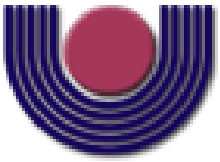


UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Soluções para o Balanceamento de Carga em Arquiteturas Paralelas Heterogêneas

Luis Fernando Veronese Trivelatto

CASCAVEL

2018

LUIS FERNANDO VERONESE TRIVELATTO

SOLUÇÕES PARA O BALANCEAMENTO DE CARGA EM ARQUITETURAS PARALELAS HETEROGÊNEAS

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada no dia 26/06/2018 pela Comissão formada pelos professores:

Prof. Guilherme Galante (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE – Campus de Cascavel

Prof. Rogério Luis Rizzi
Colegiado de Ciência da Computação,
UNIOESTE – Campus de Cascavel

Prof. Edmar André Bellorini
Colegiado de Ciência da Computação,
UNIOESTE – Campus de Cascavel

CASCAVEL

2018

“Inconformismo, insatisfação – sem isso, não se dá um passo à frente.”

Bernardinho

AGRADECIMENTOS

À minha família, em especial meus pais, Gilmar e Denise, pelo imensurável apoio ao longo não só deste trabalho ou do curso de graduação, mas de toda minha vida. Sem vocês, nada disso seria possível.

Aos professores que contribuíram para minha formação técnica e pessoal ao longo do curso, em particular: meu orientador Prof. Guilherme Galante, que me orientou desde o projeto de iniciação no primeiro ano até este trabalho de conclusão de curso, a despeito de eventuais lapsos do orientando; aos professores Clodis Boscarioli e Marcio Oyamada, que tive como tutores no PETComp, pelos aprendizados e risadas nas atividades desenvolvidas pelo grupo; e aos professores Josué Pereira de Castro e Adriana Postal, com quem pude compartilhar experiências em treinos, competições e viagens da Maratona de Programação.

Aos amigos e colegas adquiridos ao longo do curso, não só em sala de aula, como também no grupo PETComp, na Maratona de Programação, na atlética Leão Loko, e em outras atividades onde obtive experiências que certamente serão levadas para o resto da vida.

Ao Flamengo.

A todos que contribuíram direta ou indiretamente para minha formação dentro e fora da universidade.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
FPGA	*Field Programmable Gate Arrays*
GPU	*Graphics Processing Unit*
MIMD	*Multiple Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MPI	*Message Passing Interface*
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
RAP	*Resource Allocation Problem*
SIMD	*Single Instruction Multiple Data*
SISD	*Single Instruction Single Data*
SM	*Streaming Multiprocessors*
SPMD	*Single-program, multiple-data*
UP	Unidade de Processamento

Sumário

Resumo

Os sistemas de alto desempenho são cada vez mais construídos utilizando arquiteturas paralelas heterogêneas, incluindo a exploração de múltiplos níveis de paralelismo. No entanto, criar aplicações para arquiteturas heterogêneas otimizando o uso dos recursos computacionais é um desafio. Um dos fatores que influencia na complexidade de tal tarefa é o balanceamento de carga. Um balanceamento inadequado pode resultar em ociosidade dos processadores e baixo desempenho da aplicação. Desta forma, este trabalho abordou soluções para o balanceamento de carga em arquiteturas heterogêneas voltadas para aplicações paralelas iterativas, nas quais um processamento deve ser realizado repetidamente sobre um domínio, com cada iteração dependendo da anterior. Foi desenvolvida a biblioteca Multiframework Balance, que possibilita realizar um balanceamento de carga dinâmico em aplicações iterativas utilizando as ferramentas MPI e OpenMP, além de aceleradores. A biblioteca foi elaborada de forma a minimizar o esforço de programação para ser integrada a códigos já existentes, por vezes bastando a alteração de em torno de cinco linhas de código. Três problemas iterativos foram analisados como estudo de caso, e observou-se que a estratégia de balanceamento empregada na biblioteca apresentou resultados satisfatórios no que se refere à otimização do uso dos recursos computacionais. A taxa de utilização da arquitetura, calculada como uma relação entre tempo de trabalho e tempo de ociosidade dos processadores, foi elevada de 36% a 61%, nos testes sem balanceamento, para 89% a 98% nos testes com balanceamento. Também se obteve melhora no tempo de execução em todas as implementações com a biblioteca de balanceamento, com *speedup* relativo às versões não balanceadas variando entre 1,41 até 11,18, introduzindo-se um *overhead* negligível. Assim, conclui-se que realizar um balanceamento de carga adequado constitui um aspecto fundamental para obter desempenho de acordo com o potencial da arquitetura utilizada, principalmente em se tratando de arquiteturas heterogêneas.

Palavras-chave: programação paralela, arquiteturas heterogêneas, balanceamento de carga dinâmico, MPI, OpenMP, aceleradores, CUDA.

Capítulo 1

Introdução

A demanda por maior poder computacional tem motivado o desenvolvimento dos computadores desde suas origens. Mesmo com a tecnologia atual, muito superior àquela existente quando surgiram os primeiros sistemas computacionais, ainda há diversos problemas cuja execução requer tempo excessivamente longo em vista da alta demanda de computação. Alguns exemplos são modelagem climática, simulação de enovelamento de proteínas e desenvolvimento farmacêutico (PACHECO, 2011). Além disso, a tendência é que esta demanda continue aumentando, uma vez que a solução de um problema é frequentemente acompanhada pelo surgimento de outros ainda mais complexos (PACHECO, 1997).

Historicamente, os avanços tecnológicos nos processadores – como o aumento da frequência e número de *transistores* dos mesmos – fizeram com que a capacidade de processamento crescesse de forma exponencial. Entretanto, a existência de limitações físicas e tecnológicas no desenvolvimento de um único processador – tais como o consumo de energia, a dissipação de calor e o limite físico do tamanho de transistores – restringe a evolução do desempenho computacional por meio apenas do uso de processadores mais rápidos (BORKAR; CHIEN, 2011).

Desse modo, a computação paralela surgiu como solução para atender à necessidade por capacidade computacional cada vez mais elevada. De modo geral, as estratégias paralelas buscam diminuir o tempo de execução de um programa utilizando mais de um núcleo de processamento simultaneamente. Inicialmente, isto foi alcançado por meio da formação de *grids* e *clusters*, que combinam diversas máquinas em um único sistema. Posteriormente, observou-se o surgimento de processadores *multicore*, que incluem mais de um núcleo de processamento em um único processador (BINOTTO, 2011).

Por fim, na última década popularizou-se o uso de aceleradores para processamento paralelo, também chamados coprocessadores. Exemplos são as placas gráficas (GPUs) e os *Field Programmable Gate Arrays* (FPGAs). As GPUs foram inicialmente desenvolvidas para

processamento gráfico, porém sua arquitetura encontrou amplo uso no processamento de tarefas massivamente paralelas. Já os FPGAs são dispositivos de hardware reprogramáveis, podendo assim ser projetados para resolver problemas de modo eficiente (PINTO, 2011).

Os sistemas computacionais de alto desempenho atualmente têm sido construídos explorando diferentes níveis de paralelismo (MITTAL; VETTER, 2015). Pode-se, por exemplo, formar uma arquitetura híbrida composta por um *cluster* onde cada nó apresenta processadores *multicore* e aceleradores como a GPU (LU *et al*., 2012b). Ao passo que isso permite aumentar o poder de processamento disponível, paralelizar uma aplicação para um sistema heterogêneo passa a ser uma tarefa mais complexa (PINTO, 2011). Primeiro porque cada nível de paralelismo geralmente apresenta modelos de programação distintos. Assim, diferentes ferramentas são utilizadas para explorar o paralelismo, específicas para cada arquitetura. Por exemplo, ferramentas comumente utilizadas são o OpenMP, em arquiteturas *multicore*; o MPI, para comunicação em sistemas de memória distribuída (como *clusters*); e o CUDA, para criação de aplicações paralelas em GPUs (BARNEY, 2018).

Além disso, outra complexidade refere-se à otimização do uso dos recursos computacionais. As diferentes arquiteturas a serem exploradas podem apresentar distintas capacidades computacionais, não só pela própria natureza de cada arquitetura como também pela tecnologia de cada *hardware*. Isto significa que, em aplicações onde seja necessário algum tipo de sincronização entre as linhas paralelas de execução, pode-se observar gargalos e ociosidade das máquinas dependendo da distribuição da carga de trabalho (ACOSTA; BLANCO; ALMEIDA, 2013).

Um modelo de aplicação comum consiste em um domínio sobre o qual um processamento é realizado iterativamente, onde cada iteração depende da anterior. Neste tipo de aplicação, é possível extrair o paralelismo particionando-se o domínio a ser processado em diversos subdomínios, os quais são processados paralelamente; após isso, os dados são sincronizados entre as linhas de execução paralelas para permitir o processamento da próxima iteração. Este modelo iterativo aparece em diversos algoritmos paralelos, como método de Jacobi, algoritmos de programação dinâmica, problemas de caminho mínimo e simulações computacionais (ACOSTA; BLANCO; ALMEIDA, 2013).

Fica claro que, neste modelo de aplicação paralela, o desempenho do sistema como um todo é limitado pelo pior desempenho observado no processamento de um subdomínio, em função do ponto de sincronismo ao final da iteração. Nesse sentido, a otimização do uso dos

recursos computacionais depende de um balanceamento da carga de trabalho de cada unidade de processamento tendo em vista sua capacidade computacional.

A distribuição da computação em alto nível é comumente realizada pelo programador em tempo de programação, pré-definindo uma distribuição estática da carga de trabalho entre as unidades de processamento. Desse modo, as condições em tempo de execução não são consideradas neste processo. Isto gera uma limitação, uma vez que a distribuição de carga pode ficar desbalanceada por diversos fatores, como alterações na arquitetura paralela em que o programa será executado, desconhecimento por parte do programador do ambiente de execução ou distribuição manual sub-ótima, no caso de problemas onde a distribuição ótima de carga é não trivial (BINOTTO, 2011).

Dentro desse contexto, o uso dos recursos computacionais poderia ser otimizado caso a distribuição de carga ocorresse de forma dinâmica, em tempo de execução, por meio de uma ferramenta que avaliasse o ambiente computacional a ser explorado e escalonasse a carga de trabalho a ser destinada para cada unidade de processamento (BINOTTO, 2011).

Embora haja um direcionamento para diminuir a complexidade de programação em arquiteturas heterogêneas, ainda existe muito código legado que pode ser demasiadamente custoso para ser reescrito com as novas ferramentas (LU *et al*., 2012b). Por isso, também é relevante explorar soluções que permitem adaptar códigos já existentes a arquiteturas heterogêneas sem muitas alterações.

Assim, o objetivo deste trabalho foi apresentar uma solução para o balanceamento de carga em arquiteturas paralelas heterogêneas, particularmente que façam uso de múltiplas CPUs e aceleradores, considerando aplicações iterativas com particionamento da computação e sincronização em cada iteração. Os níveis de paralelismo explorados no trabalho e as ferramentas utilizadas foram: MPI para comunicação entre os nodos; OpenMP para criação de *threads* e exploração dos *cores*; e CUDA para exploração da GPU, embora a biblioteca desenvolvida não imponha restrições quanto ao uso de algum acelerador específico.

1.1 Objetivos

O objetivo geral do trabalho foi desenvolver uma biblioteca de balanceamento para realizar o balanceamento de carga em aplicações iterativas baseadas em particionamento da computação e sincronização ao fim de cada iteração. O trabalho foi desenvolvido tendo em

mente a execução de programas em arquiteturas paralelas heterogêneas, embora a biblioteca possa ser utilizada em arquiteturas homogêneas também.

Como objetivos específicos, tem-se:

1. Implementar a solução na forma de uma biblioteca visando minimizar o custo de programação para integrá-la a códigos existentes.

2. Possibilitar a exploração de múltiplos níveis de paralelismo, incluindo múltiplos *cores* e aceleradores.

3. Avaliar a diferença de desempenho entre implementações de estudos de caso com e sem a biblioteca.

1.2 Contribuições

Neste trabalho, foi desenvolvida a biblioteca Multiframework Balance, que permite realizar o balanceamento de carga em aplicações paralelas iterativas desenvolvidas com as ferramentas MPI, OpenMP e aceleradores. O código está disponível no GitHub, no endereço https://github.com/luistrivelatto/TCC. A biblioteca é uma extensão do trabalho desenvolvido por (ACOSTA; BLANCO; ALMEIDA, 2013). No artigo original, os autores desenvolveram a biblioteca *ULL_MPI_calibrate*, que realiza o balanceamento apenas no nível dos processos MPI.

A biblioteca Multiframework Balance foi desenvolvida tendo em mente um baixo custo para integração a códigos já existentes; para tal, buscou-se elaborar uma interface de programação simples e que reaproveitasse elementos comuns de aplicações paralelas, de forma a minimizar a intrusão de código. A biblioteca requer a adição de 4 métodos ao código: inicialização e finalização da biblioteca, e inicialização e finalização do trecho crítico a ser balanceado. Nos Capítulos 4 e 5 são dados exemplos de códigos nos quais, além de adicionar estas 4 chamadas de funções, basta alterar outras duas linhas de código para utilizar a biblioteca.

O algoritmo de balanceamento apresentado foi validado por meio das implementações nos estudos de caso, que atingiram resultados satisfatórios no que se refere à otimização do uso dos recursos computacionais. Foi calculada a taxa de utilização dos recursos computacionais, relacionando o tempo de trabalho e o tempo de ociosidade dos processadores, e observou-se que tal taxa subiu de 36% a 61%, nos testes sem balanceamento, para 89% a 98% nos testes com balanceamento. Registrou-se melhora no tempo de execução em todas as

implementações com a biblioteca de balanceamento, com *speedup* relativo às versões não balanceadas variando entre 1,41 até 11,18. A biblioteca apresentou *overhead* de execução baixo, entre 1 e 2 milissegundos por iteração.

