        echo "$balancer:␣${time}" >> "$RES_DIR"/"$FILE"
done

if [ ! -f "$OUTPUT_DIR"/"$FILE" ] || [ $(cat "$OUTPUT_DIR"/"$FILE" | grep -c STEP."$(($steps-2))") -lt 5 ];
    then # If file does not exist or is empty
touch "$OUTPUT_DIR"/"$FILE"

        for n_run in {1..5}
        do
                ./charmrun +p20 ./lb_test $n_elements $steps $print_freq $lb_freq $min_dur $max_dur
                    $topology +LBDebug 1 +pemap 0-19\
                        >> "$OUTPUT_DIR"/"$FILE"
        done
fi

if [ ! -f "$AVG_DIR"/"$FILE" ] || [ $(wc -l <"$AVG_DIR"/"$FILE") -lt 22 ]; then # If file does not exist or is
    empty
# Saves parsed output and average into Averages directory
        touch "$AVG_DIR"/"$FILE"
        cat "$OUTPUT_DIR"/"$FILE" \
                | grep STEP."$(($steps-2))" \
                | awk '{print $4, $5, $6}' \
                | python3 "$SCRIPT_DIR"/lbtest_avg.py >> "$AVG_DIR"/"$FILE"
fi
echo File created: "$FILE"

time=$(grep "." "$AVG_DIR"/"$FILE" | tail -1)
echo "No␣balancer:␣${time}" >> "$RES_DIR"/"$FILE"
```

A.10 compare_ml_algorithms.py

```
'''
    Compare Machine Learning algorithms
'''
import graphviz
import itertools
import matplotlib.pyplot as plt
import pandas
import numpy as np
import sys

from sklearn import preprocessing
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import log_loss
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from yellowbrick.classifier import ClassificationReport

SHOW_GRAPHICS = True

# load dataset
dataset = pandas.read_csv("dataset.csv")
values = dataset.values
header = (dataset.columns.values.tolist())
header = header[:(len(header)-1)]
features = values[:,0:len(header)]
classes = values[:,len(header)]
labels = ('GreedyCommLB', 'GreedyLB', 'GreedyRefineLB', 'RefineLB')

if (len(sys.argv) > 1):
    SHOW_GRAPHICS = False

'''
    Plots a confusion matrix
'''
def plot_confusion_matrix(cm,
```

```
                          target_names,
                          title='Confusion matrix',
                          cmap=None,
                          normalize=True):

    if cmap is None:
        cmap = plt.get_cmap('Blues')

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    fig, ax = plt.subplots()
    fig.tight_layout()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    classes = target_names
    ax.set(xticks=np.arange(cm.shape[1]),
                yticks=np.arange(cm.shape[0]),
                xticklabels=classes, yticklabels=classes,
                title=title,
                ylabel='True label',
                xlabel='Predicted label')

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
                ax.text(j, i, format(cm[i, j], fmt), ha="center", va="center", color="white" if cm[i, j] > thresh
                    else "black")

    accuracy = np.trace(cm) / float(np.sum(cm))
    misclass = 1 - accuracy

    plt.tight_layout()
    plt.grid(None)
    fig.savefig(title+'.pdf', bbox_inches='tight')
    plt.clf()
# ----------------------------
# class and features normalization
label_encoder = preprocessing.LabelEncoder()
label_encoder.fit(classes)
classes = label_encoder.transform(classes)
features = preprocessing.normalize(features)

# split train and test data
```

```
data_train, data_test, target_train, target_test = train_test_split(features, classes, test_size = 0.2, random_state
    = 14)

# compare algorithms
classifiers = [
    KNeighborsClassifier(n_neighbors=3),
    SVC(kernel="rbf", C=0.025, probability=True),
    DecisionTreeClassifier(max_depth=20),
    RandomForestClassifier(n_estimators=50, max_depth=20),
    AdaBoostClassifier(n_estimators=80),
    GradientBoostingClassifier(),
    GaussianNB(),
    LinearDiscriminantAnalysis(),
    QuadraticDiscriminantAnalysis()
    ]

results=[]
names=[]
acc_names=[]
accuracy=[]

log_cols=["Classifier", "Accuracy", "Log␣Loss"]
log = pandas.DataFrame(columns=log_cols)

# Compare classifiers
for clf in classifiers:
    clf.fit(data_train, target_train)
    name = clf.__class__.__name__

    print("="*30)
    print(name)

    print('****Results****')
    train_predictions = clf.predict(data_test)
    acc = accuracy_score(target_test, train_predictions, normalize = True)
    acc_names.append(name)
    accuracy.append(acc)
    print("Accuracy:␣{:.4%}".format(acc))

    cm = confusion_matrix(target_test, train_predictions)
    plot_confusion_matrix(cm, labels, title=name+'␣Confusion␣Matrix')

    train_predictions = clf.predict_proba(data_test)
    ll = log_loss(target_test, train_predictions)
    print("Log␣Loss:␣{}".format(ll))

    log_entry = pandas.DataFrame([[name, acc*100, ll]], columns=log_cols)
```

```
    log = log.append(log_entry)

    # prepare configuration for cross validation test harness
    seed = 7
    scoring = 'accuracy'
    kfold = KFold(n_splits=4, random_state=seed)
    cv_results = cross_val_score(clf, features, classes, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "Mean: %f\nStd: (%f)" % (cv_results.mean(), cv_results.std())
    print(msg)

    precision = cross_val_score(clf, features, classes, cv=kfold, scoring='precision_weighted')
    print ("Precision: " + str(round(100*precision.mean(), 2)) + "%")

    recall = cross_val_score(clf, features, classes, cv=kfold, scoring='recall_weighted')
    print ("Recall: " + str(round(100*recall.mean(), 2)) + "%")

    print("Accuracy: %0.2f (+/- %0.2f)" % (cv_results.mean(), cv_results.std() * 2))

    # Visualization
    if (SHOW_GRAPHICS):
        visualizer = ClassificationReport(clf, classes=labels,cmap='Blues', fontsize=16)
        visualizer.fit(data_train, target_train) # Fit the training data to the visualizer
        visualizer.score(data_test, target_test) # Evaluate the model on the test data
        g = visualizer.poof(outpath=name+'.pdf')
        plt.clf()
# ------------------------------
plt.clf()

# Plots algorithm comparison
height = accuracy
bars = acc_names
y_pos = np.arange(len(bars))

# Create horizontal bars
plt.barh(y_pos, height)
ax = plt.gca()
ax.set_ylim(ax.get_ylim()[::-1])

plt.yticks(y_pos, bars)
plt.xlabel('Accuracy (%)')
plt.title('Accuracy Results')

plt.savefig('accuracyComparision.pdf', bbox_inches='tight')
plt.close()
```

A.11 gen_ml_data.py