1.3 Organização do Texto

Este trabalho está organizado da seguinte maneira: no Capítulo 2 são apresentados fundamentos da computação paralela, detalhando as arquiteturas e ferramentas de programação que foram exploradas no trabalho, e discutindo fatores relacionados à programação em arquiteturas heterogêneas. No Capítulo 3 são discutidos os problemas de balanceamento e escalonamento de carga, e é feita uma revisão de literatura acerca de trabalhos nessa área. No Capítulo 4, é apresentada a biblioteca de balanceamento desenvolvida, detalhando a classe de aplicações paralelas à qual a biblioteca é voltada, apresentando a interface de programação e a estratégia de balanceamento empregada. No Capítulo 5, são detalhados os problemas que foram utilizados como estudos de caso, ilustrando implementações paralelas com o uso da biblioteca. No Capítulo 6, são apresentados e discutidos os resultados obtidos na execução das implementações dos estudos de caso e, por fim, no Capítulo 7 apresentam-se as conclusões do trabalho.

Capítulo 2

Computação Paralela

2.1 Arquiteturas Paralelas

Historicamente, a evolução da capacidade de processamento dos computadores esteve relacionada ao desenvolvimento de processadores mais rápidos, principalmente em função do aumento da frequência de *clock* dos mesmos e do aumento do número de transistores em um chip na medida em que estes se tornavam menores e mais rápidos. Entretanto, a partir da primeira década do século XXI, limites físicos e tecnológicos para a evolução dos processadores com estas técnicas começaram a ser atingidos, como o alto consumo de energia, a dissipação de calor e o limite físico do tamanho de transistores (BORKAR; CHIEN, 2011).

Assim, o crescimento da capacidade computacional passou a ser obtido por meio da computação paralela. De modo geral, a execução de um programa em um computador consiste em realizar uma sequência de instruções sobre um conjunto de dados. Basicamente, as estratégias paralelas buscam diminuir o tempo de execução de um programa utilizando mais de um núcleo de processamento simultaneamente – ou seja, realizando mais de uma instrução simultaneamente ou operando sobre múltiplos dados concorrentemente (PACHECO, 2011).

A partir disso, é possível classificar os computadores quanto ao paralelismo sobre o fluxo de instruções e o fluxo de dados. Essa ideia originou a Taxonomia de Flynn, proposta em 1966 e que ainda hoje é uma das principais maneiras de se classificar sistemas paralelos (SCHEPKE, 2009). Ela divide os computadores em quatro classes:

a) *Single Instruction Single Data* (SISD) – nesta classe, uma instrução opera sobre um único dado; logo, não há paralelismo presente. Os computadores com um único processador se encontram nesta classe;

b) *Multiple Instruction Single Data* (MISD) – aqui, várias instruções operam sobre um único dado. Esta é uma classificação com quase nenhum uso prático;
c) *Single Instruction Multiple Data* (SIMD) – neste caso, uma única instrução é executada simultaneamente em diversos dados. As unidades de processamento gráfico (*Graphical Processing Units* - GPUs) utilizam esta estratégia para obter paralelismo. As GPUs são descrita em mais detalhes na Seção 2.1.3;
d) *Multiple Instruction Multiple Data* (MIMD) – por fim, esta categoria engloba as máquinas em que múltiplas instruções são executadas paralelamente sobre diferentes dados. Este seria o caso de processadores *multi-core*, *grids* e *clusters*, descritos em mais detalhes nas Seções 2.1.1 e 2.1.2.

2.1.1 *Multi-core*

Os processadores *multi-core* consistem em múltiplos núcleos de processamento completos agrupados em um único *chip*, conectados a uma mesma memória. Cada núcleo pode ser visto como um processador lógico independente, com recursos próprios, de forma que cada um pode executar aplicações distintas; por isso, na Taxonomia de Flynn, os processadores *multi-core* são classificados como sistemas MIMD. Do ponto de vista da programação paralela, é possível executar uma aplicação paralelamente nos diversos núcleos.

Uma vez que estão conectados a uma mesma memória, os processadores *multi-core* são considerados um sistema de memória compartilhada. Nestes sistemas, há um mesmo espaço de memória ao qual os simultâneos fluxos de execução de uma aplicação paralela têm acesso; assim, caso uma tarefa atualize um dado na memória global, a alteração já estará visível para as demais tarefas.

De modo geral, sistemas de memória compartilhada têm escalabilidade limitada em função de restrições físicas: por exemplo, cada núcleo de processamento necessita acesso à memória, gerando um gargalo no sistema. Um método utilizado para minimizar este efeito é que cada núcleo possua uma memória *cache* local. Entretanto, isso resolve apenas parcialmente o problema, uma vez que gera problemas de coerência de *cache*: se um núcleo atualiza um dado em sua *cache*, é necessário replicar esta atualização na *cache* dos outros núcleos, caso o dado esteja compartilhado com as mesmas, para que se garanta a integridade dos dados da *cache* (SCHEPKE, 2009).

2.1.2 Multicomputadores

Os sistemas multicomputadores basicamente são computadores independentes interconectados por uma rede. Cada computador tem sua própria unidade de processamento e sua memória, sendo que um computador não consegue acessar a memória de outro, ou seja, não há uma memória global. Por isso, os sistemas multicomputadores são considerados sistemas de memória distribuída, e também classificados na categoria MIMD. Caso diferentes computadores no sistema queiram trocar dados, é necessário utilizar troca de mensagens por meio da rede que os conecta.

Estes sistemas são mais flexíveis e escalonáveis que os sistemas de memória compartilhada, uma vez que os processadores estão conectados por uma rede – e não no nível de um *chip*. Assim, é possível adicionar e remover máquinas sem interferir no restante do conjunto. Naturalmente, não há gargalo no acesso à memória porque não há uma memória global. Entretanto, compartilhar dados passa a ser uma tarefa mais custosa, uma vez que requer troca de mensagens entre os processos; além disso, há um aumento na complexidade de programação, uma vez que as operações de sincronização e troca de mensagens passam a ser responsabilidade do programador (PACHECO, 2011).

Exemplos de sistemas de memória distribuída são os *clusters* e *grids*, nos quais as múltiplas tarefas são executadas por processos nas diferentes máquinas. Basicamente, *clusters* são multicomputadores mantidos em um mesmo ambiente; desta forma, geralmente são compostos por máquinas homogêneas. Já os *grids* tem sua infraestrutura distribuída em dimensões maiores, por vezes atingindo proporções globais. Um *grid* pode, por exemplo, ser composto por uma série de *clusters* de configurações diferentes utilizando a internet como rede de interconexão (PACHECO, 2011).

2.1.3 Aceleradores

Os aceleradores, ou coprocessadores, têm se popularizado nas arquiteturas paralelas na última década (PINTO, 2011). Eles são tipicamente especializados em um tipo específico de computação, podendo obter desempenho muito superior aos processadores tradicionais nestes problemas. Dois dos principais coprocessadores são os *Field Programmable Gate Arrays* (FPGAs) e as GPUs. Os FPGAs são dispositivos de hardware reprogramáveis, podendo assim serem projetados para resolver problemas de modo eficiente.

As GPUs foram inicialmente desenvolvidas para processamento gráfico, porém sua arquitetura encontrou amplo uso no processamento de tarefas massivamente paralelas. Por serem arquiteturas vetoriais, em que uma mesma instrução é executada sobre diversos dados, as GPUs se encaixam na categoria SIMD da Taxonomia de Flynn.

Ao passo que os sistemas MIMD executam em paralelo processadores complexos, o que resulta em um tempo de latência – tempo para executar uma instrução – baixo, as GPUs focam em aumentar o *throughput*, que corresponde à taxa de execução de instruções por unidade de tempo. Por isso, as GPUs são processadores massivamente paralelos, contendo centenas ou até milhares de unidades de processamento (ao passo que os processadores *multi-core*, por exemplo, raramente apresentam mais de algumas dezenas de cores), mais simples que os processadores apresentados pelos sistemas MIMD. Isto significa que o tempo de latência não é tão bom como nos processadores dos sistemas MIMD, mas, uma vez que há um número considerável de unidades de processamento, esta deficiência acaba superada pelo ganho em *throughput* (GARLAND; KIRK, 2010).

Deste modo, as unidades de processamento de uma GPU são especializadas nas chamadas operações vetoriais – executar uma mesma instrução sobre não um, mas um conjunto de dados ao mesmo tempo, explorando o paralelismo de dados. Para somar dois vetores de oito elementos, por exemplo, um processador tradicional teria que realizar oito adições em sequência; já com operações vetoriais, as oito adições podem ser executadas concorrentemente, aproveitando-se do fato de que a instrução a ser executada pelos núcleos de processamento é a mesma – somar dois valores – alterando-se apenas os dados sobre os quais essa instrução será executada. Assim, as GPUs destinam mais transistores para operações lógicas e aritméticas e menos transistores para operações de controle de fluxo (SILVA, 2014).

2.2 Ferramentas

Ferramentas que buscam criar uma camada de abstração para unificar a exploração dos vários níveis de paralelismo foram propostas, mas sem alcançar eficiência máxima em função dos diferentes modelos de programação observados em diferentes arquiteturas paralelas. Assim, as ferramentas mais utilizadas para criação de aplicações paralelas focam em um único nível de paralelismo e modelo de programação (MAILLARD; SCHEPKE, 2008). A

seguir, estão descritas três ferramentas popularmente utilizadas para programação nas diferentes arquiteturas paralelas: o OpenMP, o MPI e o CUDA.

2.2.1 OpenMP

O OpenMP (*Open Multi-Processing*) é uma API que viabiliza a criação de aplicações paralelas em sistemas de memória compartilhada. Esta extensão para as linguagens sequenciais C, C++ e Fortran consiste em um conjunto de diretivas de compilação, funções de bibliotecas e variáveis de ambiente que determinam o comportamento de uma aplicação paralela em tempo de execução.

O OpenMP trabalha sobre o modelo *Fork-Join*, um paradigma para aplicações paralelas no qual uma *thread* principal (chamada "*master thread*") executa trechos de código sequencialmente e, em determinados pontos do programa, ramifica a execução ("*Fork*") em diversas *threads*, cada qual executando um trecho paralelo do programa. Ao final deste trecho, as *threads* são fundidas ("*Join*") novamente ao processo principal, que retoma a execução sequencial (CHAPMAN; JOST; PAS, 2008).

2.2.2 MPI

O MPI (*Message Passing Interface*) é um padrão para comunicação entre processos por meio da troca de mensagens que permite a criação de aplicações paralelas em sistemas de memória distribuída. O MPI define um conjunto de funções que cobrem os mecanismos envolvidos na comunicação entre processos, desde primitivas para comunicação entre os processos até a criação de tipos de dados derivados para serem utilizados na troca de mensagens. A comunicação mais básica presente no padrão é a comunicação ponto a ponto, envolvendo a troca de mensagens entre dois processos específicos. A partir da comunicação ponto a ponto, são construídos outros tipos de comunicação mais complexos, permitindo a troca de dados por um número maior de processos, chamados comunicações coletivas.

Diferentemente do *OpenMP*, que opera sobre o modelo *Fork-Join*, a maioria das implementações do MPI utilizam um modelo no qual um conjunto fixo de processos é criado ao início da execução do programa, cada qual com um identificador. Estes processos executam o mesmo programa, mas por vezes executam instruções diferentes devido a operações condicionais de controle de execução, as quais tipicamente são baseadas no identificador do processo. Esta técnica é chamada de "Programa único, múltiplos dados" (do

inglês SPMD – *single-program, multiple-data*), sendo uma das abordagens mais comuns para a criação de aplicações paralelas (PACHECO, 1997), (SNIR *et al*., 1998).

2.2.3 CUDA

O CUDA (*Compute Unified Device Architecture*) é um ambiente de desenvolvimento criado pela NVIDIA para criação de aplicações paralelas em GPUs. Esta ferramenta permite que o programador defina funções especiais, chamadas *kernels*, as quais são executadas em paralelo em *threads* na GPU. O fluxo do programa é controlado pela CPU, que executa trechos de código sequencialmente. O paralelismo se dá quando a CPU ativa a GPU realizando uma chamada de *kernel*. As *threads* são organizadas em conjuntos lógicos de 1, 2 ou 3 dimensões chamados blocos de *threads*. Os blocos, por sua vez, estão organizados em um *grid* lógico de 1, 2 ou 3 dimensões. Cada bloco e cada *thread* possui um identificador único que distingue sua posição nas coordenadas *x*, *y* e *z* dentro do *grid*/bloco, respectivamente.

Em nível de *hardware*, as GPUs da NVIDIA são compostas por um conjunto de blocos físicos chamados *streaming multiprocessors* (SMs), os quais são compostos por um conjunto de núcleos de processamento. Por exemplo, a arquitetura de uma GPU Pascal GeForce GTX 1080 é composta por 20 SMs, cada qual contendo 128 núcleos de processamento, combinando para um total de 2560 CUDA *cores* (NVIDIA, 2016). Para que uma *kernel* seja executada na GPU, os blocos lógicos de *threads* definidos pelo programador são divididos em conjuntos com um tamanho fixo de *threads*, cada qual é alocado para um SM. Todos os elementos de processamento da SM executam a mesma instrução simultaneamente, porém sobre dados diferentes, ou seja, explorando o paralelismo SIMD. Entretanto, é possível que as *threads* sigam diferentes caminhos de fluxo e executem instruções diferentes, o que é chamado de divergência de código; nesse caso, a execução das *threads* no SM é serializada, causando perda na eficiência do paralelismo. Fica claro, então, que o modelo de programação CUDA obtém maior desempenho paralelo para programas que apresentam baixa necessidade de controle de fluxo em relação ao número de operações lógicas e aritméticas (SILVA, 2014).