```
'''
        Creates a .csv file where every row is a set of features and its corresponding class
'''
import csv
import glob
import os
import sys

LB_SIM_DIR='./lbsim/averages/'
FEAT_SIM_DIR='./features/averages/*'
DATASET_FILE_RESULT='../lb-machine-learning/dataset.csv'
HEADER= ['AvgPELoad', 'AvgOverloadedPEs', 'LoadImbalance','AvgIdleTime',
                                'AvgObjLoad', 'CommCompRatio', 'MsgOutsidePePercent', 'Class']


def main():
        # Iterate through features directory and get the best balancer (min execution time) for the feature
            parameters configuration
        results_csv=[]

        for filename in glob.glob(FEAT_SIM_DIR):
                result = []
                with open(filename, 'r') as file:
                        name_list = (file.name).split('/')
                        name = name_list[len(name_list) - 1]
                        balancer = get_lb_algorithm(name)
                        if (balancer == ''):
                                continue
                        features = (file.readlines())[1].split('␣')
                        for feature in features:
                                result.append(feature.replace('\n', ''))
                        result.append(balancer.replace('\n', ''))
                        results_csv.append(result)
        write_csv_result(results_csv)


# Returns the algorithm whose execution time was the smallest for a given application config
def get_lb_algorithm(filename):
        balancer = ''
        min_exec = sys.float_info.max
        lines=[]
        path = LB_SIM_DIR + filename + '_*'

        for filename in glob.glob(path):
                with open(filename, 'r') as f:
                        lines = (f.readlines())
```

```
                first = lines[0].split('␣')
                exec_time = float(lines[len(lines) − 1])
                if (exec_time < min_exec):
                        balancer = first[len(first) − 1]
                        min_exec = exec_time
        return balancer


def write_csv_result(results):
        mode = 'w'
        with open(DATASET_FILE_RESULT, mode) as f:
                csv_writer = csv.writer(f);
                if (mode == 'w'):
                        csv_writer.writerow(HEADER)
                csv_writer.writerows(results)


if __name__ == '__main__':
        main()
```

A.12 train_model.py

```
# Train a machine learning model and saves it for later use

import pandas
import pickle

from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

TRAINED_MODEL_FILENAME = 'trained_model.pickle'
LABEL_ENCODER_FILENAME='label_encoder.pickle'
DATASET_FILE='dataset.csv'

# load dataset
dataset = pandas.read_csv(DATASET_FILE)
values = dataset.values
header = (dataset.columns.values.tolist())
header = header[:(len(header)-1)]

features = values[:,0:len(header)]
classes = values[:,len(header)]

# normalize classes
label_encoder = preprocessing.LabelEncoder()
label_encoder.fit(classes)
classes = label_encoder.transform(classes)

# normalize features
features = preprocessing.normalize(features)

# split train and test data
data_train, data_test, target_train, target_test = train_test_split(features, classes, test_size = 0.2, random_state
    = 14)

# train the model
model = KNeighborsClassifier(3)
model.fit(data_train, target_train)

# saves the trained model
with open(TRAINED_MODEL_FILENAME, 'wb') as file:
        pickle.dump(model, file)

with open(LABEL_ENCODER_FILENAME, 'wb') as file:
        pickle.dump(label_encoder, file, pickle.HIGHEST_PROTOCOL)
```

A.13 predict_balancer.py

```
# Predicts a load balancer for a given set of parameters (features)

import pandas
import pickle
import sys

from sklearn import preprocessing

TRAINED_MODEL_FILE = '/home/anna.oikawa/code/lb-machine-learning/trained_model.pickle'
LABEL_ENCODER_FILE='/home/anna.oikawa/code/lb-machine-learning/label_encoder.pickle'

def main():
        features = []
        for i in range(1, len(sys.argv)):
                features.append(float(sys.argv[i]))

        dataset = []
        dataset.append(features)

        features = preprocessing.normalize(dataset)

        label_encoder = pickle.load(open(LABEL_ENCODER_FILE, "rb"))
        loaded_model = pickle.load(open(TRAINED_MODEL_FILE, "rb"))

        result = loaded_model.predict(features)
        predicted_label = label_encoder.inverse_transform(result)

        print predicted_label

if __name__== "__main__":
  main()
```

A.14 exp_graph.py

```
'''
        Plots graphics comparing Meta-Scheduler execution times
'''

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import matplotlib.patches as mpatches

x = ["Meta Escalonador", "GreedyLB", "GreedyCommLB", "RefineLB", "GreedyRefineLB", "Sem LB"]
data = [[184.452368, 191.8188764, 191.73146079999998, 201.499678, 186.4801958, 205.81581539999996],
[86.97920919999999, 90.3429716, 90.5875338, 92.7292136, 87.0568098, 94.46484960000001],
[49.034089200000004, 51.00513959999999, 51.153793799999995, 48.6403998, 50.929979, 54.374901],
[3.4855270000000003, 3.683226, 3.7197158000000003, 3.597314, 3.8369302, 3.7271358],
[32.82300740000001, 33.311780399999996, 32.9810916, 32.8237392, 33.8652782, 33.686424],
[722.7277052000001, 744.6974478, 722.1529704000001, 774.2081433999999, 742.6205733999999,
    815.1975058],
[88.07283760000001, 91.01067, 88.8047752, 89.10361759999999, 90.90698979999999, 93.3027318],
[156.9267704, 159.0710558, 165.6540974, 169.6089894, 159.29496039999998, 185.18854579999999],
[218.791617, 233.80039820000002, 217.76257220000002, 229.90432700000002, 233.5519632, 238.180242],
[418.43621240000004, 425.98666440000005, 428.0666556, 455.21587639999996, 425.3755502,
    509.39649739999993],
[183.7820082, 191.3922186, 191.8776762, 187.73018739999998, 184.3617004, 194.6181548],
[349.8568832, 357.8009922, 358.64067620000003, 369.3020228, 357.219494, 355.9751616]]
names = ['ring-1', 'ring-2', 'ring-3', 'ring-4',
                'mesh2d-1', 'mesh2d-2', 'mesh2d-3', 'mesh2d-4',
                'mesh3d-1', 'mesh3d-2', 'mesh3d-3', 'mesh3d-4']

#Set tick colors:
pallete=sns.color_palette("cubehelix", 6)

for i in xrange(0, len(data)):
        title=names[i]
        plt.title(title)
        plt.ylabel(u'Tempo de Execu\u00E7\u00E3o (s)', fontsize=11)

        y = data[i]
        s = pd.Series(
            y,
            index = x
        )
        y = [round(n, 4) for n in y]

        ax = plt.gca()
        for i, v in enumerate(y):
```

```
        ax.text(i − .5, v, '{:10.4f}'.format(v), color='black', fontweight='bold')
    ax.axes.get_xaxis().set_visible(False)
    ax.set_xlim(5, None)

    # Plot the data:
    s.plot(
        kind='bar',
        color=pallete,
        width=0.4,
        fontsize=11,
    )

    me = mpatches.Patch(color=pallete[0], label=x[0])
    glb = mpatches.Patch(color=pallete[1], label=x[1])
    gclb = mpatches.Patch(color=pallete[2], label=x[2])
    grlb = mpatches.Patch(color=pallete[3], label=x[3])
    rlb = mpatches.Patch(color=pallete[4], label=x[4])
    nolb = mpatches.Patch(color=pallete[5], label=x[5])

    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1, box.width, box.height * 0.9])
    plt.legend(handles=[me,glb,gclb,grlb,rlb,nolb], loc='upper␣center', bbox_to_anchor=(0.5, −0.05),
        fancybox=True, shadow=True, ncol=3, fontsize=11)
    plt.xlim(−0.5, 5.6)
    plt.margins(0.1)
    ax.grid(which='major', axis='y', linestyle='−−')
    #plt.show()
    plt.savefig(title+'.pdf', bbox_inches='tight')
    plt.close()
```

A.15 feature_avg.py

```
'''
        Calculates the average values for a set of features
'''
import csv
import sys

avg = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
input = sys.stdin.readlines()

for line in input:
        columns = line.split(" ");
        i = 0;
        for col in columns:
                avg[i] += float(col)
                i += 1

for i in range(len(avg)):
        avg[i] = avg[i]/len(input)

print(" ".join(str(x) for x in avg))
```

A.16 lbtest_avg.py

```
'''
        Calculates the average values for a set of execution times
'''
import sys

sum = 0
count = 0;
for line in sys.stdin:
        print (line)
        count += 1
        sum += float(line.split("␣")[1])