2.3 Arquiteturas Heterogêneas

As diferentes arquiteturas paralelas apresentadas significam múltiplos níveis de paralelismo a serem explorados. Desta forma, os sistemas computacionais de alto

desempenho atualmente têm sido construídos combinando estas arquiteturas de forma a explorar os diferentes níveis de paralelismo (MITTAL; VETTER, 2015).

O desempenho destes sistemas é condicionado pela dependência existente entre a programação paralela e a arquitetura utilizada, uma vez que cada arquitetura apresenta um modelo de programação diferente. Ainda, é possível haver sistemas paralelos em que os diferentes nodos de computação apresentam diferentes capacidades computacionais. Isso significa que, em arquiteturas heterogêneas, há uma complexidade maior não só para criar aplicações paralelas como também para otimizá-las de forma a melhorar a exploração da capacidade computacional do sistema.

Uma maneira de criar aplicações paralelas que exploram mais de um nível de paralelismo é simplesmente combinar as diferentes ferramentas utilizadas em cada modelo de programação. Por exemplo, pode-se combinar as ferramentas apresentadas neste capítulo para explorar o paralelismo multinível em um *cluster* onde cada nó contém processadores *multi-core* e uma GPU. Assim, pode-se utilizar MPI para realizar a comunicação entre cada nó e OpenMP/CUDA para distribuir a computação em cada nó entre seus *cores*/GPU, respectivamente (LU *et al*., 2012a), (BARNEY, 2018).

Essa abordagem torna mais complexa a programação destas aplicações, pois não há uma camada de abstração que torne homogênea a exploração de cada arquitetura. O programador deve utilizar as diferentes ferramentas bem como determinar manualmente a divisão de trabalho, troca de dados, sincronização e qualquer outro gerenciamento necessário nos diferentes níveis sendo explorados.

Dessa forma, têm sido propostas ferramentas que buscam simplificar a programação em sistemas heterogêneas criando tal camada de abstração. Estas ferramentas podem ser frameworks, como o OpenCL (KHRONOS, 2017), ambientes de execução, como o StarPU (AUGONNET *et al*., 2009), e até linguagens de programação completas, como o Titanium (YELICK *et al*., 1998). Em comum, elas buscam abstrair do programador a necessidade de pensar especificamente em cada arquitetura a ser explorada, o que permite também escrever aplicações com maior portabilidade entre sistemas heterogêneos diferentes. Algumas ferramentas incluem ainda automatização de transferência de dados entre os recursos de processamento, escalonamento das tarefas para os recursos, entre outros.

No entanto, tais ferramentas também apresentam limitações. Algumas podem operar sobre modelos específicos de aplicações paralelas, nos quais uma dada aplicação poderia não se

encaixar. Além disso, pode haver perda de desempenho da aplicação, uma vez que aspectos como o escalonamento de tarefas e troca de dados automatizados pela ferramenta talvez pudessem ser melhor otimizados pelo programador. Neste caso, isso representaria um *trade-off* entre desempenho e complexidade de programação. Por fim, deve-se considerar que há muito código já existente escrito sem essas ferramentas, ou seja, combinando os métodos de programação específicos a cada arquitetura, e reescrever estes códigos com novos frameworks como OpenCL ou mesmo adaptá-los para utilizar tais ferramentas pode ser uma tarefa complexa e custosa (LU *et al*., 2012b). Por isso, mesmo com o desenvolvimento destas ferramentas, é relevante estudar modos de simplificar a programação e melhorar o desempenho de aplicações desenvolvidas sem esta facilidade.

Neste sentido, um dos fatores que cria complexidade ao programar aplicações com gerenciamento manual dos diferentes níveis de paralelismo é a necessidade de realizar o escalonamento das tarefas e balanceamento de carga entre os recursos de computação disponíveis. Um mau escalonamento pode fazer com que uma tarefa seja mapeada para uma unidade de processamento não adaptada para a natureza da tarefa, causando baixo desempenho. Um exemplo é uma tarefa com alta necessidade de controle de fluxo sendo executada em uma GPU; se houver divergência de código, a execução das *threads* é serializada, causando significativa redução de desempenho (SILVA, 2014). Ressalta-se que, não raramente, a diferença de desempenho de uma tarefa nas diferentes unidades de processamento não é significativa (além da diferença de capacidade das unidades), de forma que é possível considerarmos que o mapeamento pode ser feito para qualquer unidade.

Já um balanceamento inadequado pode causar perda de desempenho por causar ociosidade das unidades de processamento. Por exemplo, considere um sistema paralelo consistindo em uma CPU e uma GPU, de forma que a capacidade computacional é de 1 tarefa processada por segundo pela CPU e 1000 tarefas processadas por segundo pela GPU. Se deseja-se processar 2000 tarefas e o balanceamento for feito de forma uniforme, isto é, 1000 tarefas para a CPU e 1000 tarefas para a GPU, o tempo total de execução será de 1000 segundos, com a GPU ficando ociosa por 999 segundos (TSE *et al*., 2010). Este problema pode ser acentuado caso haja necessidade de sincronização das linhas de execução paralelas em certo ponto do processamento: todas as linhas de execução terão que aguardar pela última a chegar ao ponto de sincronização, observando-se um gargalo e ociosidade das demais máquinas. A Figura 2.1 ilustra um sistema balanceado e outro desbalanceado.

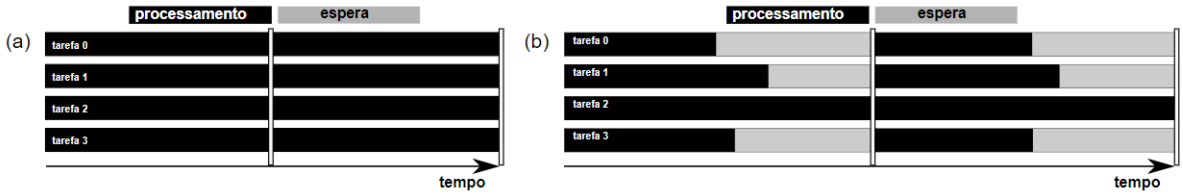


Figura 2.1: Carga balanceada (esquerda) e desbalanceamento de carga (direita)

Assim, ao se trabalhar com arquiteturas heterogêneas, é necessário observar todos esses possíveis problemas. No próximo capítulo apresenta-se um conjunto de trabalhos que objetivam tratar os problemas de escalonamento e balanceamento de carga a fim de buscar o melhor desempenho para a aplicação.

Capítulo 3

Escalonamento e Balanceamento de Carga

3.1 Escalonamento e Balanceamento

O escalonamento e balanceamento da carga de trabalho desempenham um papel chave na execução eficiente de uma aplicação paralela, seja em sistemas homogêneos ou heterogêneos. Em sistemas heterogêneos, em particular, determinar bons escalonamento e balanceamento frequentemente apresenta desafios maiores em relação a esta tarefa para sistemas homogêneos, justamente em função das diferenças entre as unidades de processamento do sistema. Além disso, o custo de execução de uma tarefa em uma unidade de processamento (UP) é não-determinístico e pode ser afetado por outros parâmetros desconhecidos a *priori*, como o tamanho do domínio do problema, a precisão desejada da solução, a ocupação de uma UP com outras aplicações, entre outros (BINOTTO, 2011).

Por isso, o escalonamento e balanceamento de carga são tópicos com considerável estudo entre as técnicas de computação heterogênea (MITTAL; VETTER, 2015). Para explorar ao máximo os recursos computacionais disponíveis, a carga de trabalho deve ser distribuída entre as unidades de processamento de acordo com suas capacidades computacionais. Isso significa tanto particionar as tarefas ou os dados a serem processados (balanceamento) como também determinar em qual UP um dado processamento será realizado (escalonamento).

A determinação do escalonamento pode ser definida de acordo com algum critério específico desejado pelo programador, motivado por características da UP e/ou das tarefas a serem executadas. Um dos critérios mais comuns é, quando as tarefas apresentam natureza comum e podem ser executadas em qualquer UP, simplesmente considerar a performance em cada uma e o balanceamento, visando o melhor tempo de execução da aplicação. Já no caso em que as tarefas apresentam diferenças, pode-se preferir por um escalonamento que mapeie uma dada tarefa para a UP que mais se adapta para executá-la; por exemplo, tarefas que

apresentam alto paralelismo de dados poderiam ser mapeadas para uma GPU, enquanto tarefas que apresentam maior necessidade de controle de fluxo poderiam ser mapeadas para uma CPU. Ainda, é possível que o escalonamento seja realizado considerando que uma tarefa não possa ser mapeada para uma UP específica, como no caso em que a memória necessária para uma tarefa excede o tamanho da memória de uma GPU, ou que seja realizado levando-se em conta a precisão de ponto flutuante das diferentes UPs (MITTAL; VETTER, 2015).

Em relação ao balanceamento, há diferentes abordagens para determinar como a carga de trabalho será distribuída entre as unidades de processamento (MITTAL; VETTER, 2015). De modo geral, estas técnicas podem ser classificadas como estáticas ou dinâmicas. Na divisão estática, o mapeamento das tarefas para as UPs em que serão executadas é predefinido, ou seja, ele é fixo durante a execução. Para isso, o tempo que será necessário para processar uma tarefa ou um conjunto de dados em uma UP é estimado pelo programador ou por uma ferramenta. Tal estimativa pode ser baseada na capacidade teórica de computação das unidades de processamento, em parâmetros em tempo de compilação, ou em um período de treinamento sobre uma fração dos dados de entrada. Desta forma, as condições em tempo de execução não são consideradas no balanceamento. Isto gera uma limitação, uma vez que pode haver desbalanceamento de carga em razão de diversos fatores, como alterações na arquitetura paralela em que o programa será executado, desconhecimento por parte do programador acerca do ambiente de execução ou distribuição manual sub-ótima (BINOTTO, 2011). Portanto, técnicas baseadas em particionamento estático não consideram as dinâmicas presentes em tempo de execução e a arquitetura na qual a aplicação é executada (BELVIRANLI; BHUYAN; GUPTA, 2013).

Já no particionamento dinâmico, a decisão sobre qual tarefa é executada em qual UP ou sobre quais dados são processados por qual UP é tomada em tempo de execução. Isso permite realizar um balanceamento de carga mais preciso. Para (BELVIRANLI; BHUYAN; GUPTA, 2013), um esquema ideal de balanceamento dinâmico deve levar em consideração dois fatores críticos: (1) medição precisa da capacidade computacional das unidades de processamento e (2) tamanho ideal das tarefas/conjunto de dados a serem processados por um acelerador. O ponto (2) é ressaltado em função do *overhead* de inicialização e transferência dos dados para o acelerador; se poucos dados serão processados, estes fatores serão preponderantes e ocuparão a maior parte do tempo de execução. Por outro lado, se muitos dados serão processados, pode ocorrer o desbalanceamento de carga. Em relação ao ponto (1), ressalta-se

que, embora (BELVIRANLI; BHUYAN; GUPTA, 2013) mencione a medição das unidades de processamento, pode haver esquemas de balanceamento sem medição explícita, onde a distribuição da carga de trabalho ocorre naturalmente pela natureza do processo adotado. Este é o caso, por exemplo, em políticas de balanceamento baseadas em fila de tarefas (CHEN *et al*., 2010), roubo de tarefas (AUGONNET *et al*., 2009) ou incremento iterativo na carga de trabalho das UPs (TSE *et al*., 2010). Estes e outros trabalhos são descritos com mais detalhes a seguir.

3.2 Trabalhos Correlatos

Esta seção apresenta os principais trabalhos na área de escalonamento e balanceamento relacionados à proposta desta monografia, ou seja, com foco em ferramentas ou métodos que realizem o balanceamento de carga preferencialmente de forma dinâmica, em arquiteturas heterogêneas com múltiplos níveis de paralelismo, possibilitando adaptação a códigos já existentes, e particularmente aplicáveis a modelos de aplicação baseados em decomposição de dados e sincronização iterativa.

(CHEN *et al*., 2010) apresenta um sistema de balanceamento dinâmico baseado em tarefas para sistemas com uma ou múltiplas GPUs, utilizando CUDA. Os autores explicam que, no paradigma de programação apresentado pelo CUDA, para executar várias tarefas diferentes na GPU, o *host* deve lançar várias *kernels*, cada qual com um *overhead* de inicialização. A principal ideia do trabalho é, ao invés de lançar uma *kernel* para cada tarefa, lançar uma *kernel* persistente com o maior número possível de *threads* e blocos de acordo com a capacidade do hardware, que fica ativa até processar todas as tarefas. Enquanto o *kernel* é executado na GPU, o processo host pode adicionar tarefas a uma fila compartilhada com o *device*, de onde o *kernel* retira as tarefas e as processa. Embora essa esquema tenha melhorado o balanceamento nas implementações testadas pelos autores, o trabalho não aborda a exploração da capacidade computacional das CPUs em conjunto com as GPUs.

(TSE *et al*., 2010) propõe dois esquemas de balanceamento dinâmico, incremental-linear e incremental-exponencial. Nessas políticas de balanceamento, cada UP é inicializada com uma tarefa de mesmo tamanho (nesse caso, o tamanho de uma tarefa é definido pelo programador de acordo com o problema; por exemplo, pode ser um conjunto dos dados a serem processados, uma parte das subtarefas a serem executadas, entre outros), de forma que o tamanho da tarefa é aumentado, respectivamente, exponencialmente, por um fator constante

m, ou linearmente, por um fator constante *c*, cada vez que uma UP encerra seu processamento atual e solicita um novo bloco de tarefas a um processo "distribuidor". Dessa forma, em ambos os esquemas, as unidades de processamento mais rápidas passam a executar tarefas maiores conforme a execução avança. Os autores ressaltam que a técnica apresenta alta escalabilidade, uso eficiente dos recursos computacionais disponíveis e que não aumenta significativamente a complexidade de programação. De fato, o método não se restringe a uma arquitetura específica; no trabalho, são realizados testes em um *cluster* de oito nodos onde cada nodo contém uma CPU *Quad-core*, uma GPU e um FPGA. (BELVIRANLI; BHUYAN; GUPTA, 2013), no entanto, aponta que o esquema incremental-linear pode resultar em execução excessiva de blocos de tamanho pequeno, subexplorando o potencial dos aceleradores, e que o esquema exponencial-linear pode causar desbalanceamento de carga conforme a execução se aproxima do fim uma vez que tarefas grandes podem ser designadas para UPs mais lentas; ainda, apresenta resultados que confirmam o exposto. No entanto, cabe ressaltar que a configuração dos parâmetros *m, c* e o tamanho inicial das tarefas influenciam significativamente a performance, e ainda que (TSE *et al*., 2010) menciona a possibilidade de empregar uma política mista de balanceamento, afirmando que os esquemas de balanceamento propostos são altamente flexíveis e podem ser otimizados para um objetivo desejado pelo programador. É possível, por exemplo, que um esquema híbrido que utilize balanceamento exponencial-linear no começo e incremental-linear após um dado ponto diminua os problemas citados por (BELVIRANLI; BHUYAN; GUPTA, 2013), porém isto não foi testado em nenhum dos dois trabalhos.

Em relação à integração com OpenCL, (BINOTTO, 2011) e (LAMA *et al*., 2012) apresentam bibliotecas de balanceamento e escalonamento dinâmicas que visam estender o OpenCL, uma vez que o mesmo não apresenta uma funcionalidade nativa de balanceamento, sendo o escalonamento das tarefas estático e decidido pelo programador. De fato, o OpenCL busca ser um padrão em arquiteturas heterogêneas visando portabilidade, ao custo de alguma eficiência na execução (BINOTTO, 2011). A biblioteca de balanceamento proposta em (BINOTTO, 2011) trabalha em duas fases: primeiro, estabelece um escalonamento inicial das tarefas a partir de um *benchmark* executado *offline* e, depois, mantém um histórico do tempo de execução das tarefas para decidir onde as próximas devem ser escalonadas. Já (LAMA *et al*., 2012) apresenta uma biblioteca que permite executar um *kernel* em vários *devices* heterogêneos, distribuindo para cada *device* uma quantia de trabalho proporcional à sua

capacidade computacional. Além disso, foi implementada uma biblioteca dinâmica que permite que a biblioteca de balanceamento seja utilizada sem necessidade de alterar códigos OpenCL já existentes, nem mesmo recompilá-los.

(LU *et al*., 2012a) apresenta um modelo de programação para otimizar a exploração dos recursos computacionais em *clusters* em que cada nodo apresenta múltiplos *cores* e um acelerador (no caso, uma GPU, mas o método pode ser adaptado para funcionar com mais de um acelerador). Neste modelo, a comunicação entre os nodos ocorre utilizando MPI, porém apenas com um processo em cada nodo; este processo ainda fica responsável por controlar a execução dos *kernels* na GPU e por criar *threads* via OpenMP para explorar os demais *cores* do nodo. Eles apresentam uma fórmula para calcular um balanceamento estático ótimo entre vários *cores* CPU e uma GPU, calculada em função do *speedup* obtido pela GPU em relação a uma CPU; no entanto, esse balanceamento é válido apenas dentro de um nodo do *cluster*, e não há uma proposta de balanceamento considerando toda a arquitetura a ser explorada. Além disso, os autores relatam que muitas aplicações paralelas existentes não exploram o potencial das CPUs, deixando-as apenas para troca de mensagens entre nodos e chamada dos *kernels* nas GPUs. Por fim, fazem considerações sobre a adaptação de códigos legados ao uso de aceleradores, afirmando que há duas principais categorias: (1) portar os trechos mais custosos de código para aceleradores e (2) desenvolver código do começo visando às arquiteturas específicas. Os autores afirmam que a maior parte dos esforços se concentram na primeira categoria, pois desenvolver novos códigos do começo pode ser muito custoso.

O trabalho apresentado em (LI *et al*., 2011) contém algumas semelhanças, ao explorar os vários níveis de paralelismo com MPI, OpenMP e CUDA. O trabalho também apresenta um método para balanceamento de carga intra-nodo entre a CPU e a GPU, o qual utiliza o tempo real de execução das unidades de processamento obtido na primeira iteração. A exploração da arquitetura é feita de forma hierárquica, explorando paralelismo de tarefas entre os nodos, paralelismo de *threads* entre as *cores* e paralelismo de dados na GPU.

(BOSQUE *et al*., 2013) propõe um algoritmo de balanceamento dinâmico para *clusters* heterogêneos. Diferentemente dos demais trabalhos citados, neste caso a heterogeneidade não se refere à colaboração de CPU/aceleradores, mas sim de um *cluster* onde cada nodo tem capacidades computacionais diferentes em relação ao número de *cores* e performance dos mesmos. O algoritmo é baseado em tarefas e é distribuído, de forma que cada nodo decide se há a necessidade de realizar um balanceamento de carga a partir de um *load index*, calculado

considerando fatores estáticos e dinâmicos, como o número de *cores* do nodo, o número de tarefas designadas para aquele nodo e o poder computacional daquele nodo em relação aos demais. Se o *load index* de um nodo está baixo, significa que ele está sobrecarregado com tarefas, e ele procura um nodo com *load index* alto para transferir tarefas de sua fila local.

(BELVIRANLI; BHUYAN; GUPTA, 2013) apresenta uma estratégia de balanceamento dinâmico para arquiteturas heterogêneas com *multicore* e aceleradores baseado em processamento iterativo (loop *for*). O trabalho considera uma abordagem comum para escalonamento em loops na qual o conjunto de dados a ser processado é dividido em grupos de iterações, denominados blocos. Os autores relatam que, em sistemas heterogêneos, o tamanho dos blocos tem mais impacto na performance do que em sistemas *multicore*; assim, o algoritmo se propõe a encontrar o tamanho ideal para os blocos em cada iteração, sendo dividido em duas fases. A primeira fase inicia com blocos pequenos e incrementa seu tamanho gradualmente até encontrar um peso computacional que reflete a capacidade computacional e custos com transferência de memória relativos a cada unidade de processamento. Isto ocorre durante as primeiras iterações da execução da aplicação paralela. Os autores limitam a duração desta fase em 20% do total de iterações, mas ela pode ser encerrada antes se o peso computacional for estabilizado para todas as UPs. Na segunda fase, as demais iterações são executadas. O tamanho do bloco para cada UP é recalculado no início da segunda fase e diminui gradualmente ao longo da execução, de forma que todas as UPs encerrem suas execuções aproximadamente no mesmo tempo.

Este esquema de balanceamento teve bons resultados nos *benchmarks* executados, obtendo balanceamento próximo do ideal da carga de trabalho entre CPU e acelerador e diferença de tempo de execução entre as *threads* próximo de zero segundos. Os *speedups* menos expressivos foram obtidos nos *benchmarks* que apresentavam dependência de processamento e necessidade de sincronização entre as iterações. Por fim, uma API foi implementada com o objetivo de simplificar o desenvolvimento ou adaptação de aplicações paralelas heterogêneas com o algoritmo. No entanto, o método foi aplicado apenas a arquiteturas compostos por um acelerador e diversos *cores*, não sendo explorado em um *cluster* onde cada nodo apresenta tal configuração.

Por fim, (ACOSTA; BLANCO; ALMEIDA, 2013) apresenta um algoritmo de balanceamento dinâmico também baseado em processamento iterativo com sincronização ao fim de cada iteração. Assim como o apresentado em (BELVIRANLI; BHUYAN; GUPTA,

2013), o método também considera a divisão dos dados a serem processados em uma iteração em blocos. O algoritmo busca balancear a carga de trabalho visando igualar o tempo de execução de cada unidade de processamento em uma iteração. Inicialmente, a carga é dividida uniformemente entre as unidades de processamento; ao fim de cada iteração, uma comunicação coletiva é realizada para compartilhar o tempo de execução de cada UP naquela iteração. O poder relativo de cada UP é calculado como a razão entre o tamanho do conjunto processado sobre o tempo de execução e é utilizado para recalcular o tamanho do conjunto para a próxima iteração. O algoritmo de balanceamento é explicado em (GALINDO; ALMEIDA; BADÍA-CONTELLES, 2008). De modo geral, uma UP que levou mais tempo que as demais receberá um conjunto menor de dados para processar, e uma UP que levou menos tempo que as demais receberá mais dados. Caso a diferença entre o maior e menor tempo de execução seja menor que um dado *threshold*, não ocorre balanceamento.

Ainda, (ACOSTA; BLANCO; ALMEIDA, 2013) afirmam que o algoritmo adiciona um *overhead* negligível à execução e que a carga de trabalho é balanceada em poucas iterações. A adaptação de um código existente para utilizar o algoritmo requer poucas mudanças; nos exemplos dados pelos autores, basta a adição de duas linhas no laço principal paralelo. Isso vai ao encontro do relatado pelos autores de que a adaptação de códigos paralelos existentes às novas técnicas é fundamental. Entretanto, assim como (BELVIRANLI; BHUYAN; GUPTA, 2013), o trabalho explorou apenas arquiteturas compostas por diversos cores ou diversas GPUs, mas não ambos; o balanceamento de sistemas com múltiplas CPUs e múltiplas GPUs é citado como trabalho futuro, porém não foi encontrado nenhuma publicação dos autores nesse sentido.

3.3 Considerações do Capítulo

O escalonamento e balanceamento de carga apresentam complexidade adicional quando se trata de arquiteturas heterogêneas. Explorar todos os recursos computacionais à disposição é um fator a mais a ser levado em consideração e um desafio em si. Apesar dos supercomputadores cada vez mais serem construídos de forma heterogênea (MITTAL; VETTER, 2015), explorar todo seu potencial ainda não é uma tarefa simples; de fato, entre os trabalhos aqui citados, apenas (TSE *et al.*, 2010), (LU *et al.*, 2012a) e (LI *et al.*, 2011) apresentam métodos escaláveis para aproveitar todos os níveis de paralelismo citados no Capítulo 2. Mesmo para computadores simples que combinem um processador *multicore* e

um acelerador como a GPU, que também são uma boa alternativa no custo-benefício para suprir demanda por computação (BINOTTO, 2011), muitas aplicações não exploram a arquitetura por completo (LI *et al*., 2011).

Embora a comunidade esteja caminhando para diminuir a complexidade de programação em arquiteturas heterogêneas, como a tentativa de padronizar a programação com OpenCL e ferramentas que auxiliem o programador como as propostas por (BINOTTO, 2011) e (LAMA *et al*., 2012), ainda há muito código legado que pode ser demasiadamente custoso para ser reescrito com as novas ferramentas. Assim, soluções que permitem adaptar códigos já existentes a arquiteturas heterogêneas sem muitas alterações também são relevantes.

Neste sentido, em que pese (BELVIRANLI; BHUYAN; GUPTA, 2013) e (ACOSTA; BLANCO; ALMEIDA, 2013) apresentarem soluções que podem ser adaptadas em códigos existentes sem muita dificuldade, em ambos os casos não se explorou a escalabilidade da solução para arquiteturas com múltiplas CPUs e GPUs, como em um *cluster* heterogêneo. Desta forma, o escopo deste trabalho foi estender tais soluções a fim de viabilizar sua aplicação em arquiteturas mais escaláveis.

Capítulo 4

Biblioteca de Balanceamento Multiframework

Este capítulo descreve a biblioteca de balanceamento Multiframework Balance implementada neste trabalho. Inicialmente, o modelo de aplicações paralelas sobre o qual a biblioteca é baseada é descrito e ilustrado, a fim de facilitar a compreensão de como a biblioteca pode ser integrada de forma simples a códigos existentes. Então, a biblioteca em si é descrita, especificando a interface de programação e o funcionamento do algoritmo de balanceamento. Para fins de simplificar a discussão da modelagem de aplicações paralelas, doravante entende-se que tais aplicações são projetadas visando serem executadas em arquiteturas heterogêneas dentro do escopo do trabalho, ou seja, *clusters* com múltiplos *cores*/GPUs a serem explorados, utilizando as ferramentas MPI para coordenação entre os nós do *cluster* e OpenMP/CUDA para execução paralela nos *cores*/GPUs.

4.1 Modelo de Aplicação Paralela

A biblioteca é desenvolvida levando em conta um modelo de aplicação no qual um processamento é realizado iterativamente sobre um certo domínio, onde cada iteração depende da anterior. Este modelo iterativo é comum a vários problemas, tais como método de Jacobi para solução de sistemas de equação lineares, algoritmos de programação dinâmica, problemas de caminho mínimo e simulações computacionais (ACOSTA; BLANCO; ALMEIDA, 2013). O Algoritmo 1 ilustra a estrutura geral destes problemas.

Pode-se extrair paralelismo deste modelo particionando-se o domínio a ser processado em diversos subdomínios, os quais são processados paralelamente; após isso, os dados são sincronizados entre as linhas de execução paralelas para permitir o processamento da próxima iteração.