avg = sum/count;
print(avg)
```

APÊNDICE B – Artigo

Dynamic Load Balancing for Parallel Applications - A Machine Learning Approach

Anna Victoria Cabrera Rondon Oikawa
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
anna4victoria@gmail.com

***Abstract*—Scientific applications demand high computational power to achieve accurate and scalable results in large systems, but in order to entirely exploit computational resources, High Performance Computing (HPC) platforms are used in association with Parallel Computing interfaces. A HPC application can be decomposed into parallel tasks, where every task has a specific purpose, a particular behavior and a different work load, which can lead to a load imbalance scenario in the system. Such class of applications can have its performance affected by the uneven distribution of work, and as a consequence, the need for rescheduling imbalanced tasks arises as a priority.**

Choosing a load balancing strategy statically may not be the right approach for applications that have a dynamic and unpredictable nature, since it is not trivial to determine which scheduling algorithm is the most suitable for a given application. Additionally, for scenarios where the application characteristics change over time, it is necessary to use a different load balancing strategy that adapts itself to a new context, which makes the manual algorithm selection approach ineffective. In this paper, we propose a Machine Learning-based Meta-Balancer to automate the load balancing algorithm decision at runtime. This approach monitors and collects information about the application dynamically, and according to the analyzed data, it makes a decision by invoking the selected load balancing strategy. Experiments show that this Meta-Balancer improves the performance of existing approaches up to 2% and up to 11.34% in the context where no load balancing occurs. Additionally, results show that Meta-Balancer is able to switch between algorithms different times during an application execution.

***Index Terms*—load balancing; machine learning; charm++; meta-balancer; hpc.**

I. Introduction

One of the main factors that contributes to the progress of Science is the performance of applications developed in large Research Institutes. A variety of fields in the context of Spacial Researches, for instance, needs complex numerical simulations and high performance in order to obtain accurate results in large scale [1], [2]. There has been a constant and significant increase in the number of cores in modern processors, and as a result, recent platforms contain hundreds of thousands of cores. But in order to fully exploit the benefits of this core growth, applications that will be executed in such enviroments must be adapted and optimized with specific purposes. Therefore, in order to satisfy the need of high computational power, HPC platforms and Parallel Computing Interfaces are used to run applications whose main goal is to exploit available resources in order to reach high performance. Thus, the development of parallel programs has extreme importance in this context, since the parallelism of processors tends to increase [3], [4].

Applications that run on large-scale environments have a complex goal to be achieved, and hence, the domain of the program needs to be decomposed into smaller tasks that will be executed in multiple processors, where each task performs a specific work so that later, it is possible to combine these smaller results into a final result. It should be taken into account that it is not always possible to predict how much work each task will receive before the application begins its execution. Therefore, a irregular behavior may arise from the tasks that perform different roles in the application, in addition to the possibility of the system load to change dynamically throughout the execution [5], [6], [7]. Task dependency brings more complexity to the system, since it is a challenge to estimate when a given task will start and finish its execution so that its dependent tasks can start executing [8].

In most situations, the overhead related to the load differences in processors is the dominant factor responsible for terminating their respective executions in different moments. Some processors may remain idle while others are overloaded, and as each task has a different purpose, the result is that each task will have a different computational load. Thus, an application is said to be unbalanced if a significant number of computational nodes depend that others finish some kind of work, resulting in a waste of resources and energy [6].

Parallel applications like Molecular Dynamics , have load balancing (LB) as a crucial aspect to achieve a good performance in large scale parallel machines [9]. Adding to that, load balancing is critical to a different set of contexts and purposes, such as efficency in peer-to-peer network operations [10], task scheduling in Cloud Computing [11], energy cost reduction with geographic LB in Green Computing [12] and scientific adaptive simulations [13]. Therefore, in order to reach high performance, load balancing strategies must be applied in the context of such systems, since a load uneveness among hardware resources can lead to a low system utilization and impact the application performance negatively.

Selecting an efficient scheduling algorithm to achieve load balance in the context of applications that run on top of unpredictable environments is a non-trivial task. Load balancing can be needed in different moments of the application execution and for each one of these situations, it may require a different scheduling approach to achieve an optimal performance. An

efficient technique to scale these applications is to perform a redistribution of work loads, but undesirable effects may arise during this process [6], [14].

It is a challenge to manage the execution of a HPC application with the purpose to exploit at most the parallel platform resources, so that we can reach performance and scalability [5]. In this paper, we propose a Meta-Balancer that automates the process of load balancing algorithm selection for HPC applications. By using Machine Learning, it is possible to analyze the current state of the application and choose the best algorithm for that specific instant of time during runtime. Meta-Balancer is adaptative, which means it is oriented to the application load and state, collecting information dynamically and then invoking the most appropriate load balancing algorithm for that given context. Thus, Meta-Balancer reacts to the current system state in order to make migration decisions.

Since applications differ from one another, they require different balancers according to their specific needs. We have implemented Meta-Balancer on top of Charm++ Load Balancing Framework [15], which provides a variety of balancing strategies. In addition, the synthetic benchmark LBTest [7] available in Charm++ will be used to simulate applications with different parameters, such as network topology, number of elements and duration range of a task. We demonstrate that Meta-Balancer improves application performance by choosing different algorithms during an execution. Results show that Meta-Balancer performance gains reach 11.34% compared to the scenario where no load balancing occurs, and 2% compared to the second best load balancing strategy.

II. Motivation

Scientific applications, such as particle simulations, have unpredictable workloads. That is, they vary according to the computational flow. To achieve high performance in these classes of applications, the workload must be distributed dynamically among the processors. However, the requirements when choosing the load balancing algorithm vary according to the nature of each context. For instance, requirements needed for a particle simulation application will be different from those needed for an Adaptive Mesh Refinement (AMR) application [16].

Existing global scheduling algorithms used in load balancers manages the decomposed tasks by distributing them among the available physical resources in a parallel environment. The main point is that finding an optimal task distribution is a NP-Hard problem, and for this reason, algorithms that are based on different heuristics must be used for different scientific applications [17]. The algorithm selection criteria includes identifying a critical aspect that needs to be improved in the application, such as to reduce the communication between nodes within a distributed system or to minimize the maximum load on a node [18].

To illustrate this fact, we can have the situation where a scheduler has as its main goal the reduction of power consumption for a given platform, while another concentrates its focus on reducing the execution time of application iterations [19]. In such scenario, there is no algorithm that is ideal for all situations, since each scientific application is executed under different conditions, such as platform description, a goal that needs to be met, characteristics that make up the application, or other elements that have an influence on the application uniqueness.

In the case of an application that is characterized by having tasks with high load dynamics, there may be a need to migrate a larger number of tasks to achieve a balanced state. On the other side, the overhead caused by the task migration process may degrade performance in such a way that, the work redistribution can make the application slower than in the situation where migrations don't occur at all. In essence, performing load balancing involves time to find a new location for the task and time to move the task itself, but the resulting overhead from this process should be less than the gain obtained after load balancing [20]. Beyond that, a statically chosen scheduling algorithm can have a good performance at the beginning of an application execution, but later become inappropriate due to its unpredictable and dynamic behavior. [6].

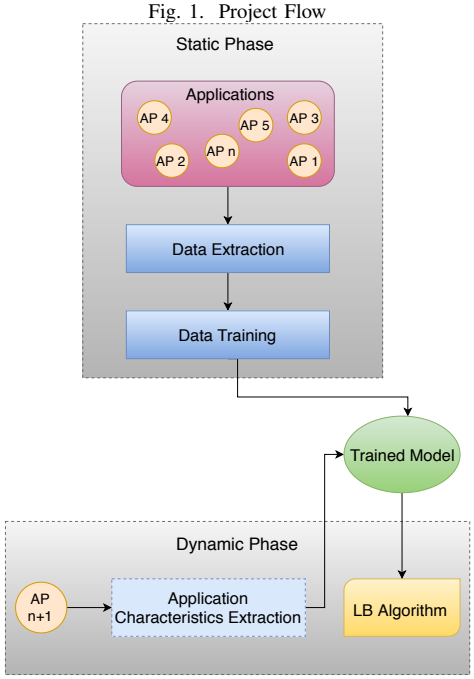
As previously seen, load balancing is a key factor that affects performance and scalability of parallel applications [21]. A HPC application with dynamic behavior tends to suffer changes in the behavior of its load, even if a load balancing has already been perfomed. Characteristics observed at a specific time, such as the computation rate, load imbalance or communication rate, interfere with the load behavior. Thus, an application that initially has its load unbalanced due to a high communication rate, may present another type of imbalance in the future, for instance, having its load unbalanced due to its computation. Going further, characteristics such as the average of object loads can impact the selection of the best load balancing strategy for that application. Thus, we note the need for a more intelligent entity to dynamically select load balancing strategies by performing an individual analysis for each scientific application in an adaptive way.

III. Machine Learning-based Meta-Balancer

A. Overview

Meta-Balancer is an entity that takes into account the different characteristics that make up the context where the application is being executed, whether they are the instant when the balancing occurs, aspects of the parallel platform of execution, or characteristics of the application itself. This gives us the ability to choose different heuristics at different times of the same application execution, according to the set of characteristics collected. Additionally, this approach avoids that load balancing decisions are taken manually by the user for each different type of HPC application. This manual process demands time and knowledge of the application context, since the user must estimate each task load, predict their behavior to decide which algorithm should be used and define the frequency that the balancing should occur.

Fig. 1. Project Flow



There are several benefits that Charm++ Parallel Framework offers in the context of this work, such as its adaptive runtime system (*Charm++ RTS*), overdecomposition, migrability and introspection. *Overdecomposition* allows the division of computation into independent units, resulting in data and work units that will be mapped to each processor element by Charm++ RTS [15]. This initial mapping can be changed during execution by migrating objects (*Charm++ chares*) to other PEs if the application is in an unbalanced state in terms of work load, exploiting the objects *migrability*. *Introspection* allows that the system collects information about the application during runtime, so it is possible to know the existing work load and make load balancing decisions.

In addition, the framework provides different LB heuristics, such as greedy and refinement. The greedy approach is characterized by moving the heaviest object to the least loaded processor, leading to a significant amount of communication. In contrast, refinement strategy migrates heavy objects from the most overloaded processors to the leasted loaded ones, generating less communication [22].

The Charm++ LB strategies used in this work are presented in table I. *GreedyLB*, *GreedyRefineLB* and *RefineLB* are centralized strategies commonly used in Charm++ and were chosen due to their stability. Thus, there is a greedy strategy, a refinement-based strategy and a combination of both, leading to a representative set of algorithms. Lastly, GreedyCommLB was selected because it is a LB strategy that uses the communication graph into account, and thus, being a good approach for cases where the application has a more significant communication load when compared to its computational load.

TABLE I
CHARM++ LB STRATEGIES

Algorithm	**Heuristic**	**Considers communication**	**Considers computation**
GreedyLB	Greedy	No	Yes
RefineLB	Iteratiea	No	Yes
GreedyCommLB	Greedy	Yes	Yes
GreedyRefineLB	Greedy	No	Yes

Figure 1 shows that the project flow is divided in a static phase and a dynamic phase. The main goal of the static phase is to obtain the Meta-Balancer, which is responsible for selecting the most appropriate LB approach for different types of applications. Once that goal is achieved, the dynamic phase focuses on validating the Meta-Balancer, that is, given an application that has not been seen during the previous phase, it will be analyzed at runtime and based on its characteristics, a LB strategy will be selected to that application.

B. Input Data Extraction

The first step to be taken in the static phase is to generate a set of synthetic applications using LBTest. This set should contain applications that vary in communication and computation load, leading to samples with characteristis and behaviors that differ from the one another. This first stage is essential so that the Meta-Balancer is well trained with any possible scenarios that may come up in the future. In a real situation, the selection of the best algorithm will only be successful if, during the training phase, there was an instance with characteristics that are similar to the ones of the real application being currently analyzed. Hence, we generate input data that contains a variety of parameter values to illustrate different scenarios of real applications that may arise at a later time.

The main goal of the *Data Extraction* stage is to obtain, for every instance of applications in the training set, the characteristics it is composed of and the best LB strategy for those characteristics. In other words, characteristics will become the features for the future machine learning model, and the algorithms will become their respective class/label, modeling a classification problem. Therefore, this phase gathers the data for training and validating the machine learning model.

For a given application, we want to know its characteristics at a given instant and which algorithm is the most appropriated one, given the data extracted from that instance. This is done by running this application *five times* for each one of the *four LB strategies* exposed in tableI and extracting their respective total execution time. It is necessary to run a scenario more than once to avoid any outlier value in the execution time. The results of these executions are combined to obtain the average execution time, and after that, it is possible to compare the values to find the LB strategy with the smallest execution time. That strategy is said to be the most appropriate one for that application being analyzed.

Once we have the best algorithm for each one of the applications, we need information about *how to represent* the application based on its characteristics, hence, we need the

application features. We implemented a *Profiler* to monitor the application at runtime with the purpose to collect relevant data about its load and behavior. This means that not only we need to define which type of information represents well an application, but the overhead to calculate this data must be small enough to keep the Meta-Balancer a benefit.

Charm++ RTS provides system load information by having a LB manager that resides on each processor, monitoring the CPU load, objects load, background load and other statistics, such as idle time. All of this information is stored in a LB database that is accessed during this phase, allowing the Meta-Balancer to use the stored data on the decision making process for the new mapping of objects to be performed [23]. This work focuses on centralized strategies, where there is a dedicated processor that collects global information about the state of the entire machine.

Table II shows the features selected. It is important to choose the right amount of information to represent an application, not only to avoid unnecessary overhead, but also to prevent an overfitting situation in our Machine Learning model. Hence, a study was made to discard ambiguous features and to keep the ones whose time to calculate their values does not degrade overall performance.

TABLE II
SELECTED FEATURES

Feature	**Description**
AvgPELoad	Processors average load
AvgOverloadedPEs	Average overloaded processors
LoadImbalance	(MaxPeLoad/AvgPeLoad)
AvgIdleTime	Average processors idle time
AvgObjLoad	Average object loads
CommCompRatio	Communication/Computation ratio
MsgOutsidePePercent	Messages sent to external processors (%)

For each feature, we run the Profiler *five times* and calculate the average value, similarly to the previous step. At this point, we have generated a set with a variety of synthetic applications, executed each one of them with the LB strategies present in Charm++ LB Framework, selected the LB algorithm that led to the smallest execution time and extracted the characteristics for each application. Combining this data, we have the set of features and classes to be used in our Machine Learning model.

C. Training

The final stage in the static phase is training our Machine Learning model. We have a set of applications that have been described by its features (characteristics) and by its class/label (the chosen algorithm). This fits perfectly in a classification problem for a supervised machine learning approach. Thus, given the input data generated in the last step, we implemented scripts that test that information against different machine learning algorithms.

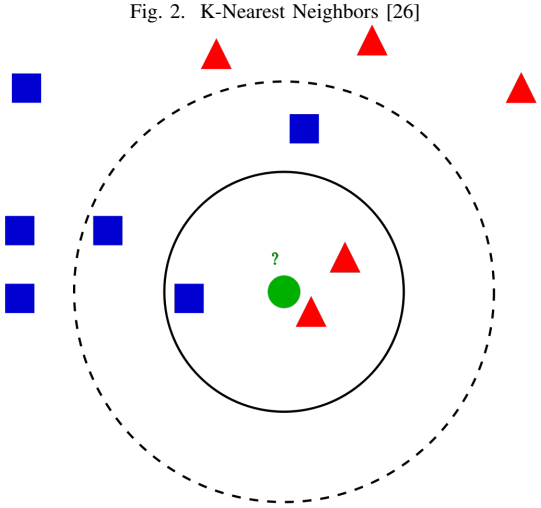
We used the *scikit-learn* library [24], available for the Python Programming Language. One advantage of using this library is the support for a variety of Machine Learning algorithms, such as Support Vector Machines, Random Forests, K-means, among others. Additionally, Python has become popular in the field of scientific computing due to its interative nature, support to data analysis and visualization and scientific libraries, such as NumPy, SciPy and Cython [25].

Given the data generated in the previous stage, we made an analysis based on the accuracy, classification report and confusion matrix for each classificator. A data normalization is performed, since the values from the features have a significant range variation. Finally, the input dataset is splitted into training set (70%) and validation set (30%), allowing to train the model with a significant amount of data and later, it is possible to validate the trained model to verify the accuracy of the model. The Machine Learning algorithms that were candidates are listed below.

- K-Nearest Neighbors Classifier (KNN)
- Support Vector Machines (SVM)
- Decision Tree Classifier
- Random Forest Classifier
- Ada Boost Classifier
- Gaussian Naive Bayes
- Linear Discriminant Analysis
- Quadratic Discriminant Analysis

Results shows that the best Machine Learning algorithm for the context of this work is the K-Neighbors Classifier, which performs the classification by computing the similarity based on the distance from the data to its closest neighbors. That is, the classificator verifies the distance from the data being tested to its neighbors, who have already been classified previously in the training phase. The data group (class) that presents the smallest distance between the training point and the testing point is then selected. Figure 2 illustrates this behavior.

Fig. 2. K-Nearest Neighbors [26]



Before moving to the dynamic phase, the model is trained with the input dataset and the trained result is saved in order

Algorithm 1: Invoking selected balancer