Algoritmo 1 – Esqueleto de código para uma implementação sequencial de um problema iterativo

```
1  // N é o tamanho do problema. Em cada iteração, é realizado
2  // um processamento no intervalo [0, N), que depende do
3  // processamento realizado na iteração anterior.
4
5  for(int it = 0; it < num_iteracoes; it++)
6  {
7      for(int i = 0; i < N; i++)
8          processa(i);
9  }
```

A forma como os dados são sincronizados pode variar de acordo com a dependência de dados do problema. Se o problema é tal que, para processar um subdomínio em uma iteração, são necessários os dados do domínio inteiro processados na iteração anterior, então a sincronização pode ser feita como uma comunicação coletiva, em que cada processo envia seu subdomínio aos demais e recebe destes os outros subdomínios (nesse contexto, consideramos os processos como parte de um sistema de memória distribuída). Já caso não seja necessário os dados do domínio inteiro para processar um subdomínio, como em problemas que apresentam dependência de vizinhança, de forma que um subdomínio depende apenas de seus vizinhos, então a sincronização pode ser feita como uma série de comunicações ponto a ponto, em que cada processo envia seu subdomínio ou parte dele apenas aos processos que dele dependem, e recebe dados apenas dos processos dos quais depende. Naturalmente, também pode-se compartilhar o domínio completo com todos os processos, mas isso tende a ser significativamente mais custoso.

A biblioteca Multiframework Balance é mais adaptada para realizar o balanceamento em problemas do primeiro caso, onde todos compartilham o domínio inteiro após cada iteração; desta forma, a discussão procede considerando essa classe de problemas. No entanto, também é possível utilizar a biblioteca para problemas do segundo caso, como será ilustrado em um dos estudos de caso no Capítulo 5.

Uma comunicação coletiva na qual os dados de cada processo são combinados e enviados a todos os processos é uma primitiva paralela comum. De fato, o padrão MPI contém uma comunicação todos-para-todos que realiza exatamente isso, o *MPI_Allgatherv*, que, em C, apresenta o seguinte protótipo:

Algoritmo 2 – Protótipo do método MPI_Allgatherv em C

```
1  int MPI_Allgatherv(const void* sendbuf, int sendcount,
      MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
      const int displs[], MPI_Datatype recvtype, MPI_Comm comm);
```

Os parâmetros do método têm o seguinte significado:

a) sendbuf: endereço do *buffer* de onde serão enviados os dados do processo;
b) sendcount: número de elementos no *buffer*;
c) sendtype: tipo dos elementos do *buffer*;
d) recvbuf: endereço do *buffer* onde serão recebidos os dados de todos os processos;
e) recvcounts: um vetor que contém o número de elementos a ser recebido de cada processo;
f) displs: um vetor que contém, na posição *i*, o *offset* (relativo à recvbuf) aonde os dados vindos do processo *i* serão colocados (o nome do parâmetro deriva do inglês *displacements*);
g) recvtype: tipo dos elementos que serão recebidos;
h) comm: comunicador MPI.

A Figura 4.1 ilustra o funcionamento do *MPI_Allgatherv*. Os processos 0, 1 e 2 contém, respectivamente, 3, 4 e 2 elementos do conjunto a ser compartilhado entre os três (especificado pelos parâmetros sendcount e recvcounts). Os dados são concatenados de acordo com as posições especificadas no parâmetro displs, e então são replicados em todos os processos.

Desta forma, para particionar o domínio entre os processos MPI, é conveniente utilizar dois vetores, um que indica a quantidade de elementos que cada processo computará, e outro que indica o *offset* com relação ao início do intervalo de trabalho em que cada processo computará. Com isso, a sincronização dos dados ao fim das iterações pode ser feita utilizando *MPI_Allgatherv* de forma simples.

Sejam estes vetores *counts* e *displs*, seguindo a nomenclatura do protótipo do método *MPI_Allgatherv*. Então o intervalo de trabalho do processo *i* é

$$[displs[i], displs[i] + counts[i]) \tag{4.1}$$

Isso permite distribuir a carga de trabalho entre os processos MPI. Internamente, cada nó MPI também pode distribuir seu intervalo de trabalho entre os *cores*/GPUs presentes no nó, a

fim de explorar ao máximo a arquitetura paralela. Para tal, pode-se usar a diretiva do OpenMP *#pragma omp parallel* para criar uma região paralela e *#pragma omp for* para paralelizar o

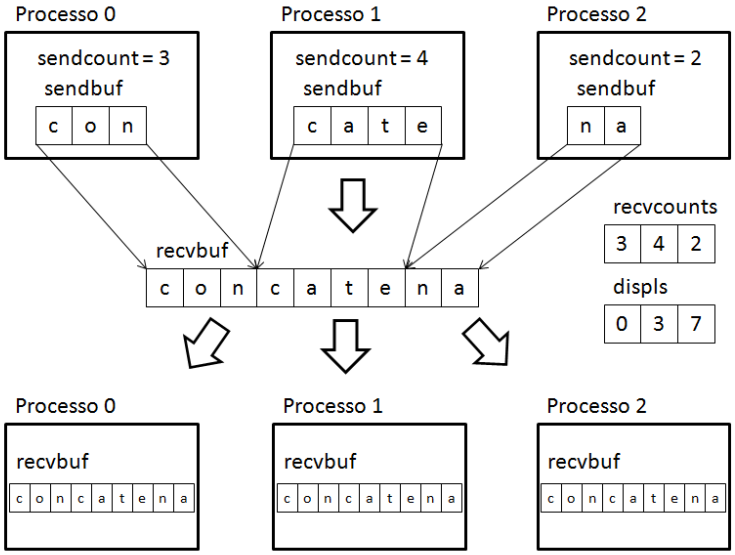


Figura 4.1: Funcionamento do método *MPI_Allgatherv* com 3 processos

laço de processamento. Para cada GPU presente no nodo, uma *thread* fica responsável por invocar um *kernel* CUDA para realizar processamento na GPU. Após o laço, deve-se garantir que todas as *threads* finalizaram seu trabalho antes de compartilhar os resultados; para tal, usa-se a diretiva *#pragma omp barrier*. Por fim, apenas uma *thread* deve realizar a comunicação coletiva; para tal, usa-se a diretiva *#pragma omp single*.

Combinando esses métodos, o Algoritmo 3 ilustra a estrutura geral de uma implementação usando MPI, OpenMP e CUDA para um problema iterativo.

Algoritmo 3 – Esqueleto de código para uma implementação paralela de um problema iterativo

```
1  // N é o tamanho do problema. Em cada iteração, é realizado
2  // um processamento no intervalo [0, N). Os resultados do
3  // processamento são armazenados no vetor result[], que,
4  // portanto, representa o domínio do problema a ser
5  // compartilhado pelos processos MPI.
6
7  int proc;           // Id do processo MPI
8  int num_proc;       // Número de processos MPI
9  int num_threads;    // Número de threads neste nó
10
11
12
13
```

```
14 // Vetor com o offset do intervalo de trabalho de cada processo.
15 int displs[num_proc];
16
17 // Vetor com a qtde de elementos computados por cada processo.
18 int counts[num_proc];
19
20 ...
21
22 #pragma omp parallel num_threads(num_threads)
23 for(int it = 0; it < num_iteracoes; it++)
24 {
25     int tid = omp_get_thread_num();  // Id da thread
26     int ini = displs[proc];
27     int fim = displs[proc] + counts[proc];
28
29     #pragma omp for
30     for(int i = ini; i < fim; i++)
31     {
32         if(thread_usa_GPU(tid)) CUDA_processa(i);
33         else                    processa(i);
34     }
35
36     #pragma omp barrier
37     #pragma omp single
38     MPI_Allgatherv(&result[displs[proc]],
39                    counts[proc],
40                    MPI_DATATYPE,
41                    result,
42                    counts,
43                    displs,
44                    MPI_DATATYPE,
45                    MPI_COMM
46     );
47 }
```

4.2 Multiframework Balance

Multiframework Balance é uma biblioteca que permite balancear a carga de trabalho entre as unidades de processamento em aplicações paralelas utilizando MPI + OpenMP, possibilitando simples adaptação a códigos já existentes. A biblioteca realiza o balanceamento em dois níveis – no nível dos processos MPI e no nível das *threads* OpenMP. As *threads* são abstraídas como linhas paralelas independentes que podem obter cargas de trabalho diferentes – desta forma, é possível também utilizar aceleradores independente da tecnologia empregada. A biblioteca está disponível no GitHub, no endereço https://github.com/luistrivelatto/TCC.

Assim, a biblioteca é uma extensão do trabalho desenvolvido por (ACOSTA; BLANCO; ALMEIDA, 2013). No artigo original, os autores desenvolveram a biblioteca *ULL_MPI_calibrate*, que realiza o balanceamento apenas no nível dos processos MPI.

A biblioteca Multiframework realiza o balanceamento alterando os vetores *counts* e *displs*, descritos na seção anterior, para alterar a carga dos processos MPI; e, internamente, mantém vetores equivalentes para as *threads*, a partir dos quais fornece um subintervalo de processamento para cada *thread* (os vetores *counts* e *displs* não podem ser mantidos internamente pois a sincronização dos dados requer os mesmos, como no caso do *MPI_Allgatherv*).

Desta forma, a carga de trabalho de um problema é abstraída pela biblioteca como um intervalo discreto de tamanho N. Esse intervalo comumente representa um vetor a ser processado, mas pode ter outros significados, como um intervalo de linhas de uma matriz ou uma quantidade de tarefas de uma lista de tarefas a serem executadas. A biblioteca então divide o intervalo em subintervalos contínuos a serem processados por cada processo/*thread*. Um intervalo de tamanho N geralmente significa o intervalo [0, N), porém o programador pode especificar o início e fim do intervalo a ser dividido. Por exemplo, em um dos estudos de caso, o intervalo a ser processado era [1, N+1) pois o elemento 0 era um caso especial, por ser um elemento de borda.

A aplicação da biblioteca em um código paralelo é ilustrada no Algoritmo 4, no qual a biblioteca foi adicionada ao código do algoritmo 3. O balanceamento ocorre por meio da inserção de duas funções, no início e no final do trecho a ser balanceado, dentro do laço principal do código. As linhas alteradas em relação ao algoritmo 3 estão destacadas.

Observa-se que basta alterar 3 linhas do laço principal para utilizar a biblioteca. A linha 8 chama um método da biblioteca que realiza o balanceamento, marca o tempo de início do trabalho para a *thread* que a chamou e retorna o subintervalo daquela *thread*, por meio dos argumentos *ini* e *fim*, que são passados por referência. A linha 17 chama o segundo método da biblioteca, que marca o tempo de fim do trabalho para aquela *thread*. E a linha 10 remove a diretiva *#pragma omp for*, que não é mais necessária, pois a divisão do laço entre as *threads* é realizada pela biblioteca.

Além destas mudanças, fora do laço principal deve-se chamar métodos para inicializar e encerrar a biblioteca. No total, o programador deve adicionar 4 chamadas de função à biblioteca para realizar o balanceamento. Estes métodos são descritos a seguir.

Algoritmo 4 – Esqueleto de código para uma implementação paralela de um problema iterativo usando Multiframework Balance

```
1  #pragma omp parallel num_threads(num_threads)
2  for(int it = 0; it < num_iteracoes; it++)
3  {
4      int tid = omp_get_thread_num();  // Id da thread
5      int ini = displs[proc];
6      int fim = displs[proc] + counts[proc];
7
8      Multiframework_init_section(it, counts, displs, THRESHOLD,
       tid, &ini, &fim);
9
10     #pragma omp for
11     for(int i = ini; i < fim; i++)
12     {
13         if(thread_usa_GPU(tid)) CUDA_processa(i);
14         else                    processa(i);
15     }
16
17     Multiframework_end_section(it, tid);
18
19     #pragma omp barrier
20     #pragma omp single
21     MPI_Allgatherv(&result[displs[proc]],
22                    counts[proc],
23                    MPI_DATATYPE,
24                    result,
25                    counts,
26                    displs,
27                    MPI_DATATYPE,
28                    MPI_COMM
29     );
30 }
31
```

Algoritmo 5 – Protótipo do método Multiframework_Init_lib

```
1  void Multiframework_Init_lib(
2      int problem_begin,
3      int problem_end,
4      int num_threads,
5      int proc_id,
6      int num_proc,
7      MPI_Comm comm
8  );
```

O Algoritmo 5 apresenta o protótipo do método Multiframework_Init_lib, que deve ser chamado antes do trecho a ser balanceado, para inicializar a biblioteca. Os parâmetros têm o seguinte significado:

a) Problem_begin: inteiro representando o início do intervalo a ser dividido. Tipicamente, 0;
b) Problem_end: inteiro representando o fim do intervalo a ser dividido. Tipicamente, N;
c) Num_threads: número de *threads* neste nó;
d) Proc_id: id do processo MPI neste nó;
e) Num_proc: número de processos MPI;
f) Comm: comunicador MPI do grupo de processos.

Algoritmo 6 – Protótipo do método Multiframework_Finalize_lib

```
1  void Multiframework_Finalize_lib()
```

O Algoritmo 6 apresenta o protótipo do método Multiframework_Finalize_lib, que deve ser chamado após o trecho a ser balanceado, para liberar a memória alocada e encerrar a biblioteca.