```
features ← profiler.extract_features();
selected_balancer ← predict_balancer(features);
if selected_balancer == GreedyLB then
    greedyLB → work()
else if selected_balancer == GreedyCommLB then
    greedyCommLB → work()
else if selected_balancer == GreedyRefineLB then
    greedyRefineLB → work()
else
    refineLB → work()
```

to be used in future predictions by the Meta-Balancer. This is possible by using the *pickle module* [27], which implements serialization and deserialization protocols for objects in Python. There, it becomes possible to store the trained model and retrieve it for the decision making process, at a later time.

D. Online Predictor

The last stage on the project flow is the dynamic phase, having as the final goal to predict load balancing algorithms for an application that has not been seen during the training stage. That is, given the trained model obtained in the static phase, new applications can be monitored during runtime, have their characteristics extracted at the load balancing instant and this information is sent to our model.

Based on the features collected at runtime from the new application, Meta-Scheduler is able to identify which algorithm is the most suitable one for the application and moment being analyzed. This is done by having a system call in the Meta-Balancer code in Charm++, which invokes Machine Learning scripts to deserialize the trained model, send the application characteristics and predict which balancer should be used for that context.

It is important to point that during this phase we can see one of the advantages of using Meta-Balancer, since this new approach is able to choose the load balancing algorithm dynamically. This means that, for a given application that has an unpredictable and irregular behavior, Meta-Balancer is able to switch between LB strategies at every load balancing period, if the switch is necessary. Listing shows 1.