Algoritmo 7 – Protótipo do método Multiframework_begin_section

```
1  void Multiframework_begin_section(
2      int iteration,
3      int *counts,
4      int *displs,
5      int threshold,
6      int thread_id,
7      int *my_begin,
8      int *my_end
9  );
```

O Algoritmo 7 apresenta o protótipo do método Multiframework_begin_section, que deve ser chamado no início do trecho paralelo a ser balanceado. Os parâmetros têm o seguinte significado:

a) Iteration: a iteração atual. O valor 0 indica a primeira iteração, que tem um tratamento especial (a carga é distribuída uniformemente entre os processos e, dentro de cada processo, homogeneamente entre as *threads*);
b) Counts: um vetor que armazena o número de elementos processados por cada processo, conforme descrito na seção anterior. Esse vetor é modificado pela biblioteca, e não precisa ser inicializado antes de ser passado a este método;
c) Displs: um vetor que armazena início do intervalo de trabalho de cada processo, conforme descrito na seção anterior. Esse vetor é modificado pela biblioteca, e não

precisa ser inicializado antes de ser passado a este método. Mais especificamente, na iteração 0 o método sobrescreverá os valores em *counts* e *displs*;

d) Threshold: Corresponde a uma porcentagem que indica se o balanceamento deve ser realizado ou não. Quando a diferença entre os tempos de processamento em nível de *threads* ou de processos não exceder o *threshold*, então assume-se que o sistema está balanceado e não é realizado um rebalanceamento. Especificamente, esse parâmetro funciona da seguinte forma: seja T_i o tempo que o *i*-ésimo processo/*thread* levou para processar a última iteração. Seja T_{max} = maior(T_i) e T_{min} = menor(T_i). Então o sistema é considerado balanceado se

$$100 - \ 100 * \frac{T_{min}}{T_{max}} \leq threshold \tag{4.2}$$

e) Thread_id: id desta *thread* (todas as *threads* devem chamar o método);

f) My_begin: parâmetro de saída. Ponteiro para um inteiro que receberá o início do intervalo de trabalho da thread *thread_id*;

g) My_end: parâmetro de saída. Ponteiro para um inteiro que receberá o fim do intervalo de trabalho da thread *thread_id*. Ou seja, o intervalo de trabalho desta thread é [my_begin, my_end).

Algoritmo 8 – Protótipo do método Multiframework_end_section

```
1  void Multiframework_end_section(
2      int iteration,
3      int thread_id
4  );
```

Por fim, o Algoritmo 8 apresenta o protótipo do método Multiframework_end_section que deve ser chamado no fim do trecho paralelo a ser balanceado. Os parâmetros têm o seguinte significado:

a) Iteration: a iteração atual;

b) Thread_id: id desta *thread* (todas as *threads* devem chamar o método).

4.3 Algoritmo de Balanceamento

O algoritmo de balanceamento implementado na biblioteca utiliza a mesma estratégia do algoritmo apresentado em (GALINDO; ALMEIDA; BADÍA-CONTELLES, 2008). Os autores descrevem o método como um algoritmo simples, mas que obteve resultados

satisfatórios. O princípio básico é tentar igualar o tempo de execução de cada unidade de processamento; unidades de processamento com tempo de execução alto recebem menos carga, e unidades de processamento com tempo de execução baixo recebem mais carga.

O algoritmo é aplicado utilizando os tempos de trabalho obtidos na iteração anterior. No nível dos processos, os tempos são compartilhados entre os mesmos por meio de uma comunicação coletiva MPI; no nível das *threads*, não é necessário comunicação pois cada processo só precisa do tempo das *threads* em seu nó. O tempo de trabalho das *threads* é calculado como a diferença entre o tempo registrado pela *thread* ao chamar *Multiframework_begin_section* e *Multiframework_end_section*. Já o tempo do processo é calculado como a diferença entre o tempo registrado pela primeira *thread* a chamar *Multiframework_begin_section* e o tempo registrado pela última *thread* a chamar *Multiframework_end_section*, em cada iteração.

O balanceamento funciona como se segue. Seja T[] um vetor com o tempo de trabalho de cada unidade de processamento, e counts[] um vetor com a carga de trabalho. Primeiramente, obtém-se o maior e menor T[i] para observar se o *threshold* foi excedido. Caso não tenha sido, considera-se que o sistema está balanceado e o algoritmo é encerrado. Senão, calcula-se o poder relativo de cada unidade de processamento como a razão entre a carga de trabalho e o tempo de trabalho de tal unidade; este valor é armazenado no vetor RP[]. Também é calculada a soma do vetor RP[], armazenado em SRP. Ou seja,

$$\begin{aligned} RP[i] &= \frac{counts[i]}{T[i]} \\ SRP &= \sum_{i} RP[i] \end{aligned} \tag{4.3}$$

Após isso, o vetor counts[] é recalculado de forma que a carga de trabalho é distribuída para cada processo de acordo com a razão entre RP[i] e SRP. Para garantir que a distribuição ocorre corretamente, o último elemento é calculado especialmente como a diferença entre a soma dos demais e o tamanho da carga. Ou seja, se *problemSize* é o tamanho do intervalo a ser dividido, então

$$\begin{aligned} counts[i] &= round\left(problemSize * \frac{RP[i]}{SRP}\right) && se\ 0 \leq i \leq n-2 \\ counts[i] &= problemSize - \sum_{i=0}^{n-2} counts[i] && se\ i = n-1 \end{aligned} \tag{4.4}$$

Tendo calculado o vetor counts[], o vetor displs[] é atualizado de acordo. Se [L, R) representa o intervalo a ser dividido (sendo *problemSize* a diferença entre R e L), então

$$displs[0] = L$$
$$displs[i] = displs[i-1] + count[i-1] \qquad para\ i \geq 1 \tag{4.5}$$

O Algoritmo 9 apresenta o código em C que realiza o balanceamento. No início de cada iteração, na chamada a *Multiframework_begin_section*, cada processo MPI executa este método, calculando a distribuição de carga entre os processos. Após, cada processo executa este método considerando a distribuição entre as *threads* daquele nó. Dessa forma, na primeira execução os parâmetros [L, R) representam o intervalo total de trabalho, e, na segunda, representam o subintervalo daquele nó.

Algoritmo 9 – Método que realiza o balanceamento de carga entre as unidades de processamento

```
1  int balance(int *count, int *displ, double *t, int n, int l, int
      r, int threshold)
2  {
3      double tmax = t[0], tmin = t[0];
4      for(int i = 1; i < n; i++)
5      {
6          if(t[i] > tmax) tmax = t[i];
7          if(t[i] < tmin) tmin = t[i];
8      }
9
10     if(100 - tmin * 100 / tmax <= threshold)
11         return 0;
12
13     double rp[n], srp = 0;
14     for(int i = 0; i < n; i++)
15     {
16         rp[i] = count[i] / t[i];
17         srp += rp[i];
18     }
19
20     int problem_size = r - l;
21     int sc = 0;
22     for(int i = 0; i+1 < n; i++)
23     {
24         count[i] = round(problem_size * rp[i] / srp);
25         sc += count[i];
26     }
27     count[n-1] = problem_size - sc;
28
29     displ[0] = l;
30     for(int i = 1; i < n; i++)
31         displ[i] = displ[i-1] + count[i-1];
32     return 1;
33 }
```

Capítulo 5

Experimentos: Estudos de Caso

Para validar a biblioteca de balanceamento, foram implementados três problemas dentro do modelo iterativo especificado: o método de Jacobi para resolução de sistemas de equações lineares, uma solução com programação dinâmica para o Problema da Alocação de Recursos (em inglês, RAP – *Resource Allocation Problem*) e uma simulação computacional para o problema da transferência de calor. Para ilustrar o uso da biblioteca no código, são apresentados os códigos paralelos do Método de Jacobi, utilizando MPI e OpenMP, e do RAP, utilizando MPI, OpenMP e CUDA. Os problemas são descritos a seguir.

5.1 Método de Jacobi

O método de Jacobi é um algoritmo para determinar as soluções de um sistema de equações lineares de forma iterativa (BINOTTO, 2011). Seja

$$Ax = B \tag{5.1}$$

um sistema de *N* equações lineares, onde

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \qquad B = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{5.2}$$

Por meio de operações algébricas, é possível obter a seguinte aproximação para *x*:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \qquad i = 1,2, \ldots n \tag{5.3}$$

Em que x^k representa a *k*-ésima aproximação de *x*, e x^{k+1} representa a próxima aproximação, na iteração seguinte. Desta forma, x^0 é obtido por meio de uma estimativa inicial e, a partir daí, a solução é obtida iterativamente aproximando mais a solução anterior. Observa-se que cálculo de x_i^{k+1} depende de todos os elementos de x^k (exceto de si mesmo). O código sequencial do método de Jacobi é apresentado no Algoritmo 10.

Algoritmo 10 – Código sequencial para o método de Jacobi

```
1  for(int it = 0; it < ITERACOES; it++)
2  {
3      for(int i = 0; i < N; i++)
4      {
5          double soma = 0;
6          for(int j = 0; j < N; j++)
7              soma += a[i][j] * x[j];
8          soma -= a[i][i] * x[i];
9          new_x[i] = (b[i] - soma) / a[i][i];
10     }
11
12     for(int i = 0; i < N; i++)
13         x[i] = new_x[i];
14 }
```

Para paralelizar o algoritmo, podemos dividir a computação do vetor *new_x[]* entre as unidades de processamento. Ao fim de cada iteração, utilizamos uma comunicação coletiva para compartilhar os valores calculados neste vetor entre os processos. Um código paralelo utilizando OpenMP e MPI é apresentado no Algoritmo 11. Já o Algoritmo 12 apresenta o mesmo código com a inclusão da biblioteca Multiframework Balance. As linhas 1, 10, 12, 22, 24 e 37 estão destacadas pois representam as diferenças entre os códigos.

Algoritmo 11 – Código paralelo para o método de Jacobi

```
1  #pragma omp parallel num_threads(num_threads) default(shared)
2  for(int it = 0; it < ITERACOES; it++){
3     int tid = omp_get_thread_num();
4     int ini = displs[proc];
5     int fim = displs[proc] + counts[proc];
6     #pragma omp for
7     for(int i = ini; i < fim; i++)
8     {
9        double soma = 0;
10       for(int j = 0; j < N; j++)
11            soma += a[i][j] * x[j];
12       soma -= a[i][i] * x[i];
13       new_x[i] = (b[i] - soma) / a[i][i];
14    }
15    #pragma omp single
16    MPI_Allgatherv(&new_x[displs[proc]],
17                         counts[proc],
18                         MPI_DOUBLE,
19                         x,
20                         counts,
21                         displs,
22                         MPI_DOUBLE,
23                         MPI_COMM_WORLD
24    );
25 }
```

Algoritmo 12 – Código paralelo com Multiframework Balance para o método de Jacobi

```
1  Multiframework_Init_lib(0,   N,   num_threads,   proc,   num_proc,
     MPI_COMM_WORLD);
2
3  #pragma omp parallel num_threads(num_threads) default(shared)
4  for(int it = 0; it < ITERACOES; it++)
5  {
6     int tid = omp_get_thread_num();
7     int ini = displs[proc];
8     int fim = displs[proc] + counts[proc];
9
10    Multiframework_begin_section(it, counts, displs, MF_THRESHOLD,
      tid, &ini, &fim);
11
12    #pragma omp for
13    for(int i = ini; i < fim; i++)
14    {
15       double soma = 0;
16       for(int j = 0; j < N; j++)
17            soma += a[i][j] * x[j];
18       soma -= a[i][i] * x[i];
19       new_x[i] = (b[i] - soma) / a[i][i];
20    }
21
22    Multiframework_end_section(it, tid);
23
24    #pragma omp barrier
25    #pragma omp single
26    MPI_Allgatherv(&new_x[displs[proc]],
27                         counts[proc],
28                         MPI_DOUBLE,
29                         x,
30                         counts,
31                         displs,
32                         MPI_DOUBLE,
33                         MPI_COMM_WORLD
34    );
35 }
36
37 Multiframework_Finalize_lib();
```

5.2 RAP

O Problema da Alocação de Recursos (do inglês *Resource Allocation Problem* – RAP) consiste em alocar *M* unidades de um recurso indivisível para *N* tarefas de forma a maximizar a eficiência do sistema. O problema pode ser enunciado como:

$$\max z = \sum_{j=1}^{N} f_j(x_j) \quad sujeito\ a \sum_{j=1}^{N} x_j = M \tag{5.4}$$

em que $f_j(x_j)$ é o ganho de se alocar x_j unidades do recurso para a tarefa *j*. Uma abordagem para resolver o problema é utilizar programação dinâmica. Seja $G[i][j]$ o melhor ganho considerando as primeiras *i* tarefas e as primeiras *j* unidades do recurso. Então pode-se definir as seguintes equações de recorrência (ACOSTA *et al.*, 2010):

$$\begin{aligned} G[i][j] &= \max\{G[i-1][j-x] + f_i(x), 0 < x \leq j\} && para\ i \geq 2 \\ G[1][j] &= f_1(j) && para\ 0 < j \leq M \\ G[i][0] &= 0 \end{aligned} \tag{5.5}$$

O valor de $G[N][M]$ então contém o benefício máximo para o problema. Para calcular esse valor, deve-se calcular todos os valores da tabela. Uma vez que cada linha depende dos valores da linha anterior, a tabela pode ser computada linha a linha. O Algoritmo 13 apresenta o código sequencial para o RAP, onde *N* é o número de tarefas, *M* é a quantidade do recurso, *P* é a função que indica o ganho de se alocar *x* unidades do recurso para cada tarefa e *G* é a tabela da programação dinâmica.