```
Selected balancer: ['GreedyLB']
Selected balancer: ['GreedyCommLB']
Selected balancer: ['GreedyRefineLB']
Selected balancer: ['GreedyRefineLB']
Selected balancer: ['GreedyCommLB']
Selected balancer: ['GreedyRefineLB']
```

Listing 1. LB strategies selected by Meta-Balancer at every load balancing instant.

Once a balancer has been selected for a given application at a given instant of time, Meta-Balancer invokes the selected strategy by calling its *work()* function. In Charm++, every balancer present in its LB Framework implements a *work()* method, which is responsible for performing the migrations according to their respective heuristics. Algorithm 1 shows in practice how this process works, and in this case, GreedyLB was chosen for the first load balancing period, but in the second invocation of Meta-Balancer, it was seen that the application state changed, so a more appropriate balancer should be used for that given context.

Finally, it is necessary to validate and analyze the results obtained from different executions of new applications with Meta-Balancer. This involves experimentations and tests in a real parallel platform, which is going to be seen in the next section.

IV. Experimental Setup

A. Platform

Both static and dynamic phases of the project were implemented and tested in *Tesla* Machine from Distributed Systems Research Laboratory, located in Federal University of Santa Catarina. Tesla technical description is found below:

- Intel(R) Xeon(R) Processor CPU ES-2640 v4 @ 2.40GHz
- 10 physical cores with hyper-threading enabled
- Cache L1: 32 KB of data, 32 KB of instructions per core
- Cache L2: 256 KB per core
- Cache L3: 25 MB accessible in every CPU core
- 2 NUMA nodes (20 physical cores + hyperthreading)
- 128 GB RAM
- NVIDIA Tesla K40c

B. LBTest

LBTest is a synthetic benchmark implemented in Charm++ with the purpose to test load balancers. The advantage of its use is the flexibility in its parameters tuning, allowing to simulate a wide variety of computation and communication load distribution. The parameters include:

- Number of elements
- Number of iterations
- Load Balancing Frequency
- Network topology (ring, mesh2D, mesh3D)
- Print frequency
- Minimum duration time for a task
- Maximum duration time for a task
- Load Balancing algorithm

V. Experimental Results

In this section, we present a comparison of the Meta-Balancer performance with respect to four existing Load Balancing strategies in Charm++ presented in Table I. Additionally, our approach is also compared to the scenario where load balancing does not occur at all. We generated 12 new synthetic applications using LBTest, taking into account that these new instances have not been seen by the Meta-Balancer in the training stage.

In Figure 3, Meta-Balancer was able to obtain a gain of approximately 28.26 seconds compared to the case where no load balancing occurs. If compared to the second best time, which is GreedyLB, Meta-Balancer was 2.14 seconds faster.

Fig. 5. Experiment *Ring-3*

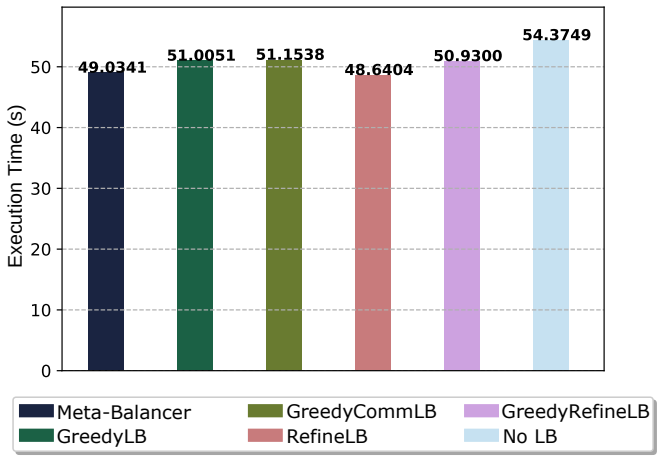


Fig. 6. Experiment *Mesh2D-2*

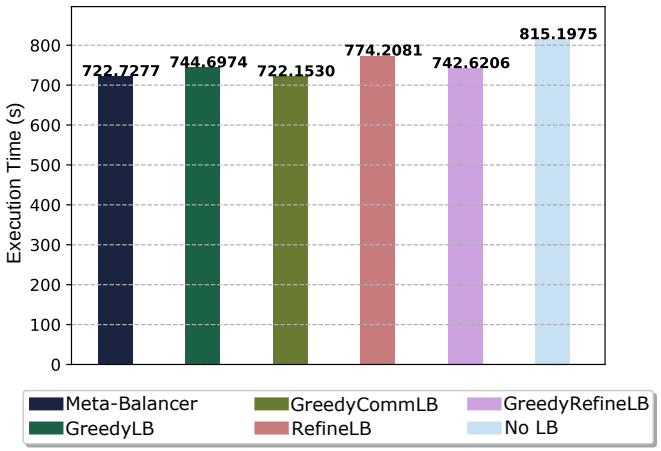
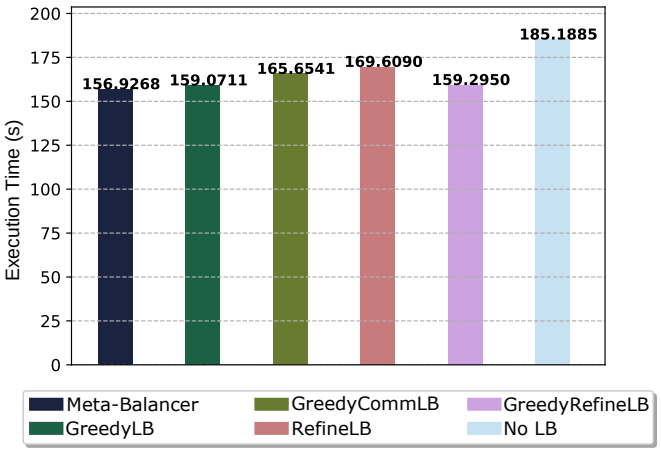
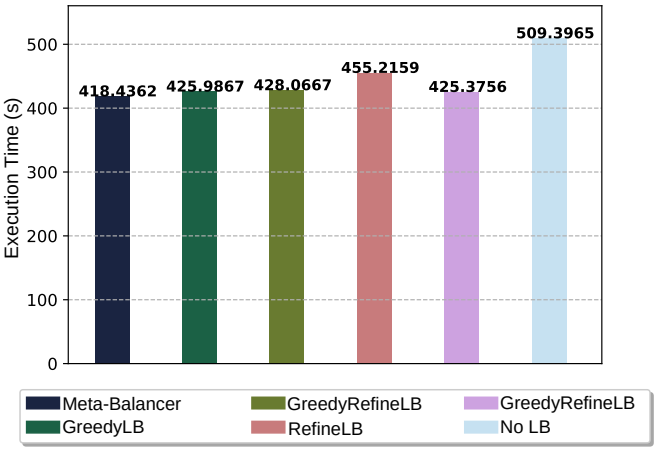


Fig. 3. Experiment *Mesh2D-4*



The second experiment shows that Meta-Balancer was able to obtain a gain of approximately 6.94 seconds compared to the second best, in this case, GreedyRefineLB. Furthermore, our approach is 90.96 seconds faster than the scenario where no load balancing occurs, as shown in Figure 4

Fig. 4. Experiment *Mesh2D-2*



In the third scenario, Figure 5 shows that Meta-Balancer was very close to the first placed balancer (RefineLB), having a time 0.39 seconds slower. But if compared to the context where the application does not perform load balancing, Meta-Balancer was able to obtain a gain of 5.34 seconds.

Finally, the last graph shown in Figure 6 shows the situation that Meta-Balancer was behind GreedyCommLB, having an execution time 0.58 seconds slower. However, in comparison to the scenario where there is no load balancing at all, our approach had a better performance, being 92.47 seconds faster.

Table III summarizes Meta-Balancer performance throughout the experiments made. The positive values represent the cases where Meta-Balancer was faster than the approach it is being compared to, and the negative values represent the cases where our strategy was slower. It is important to note that there is an *overhead* of 0.69 seconds due to the invocation of the trained Machine Learning model. That is, if that overhead is improved in future works, then Meta-Balancer would have the best performance in 11 out of the 12 experiments made.

An interesting scenario that is worth mentioning is Experiment *Mesh3D-4*. Note that Meta-Balancer was 7.36 seconds faster than the second best balancer and 6.12 seconds faster than the case where no load balancing occurs. In other words, if we were to compare only the four LB strategies from table I with the scenario without load balancing, then we see that using load balancing is *slower* than if we didn't use it at all. For the application being executed in that experiment, its behavior shows that using the *same* balancer throughout its *entire* lifetime doesn't present any benefits, since the cost of migrations made the overall performance worse. However, the fact that Meta-Balancer switched between LB strategies throughout the application execution was able to guarantee a better performance, since our approach has an adaptive nature and reacts to the current state of the application.

TABLE III
META-BALANCER OVERALL PERFORMANCE

Experiment	Performance Gain (wth LB) (s)	Performance Gain (No LB) (s)
ring-1	2.0278	21.3634
ring-2	0.0776	7.4856
ring-3	-0.3937	5.3408
ring-4	0.1118	0.2416
mesh2d-1	0.0007	0.8634
mesh2d-2	-0,5747	92.4698
mesh2d-3	0.7320	5.2299
mesh2d-4	2.1443	28.2617
mesh3d-1	-1.0290	19.3386
mesh3d-2	6.9394	90.9603
mesh3d-3	0.5797	10.8362
mesh3d-4	7.3626	6.1183

VI. RELATED WORK

Due to the unpredictable nature of many HPC applications, dynamic load balancing studies have been made througout the years [28], [29], [30], [31]. Charm++ [23] presents many benefits in its LB Framework, since it includes aspects like objects migratability, overdecomposition, introspection and adaptative runtime system [15]. There are works with different purposes that have exploited Charm++ advantadges [3], [32], [33], [34].

Load balancing is a key factor for scientific applications, since it can have a negative impact on the application performance [16], [12], [22]. As described in [17], finding an optimal task distribution is a NP-Hard problem, and therefore, different heuristics must be used for different applications.

Entities with an adaptive natures are proposed in [20], [35], [36], which present the concept of analyzing an application, gathering its characteristics and verifying if load balancing needs to occur at a given instant. Moreover, [22] also presents techniques for automating load balancing decisions. Machine learning can be used when it comes to adapt an entity to a given environment and [37] presents a similar concept, where a Profiler extracts information about input instances, a model is trained and a prediction can be made for a new input data.

The proposed work in this paper is completely adaptive, that is, Meta-Balancer is platform and application-independent. The selection of the ML model can also be modified, since there is a technique that presents the best ML algorithm for a given input dataset. This work does not focus on automating the decision for what *period* load balancing should occur, but to select the most appropriate LB strategy for a given application at a given instant of time.

VII. CONCLUSION

Load Balancing is a critical point that affects performance and scalability of parallel applications with different purposes. Selecting an efficient scheduling algorithm for applications that are unpredictable, irregular and dynamic is a non-trivial task. Additionally, choosing a LB strategy statically may present a good performance at an initial moment, but due to the application dynamic behavior, that strategy can become inappropriate throughout the execution. Besides that, a manual balancer selection means that the programmer shoud carefully study the application in order to make correct load balancing decisions. In this paper, we presented adaptative techniques for automating the LB strategy decision at runtime by using Machine Learning. Meta-Balancer is application independent, which means that it reacts to the current environment and responds with the best solution according to the environment recently analyzed.

Our ML-based approach involves a static phase to obtain our model and a dynamic phase to valide it. In our results, we profiled applications generated in LBTest Benchmark to collect information that becomes ML features. Besides that, we also run the same applications with four balancers available in Charm++ LB Framework in order to extract the smallest execution time, and hence, the best balancer for each application.

We demonstrated that Meta-Balancer was able to switch between LB strategies according to the needs that a given application presents at a given instant of time. We showed that Meta-Balancer presented advantages over the scenario where no load balancing occurs, even in the case that a statically balancer selection had a slower performance compared to the context without load balancing. We presented that Meta-Balancer had the best performance in most cases. In the cases where it was the second best strategy, it was approximately one second slower than the fastest approach, however, it is important to consider the overhead of 0.69 seconds originated from the trained model invocation. We also showed that our approach is able to improve performance in most cases and was capable of switching the LB strategy for the same execution of an application.

As future works, we intend to reduce the overhead caused by the ML model invocation, leading to a better performance. Additionally, it is desirable to improve the amount of training data in order to get a higher accuracy when training the model, that is, more data to represent different types of application scenarios. It is also important to improve the features selection, since these features are the core of an application representation.

REFERENCES

[1] A. De Marco, F. Nicolosi, D. Coiro, R. Tognaccini, F. Calise, P. Vecchia, S. Corcione, D. Ciliberti, and B. Mele, *High Performance Computing (HPC) and Aerospace Research Activities at the University of Naples Federico II*, 01 2017.

[2] B. Mannel, "HPE and NASA Increasingly Transform HPC and Space Exploration with Spaceborne Computer," 2018, https://www.hpcwire.com/solution_content/hpe/government-academia/hpe-and-nasa-increasingly-transform-hpc-and-space-exploration-with-spaceborne-computer/, accessed 08-07-2019.

[3] L. L. Pilla and E. Meneses, "Programação paralela em charm++," 2015.

[4] S. J. Kang, S. Y. Lee, and K. M. Lee, "Performance comparison of openmp, mpi, and mapreduce in practical problems," *Advances in Multimedia*, 2015.

[5] L. L. Pilla, "Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems," Ph.D. dissertation, Université de Grenoble, 2014.

[6] M. Rashid, I. Banicescu, and R. L. Carino, "Investigating a dynamic loop scheduling with reinforcement learning approach to load balancing in scientific applications," in *International Symposium on Parallel and Distributed Computing*, 2008.

[7] E. Meneses, L. V. Kale, and G. Bronevetsky, "Dynamic load balance for optimized message logging in fault tolerant hpc applications," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 281–289.

[8] C. B. Rodamilans, "Análise de desempenho de algoritmos de escalonamento de tarefas em grids computacionais usando simuladores," Master's thesis, Universidade de São Paulo, 2009.

[9] A. Bathele, L. Kale, and K. S., "Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications," in *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[10] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. ACM, 2004.

[11] B. L. D. Dhinesh and P. V. Krishna, "Honey bee behavior inspired load balancing of tasks in cloud computing environments," *Applied Soft Computing*, vol. 13, no. 5, 2013.

[12] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. H. Andrew, "Greening Geographical Load Balancing," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '11. ACM, 2011.

[13] K. Schloegel, G. Karypis, and V. Kumar, "A Unified Algorithm for Load-balancing Adaptive Scientific Simulations," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. IEEE Computer Society, 2000.

[14] V. M. C. T. Freitas and L. L. Pilla, "Comparando diferentes ordenações de tarefas em balanceamento de carga distribuido," in *XVII Escola Regional de Alto Desempenho*, 2017.

[15] A. Bilge, G. Abhishek, J. Nikhil, L. Akhil, M. Harshitha, M. Eric, N. Xiang, R. Michael, S. Yanhua, T. Ehsan, W. Lukasz, and K. Laxmikant, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[16] R. Chaube, R. Carino, and I. Banicescu, "Effectiveness of a Dynamic Load Balancing Library for Scientific Applications," in *Sixth International Symposium on Parallel and Distributed Computing*, 2007.

[17] J. Y. T. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman and Hall/CRC, 2004.

[18] D. Padua, *Encyclopedia of Parallel Computing*. Springer Publishing Company, Incorporated, 2011.

[19] L. Pilla, "Escalonamento Global Adaptativo para Aplicações Científicas," Feb 2017.

[20] H. Menon, N. Jain, G. Zheng, and L. Kalé, "Automated Load Balancing Invocation Based on Application Characteristics," in *IEEE International Conference on Cluster Computing*, 2012.

[21] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New Challenges in Dynamic Load Balancing," *Appl. Numer. Math.*, vol. 52, 2005.

[22] O. T. Pearce, "Load Balancing Scientific Applications," Ph.D. dissertation, Texas A&M University, 2014.

[23] P. P. Laboratory, "The Charm++ Parallel Programming System Manual," 2017, http://charm.cs.illinois.edu/manuals/html/charm++/, accessed on 30-04-2019.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] S. van der Walt, S. Colbert Chris, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science and Eng.*, vol. 13, pp. 22–30, 2011.

[26] C. c. Wikimedia, "File:KnnClassification.svg," 2016, https://commons.wikimedia.org/w/index.php?title=File:KnnClassification.svg&oldid=209850200, accessed on 25-05-2019.

[27] P. Documentation, "pickle - Python Object Serialization," 2019, https://docs.python.org/3/library/pickle.html, accessed on 29-04-2019.

[28] M. Bokhari, M. Alam, and F. Hasan, "Performance analysis of dynamic load balancing algorithm for multiprocessor interconnection network," *Perspectives in Science*, vol. 8, pp. 564–566, 2016.

[29] Y. Kaushik and C. K. Jha, "Performance Comparision of Dynamic Load Balancing Algorithm in Cloud Computing," *Int. J. Advanced Networking and Applications*, vol. 8, pp. 2986–2990, 2016.

[30] L. V. Kale, "Comparing the performance of two dynamic load distribution methods," *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 8–12, 1988.

[31] B. Acun and L. V. Kale, "Mitigating processor variation through dynamic load balancing," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[32] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Integrating openmp into the charm++ programming model," in *Third International Workshop on Extreme Scale Programming Models and Middleware*, 2017.

[33] J. Nikhil, B. Abhinav, Y. Jae-Seung, F. A. Mark, M. Francesco, M. Chao, and V. K. Laxmikant, "Charm++ and mpi: Combining the best of both worlds," in *IEEE 29th International Parallel and Distributed Processing Symposium*, 2015.

[34] N. Xiang, V. K. Laxmikant, and T. Rasmus, "Scalable asynchronous contact mechanics using charm++," in *IEEE 29th International Parallel and Distributed Processing Symposium*, 2015.

[35] H. Menon, "METABALANCER AUTOMATED LOAD BALANCING BASED ON APPLICATION CHARACTERISTICS," Master's thesis, University of Illinois at Urbana-Champaign, 2012.

[36] ——, "Adaptive load balancing for hpc applications," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2016.

[37] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *2011 18th International Conference on High Performance Computing*, 2011, pp. 1–10.