Algoritmo 13 – Código sequencial para solução do RAP com programação dinâmica

```
1  for(int i = 1; i <= N; i++)
2  {
3      for(int j = 0; j <= M; j++)
4      {
5          G[i][j] = P[i][0];
6          for(int x = 0; x <= j; x++)
7          {
8              int fij = G[i-1][j-x] + P[i][x];
9              if(G[i][j] < fij)
10                 G[i][j] = fij;
11         }
12     }
13 }
```

O algoritmo pode ser paralelizado dividindo o intervalo [0, *M*+1) entre as unidades de processamento, de forma que, a cada uma das *N* iterações, cada unidade é responsável por computar parte das colunas da matriz. Ao fim da *i*-ésima iteração, uma comunicação coletiva replica a linha *i* da matriz em todos os processos, de forma a calcular a próxima iteração.

Observa-se que o laço interno é irregular, uma vez que as iterações não realizam a mesma quantidade de processamento. Especificamente, a computação da *j*-ésima iteração custa *O*(j).

Neste caso, a aplicação da biblioteca de balanceamento torna-se ainda mais interessante, até mesmo em um arquitetura homogênea, pois a divisão ótima da carga de trabalho é não trivial (diferentemente do método de Jacobi, onde a divisão homogênea do trabalho, em uma arquitetura homogênea, seria a mais eficiente). O Algoritmo 14 apresenta o código paralelo do RAP com MPI, OpenMP e CUDA, e a biblioteca Multiframework Balance.

Algoritmo 14 – Código paralelo para solução do RAP com MPI, OpenMP, CUDA e Multiframework Balance

```
1  __global__ void kernel_compute(int *d_G, int *d_P, int i, int
     begin, int end)
2  {
3      #define _G(i, j) (d_G[(i) * TAM + (j)])
4      #define _P(i, j) (d_P[(i) * TAM + (j)])
5
6      int index = blockIdx.x * blockDim.x + threadIdx.x;
7      int stride = blockDim.x * gridDim.x;
8
9      for(int j = begin + index; j < end; j += stride)
10     {
11         int aux = _P(i, 0);
12         for(int x = 0; x <= j; x++)
13         {
14             int fij = _G(i-1, j-x) + _P(i, x);
15             if(aux < fij)
16                 aux = fij;
17         }
18         _G(i, j) = aux;
19     }
20
21     #undef _G
22     #undef _P
23 }
24
25 ...
26
27
28 Multiframework_Init_lib(0,  M+1,  num_threads,  proc,  num_proc,
     MPI_COMM_WORLD);
29
30 #pragma omp parallel num_threads(num_threads) default(shared)
31 {
32     int tid = omp_get_thread_num();
33
34     if(thread_usa_GPU(tid))
35     {
36         cudaSetDevice(tid);
37         cudaMalloc(&d_G, sizeof(G));
38         cudaMalloc(&d_P, sizeof(P));
39         cudaMemcpy(d_P, P, sizeof(P), cudaMemcpyHostToDevice);
40     }
```

```
41
42      for(int i = 1; i <= N; i++)
43      {
44          int it = i-1, ini, fim;
45
46          Multiframework_begin_section(it,      counts,      displs,
    MF_THRESHOLD, tid, &ini, &fim);
47
48          if(thread_usa_GPU(tid))
49          {
50              int block_size = 1024;
51              int num_blocks = (fim - ini + block_size - 1)
52                               / block_size;
53
54              cudaMemcpy(&d_G[(i-1) * TAM + 0],
55                         &G[i-1][0],
56                         sizeof(int) * TAM,
57                         cudaMemcpyHostToDevice);
58
59              kernel_compute<<<num_blocks,  block_size>>>(d_G,  d_P,
    i, ini, fim);
60              cudaDeviceSynchronize();
61
62              cudaMemcpy(&G[i][ini],
63                         &d_G[i * TAM + ini],
64                         sizeof(int) * (fim - ini),
65                         cudaMemcpyDeviceToHost);
66          }
67          else
68          {
69              for(int j = ini; j < fim; j++)
70              {
71                  G[i][j] = P[i][0];
72                  for(int x = 0; x <= j; x++)
73                  {
74                      int fij = G[i-1][j-x] + P[i][x];
75                      if(G[i][j] < fij)
76                          G[i][j] = fij;
77                  }
78              }
79          }
80
81          Multiframework_end_section(it, tid);
82
83
84
85
86
87
88
89
90
91
```

```
92          #pragma omp barrier
93          #pragma omp single
94          MPI_Allgatherv(&G[i][displs[proc]],
95                         counts[proc],
96                         MPI_INT,
97                        &G[i][0],
98                         counts,
99                         displs,
100                        MPI_INT,
101                        MPI_COMM_WORLD
102          );
103       }
104
105       if(thread_usa_GPU(tid))
106       {
107           cudaFree(d_G);
108           cudaFree(d_P);
109       }
110   }
111
112   Multiframework_Finalize_lib();
```

5.3 Transferência de Calor

O problema da transferência de calor busca simular a propagação de calor em um corpo a fim de determinar a temperatura em um dado instante de tempo. Em uma placa retangular, caso existam pontos submetidos a diferentes temperaturas, ao longo do tempo ocorrerá dissipação do calor por meio de troca de energia entre as partículas do material a fim de alcançar um equilíbrio térmico. Em uma placa isotrópica e homogênea, na qual as propriedades térmicas do material são as mesmas em qualquer direção, este processo pode ser modelado por meio da equação do calor.

Para isso, é necessário que haja uma discretização do domínio, que consiste em considerar uma malha com distância fixa entre os nós de Δx no eixo das abscissas e Δy no eixo das ordenadas. O processo pode então ser modelado pela seguinte equação (BORGES; PADOIN, 2006):

$$U_{i,j}^{n+1} = U_{i,j}^{n} + s_x\left(U_{i+1,j}^{n} - 2U_{i,j}^{n} + U_{i-1,j}^{n}\right) + s_z\left(U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}\right) \quad (5.6)$$

em que $U_{i,j}^{n}$ representa a temperatura no ponto i, j da malha na iteração n, e s_x, s_z representam parâmetros definidos em função das medidas escolhidas para as grandezas físicas e da difusividade térmica do material. A Figura 5.1 apresenta os resultados de uma simulação

do problema da transferência de calor, onde quatro momentos diferentes da simulação podem ser observados. A imagem retrata um experimento no qual uma fonte de calor (100º C) é posta no lado inferior do domínio; conforme a simulação avança, o calor propaga-se ao longo da placa quadrada.

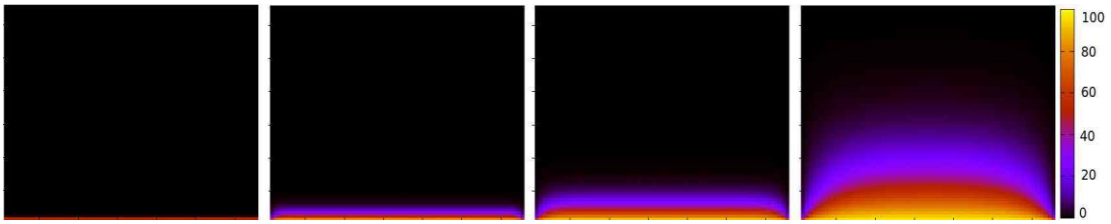


Figura 5.1: Exemplo de resolução do problema de transferência de calor

O Algoritmo 15 apresenta uma implementação sequencial para o problema da transferência de calor.

Algoritmo 15 – Código sequencial para problema da transferência de calor.

```
1  for(int it = 0; it < ITERACOES; it++)
2  {
3      for(int i = 1; i <= TAM; i++)
4          for(int j = 1; j <= TAM; j++)
5              novo_grid[i][j] = calcula_valor(
6                          grid[i][j],
7                          grid[i+1][j],
8                          grid[i-1][j],
9                          grid[i][j+1],
10                         grid[i][j-1]
11                     );
12
13     copia_matriz(grid, novo_grid);
14 }
```

Para paralelizar este algoritmo, pode-se dividir o *grid* em *grids* menores, que são processados concorrentemente. No entanto, diferentemente dos outros dois problemas analisados, no problema da transferência de calor, os processos não necessitam de todo o domínio do problema para computar seu subdomínio, mas apenas das fronteiras dos subdomínios vizinhos. Desta forma, se dividirmos o *grid* ao longo das linhas, a sincronização dos dados consiste apenas em cada processo enviando e recebendo as duas primeiras e duas últimas linhas do *subgrid* que processou.

A biblioteca Multiframework Balance não se adapta tão bem a esta forma de sincronização dos dados pois, ao alterar a carga dos processos MPI, pode ser necessário que um processo

envie e receba diversas linhas do *grid* para diversos outros processos, fazendo com que o gargalo do tempo de execução passe a ser a troca de dados, anulando os benefícios do balanceamento.

Desta forma, utilizou-se uma estratégia diferente para balancear a aplicação, consistindo em duas fases. Na primeira fase, os processos compartilhavam entre si o domínio inteiro, permitindo que a biblioteca realizasse a redistribuição da carga de trabalho normalmente; a partir de então, na segunda fase, a distribuição encontrada pela biblioteca ao fim da primeira fase era mantida fixa, e os processos passavam a compartilhar os dados apenas com seus vizinhos.

Empiricamente, foi possível determinar que, com tão pouco quanto 3 iterações, a carga já era balanceada dentro do *threshold* de 10% da biblioteca. Desta forma, fixou-se que a primeira fase consistiria nas 4 primeiras iterações, e a segunda fase consistiria nas demais. Uma alternativa seria rodar a primeira fase enquanto a biblioteca não indicasse que a carga estava balanceada dentro do *threshold* desejado.

Capítulo 6

Experimentos: Resultados

Este capítulo apresenta os resultados das implementações dos estudos de caso. Foram implementadas cinco versões para o método de Jacobi e para o RAP: sequencial, paralela com OpenMP e MPI com e sem balanceamento e paralela com OpenMP, MPI e CUDA com e sem balanceamento (para o problema da transferência de calor, foram implementadas apenas as versões sequencial e com OpenMP e MPI com e sem balanceamento). Nas versões sem balanceamento, utilizou-se uma distribuição uniforme da carga de trabalho entre os processos MPI e uma distribuição uniforme da carga de cada processo entre as *threads* disponíveis ao mesmo. Nos testes com balanceamento, utilizou-se *threshold* de 0% para a biblioteca de balanceamento, exceto nos testes do problema da transferência de calor, em que utilizou-se 10%.

As implementações foram executadas em um *cluster* com 3 nós com as seguintes configurações:

a) máquina 1: um processador Intel Core i3-2100, com 2 *cores* e frequência de 3.10 GHz, contendo 4 GB de memória RAM;
b) máquina 2: dois processadores Intel Xeon E5620, com 4 *cores* cada e frequência 2.40 GHz, e 16 GB de memória RAM, além de uma placa gráfica Tesla K20c contendo 2560 núcleos CUDA com frequência de 706 MHz e 5 GB de memória;
c) máquina 3: um processador Pentium Dual-Core E5200, com 2 *cores* e frequência de 2.50 GHz, e 1 GB de memória RAM.

As implementações sequenciais foram executadas na máquina 1, que apresentou os melhores resultados.

Embora a máquina 2 possua 8 *cores*, nos testes foram utilizados apenas 4 em função da ocupação de alguns *cores* com outros processos da máquina, o que se observou estar influenciando as execuções. Além disso, nas execuções com CUDA, uma *thread* fica

responsável por cuidar somente da execução na GPU, de forma que a execução ocorre em 4 linhas de execução paralela nesta máquina: três *threads* executando na CPU e uma executando na GPU, utilizando a CPU apenas para controle da execução.

Nos dados apresentados a seguir, os processos MPI 1, 2 e 3 referem-se sempre às máquinas 1, 2 e 3, respectivamente. Além disso, nas execuções com CUDA, a GPU é sempre controlada pela *thread* 1 do processo 2. Nos gráficos, os processos são referidos como P1, P2 e P3, enquanto as *threads* são referidas como P1 T1 para indicar a *thread* 1 do processo 1, P1 T2 para indicar a *thread* 2 do processo 1, e assim por diante.

Os dados são apresentados a partir da média de 15 testes, exceto nas subseções intituladas "Tempo e Carga de Trabalho", nos quais os dados apresentados são obtidos a partir de um teste único escolhido aleatoriamente.

6.1 Método de Jacobi

Os testes do método de Jacobi utilizaram 10.000 variáveis e executaram 5.000 iterações. A Tabela 6.1 apresenta o tempo de execução de cada implementação.

Tabela 6.1: tempo de execução e *speedup* por implementação

	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
Tempo (s)	1564	470	303	470	245
Speedup vs sequencial	-	3,32	5,17	3,32	6,39
Speedup vs não balanc.	-	-	1,55	-	1,92

Observa-se que as versões balanceadas apresentaram *speedup* de 1,55 e 1,92 com relação às implementações equivalentes sem balanceamento. O *speedup* maior na implementação com CUDA é explicado pelo mesmo motivo pelo qual as implementações com e sem CUDA, não balanceadas, apresentam tempos de execução praticamente iguais: o gargalo de cada iteração ocorre no processo 3, e os outros processos ficam ociosos até que o processo 3 finalize sua parte do processamento. Ou seja, em função do desbalanceamento de carga, não há benefício nenhum em utilizar a GPU ao invés de utilizar um *core* normal da CPU. Esse comportamento pode ser observado nas Figuras 6.1 e 6.2, que ilustram o tempo médio e a carga média por iteração para os processos/*threads*.

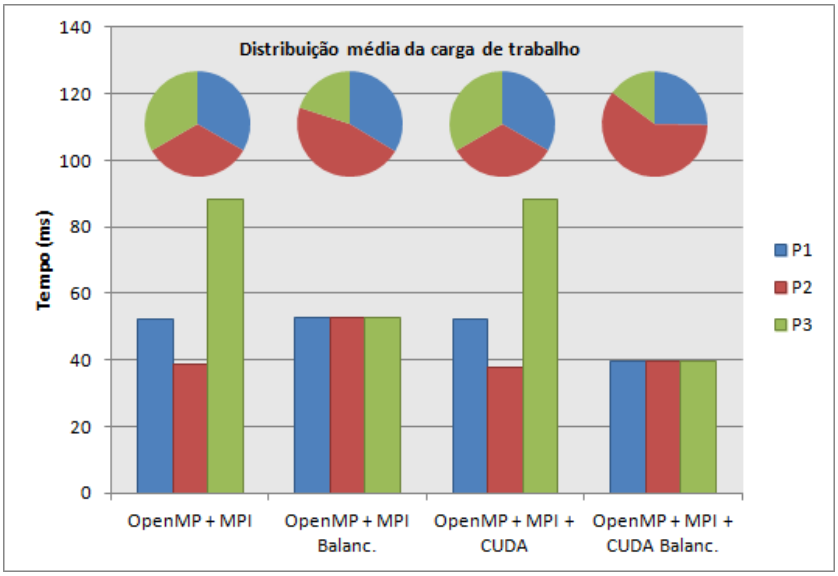


Figura 6.1: Tempo de trabalho médio por iteração por processo. Em cima, distribuição média da carga de trabalho entre os processos

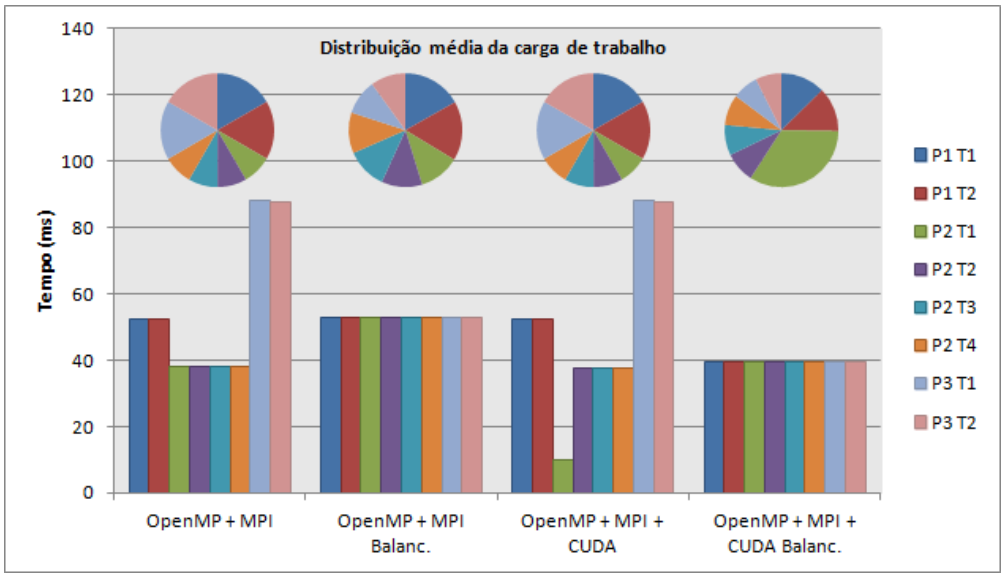
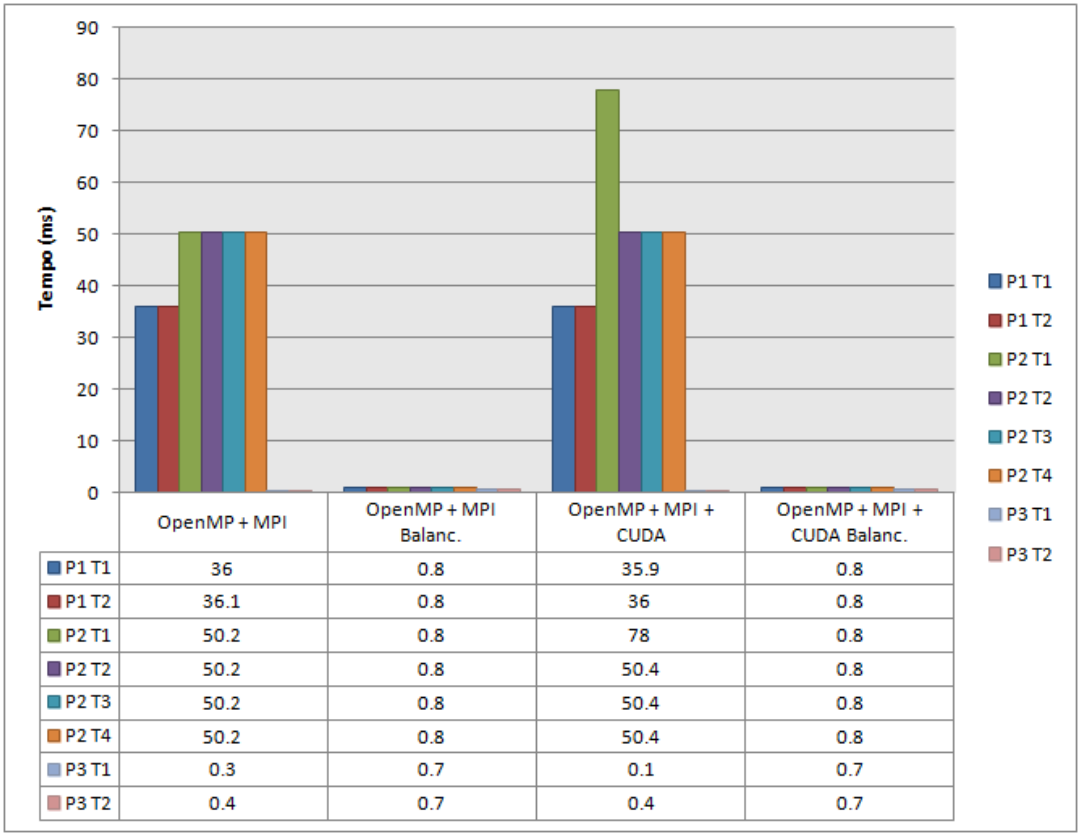


Figura 6.2: Tempo de trabalho médio por iteração por *thread*. Em cima, distribuição média da carga de trabalho entre as *threads*

Destaca-se como o balanceamento "aproveita" o maior poder de execução da GPU, que recebe uma fração significativa da carga de trabalho, igualando seu tempo de trabalho com o tempo das demais *threads*. Também pode-se observar que, nas execuções balanceadas, o tempo médio de trabalho em cada iteração de todas as *threads* é praticamente o mesmo, o que diminui o tempo ocioso de cada *thread*, otimizando o uso da arquitetura. O tempo ocioso médio por iteração para cada *thread*, calculado em cada iteração como a diferença entre o

tempo de trabalho de uma *thread* e o maior tempo de trabalho observado entre todas as *threads* naquela iteração, pode ser visto na Figura 6.3.



	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
P1 T1	36	0.8	35.9	0.8
P1 T2	36.1	0.8	36	0.8
P2 T1	50.2	0.8	78	0.8
P2 T2	50.2	0.8	50.4	0.8
P2 T3	50.2	0.8	50.4	0.8
P2 T4	50.2	0.8	50.4	0.8
P3 T1	0.3	0.7	0.1	0.7
P3 T2	0.4	0.7	0.4	0.7

Figura 6.3: Tempo ocioso médio por iteração por *thread*

6.1.1 Tempo e Carga de Trabalho – OpenMP + MPI

As Figuras 6.4 e 6.5 apresentam o tempo de trabalho e carga dos processos e das *threads* nas primeiras iterações do teste. Percebe-se como logo na segunda iteração a carga já está próxima do balanceamento ideal, apresentando 2,9 milissegundos de diferença entre o maior e o menor tempo de trabalho entre as *threads*, em contraste com os 48,2 milissegundos da primeira iteração. Isso seria suficiente para considerar o sistema balanceado usando um *threshold* de 6%. A Figura 6.6 apresenta a distribuição da carga ao longo da execução, mostrando que, logo após o rebalanceamento das primeiras iterações, a carga de cada processo/*thread* continua praticamente a mesma durante toda a execução.

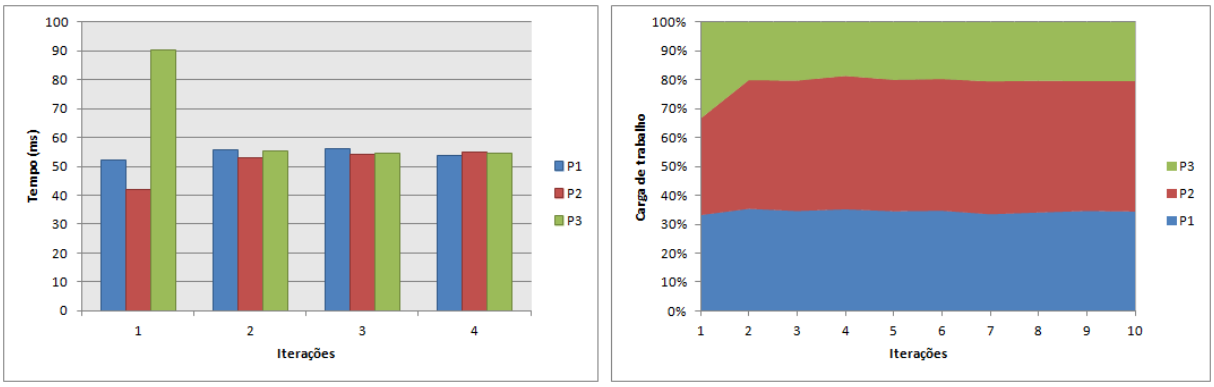


Figura 6.4: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada processo nas primeiras iterações da execução

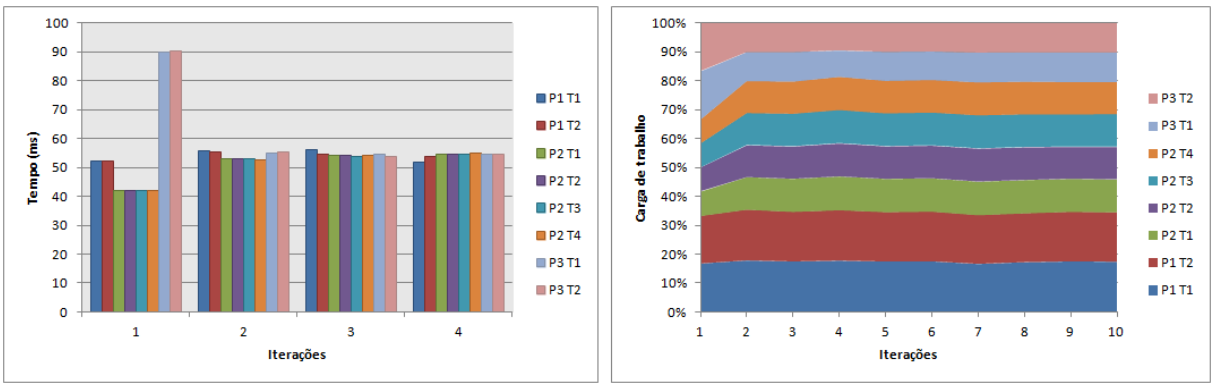


Figura 6.5: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada *thread* nas primeiras iterações da execução

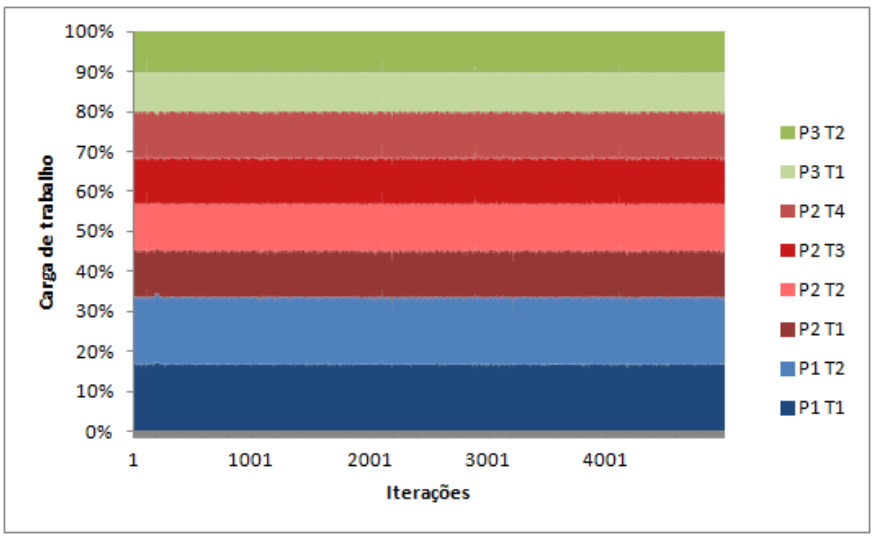


Figura 6.6: Distribuição da carga de trabalho entre as *threads* ao longo da execução

6.1.2 Tempo e Carga de Trabalho – OpenMP + MPI + CUDA

As Figuras 6.7 e 6.8 apresentam o tempo de trabalho e carga dos processos e das *threads* nas primeiras iterações do teste. Novamente, em poucas iterações a carga já está próxima do balanceamento ideal – na quarta iteração, a diferença entre o maior e o menor tempo de trabalho entre as *threads* foi de 0,4 milissegundos, suficiente para considerar o sistema balanceado usando um *threshold* de 1%. A Figura 6.9 apresenta a distribuição da carga ao longo da execução, novamente mostrando que o balanceamento rapidamente se estabilizou, mantendo-se similar durante toda a execução.

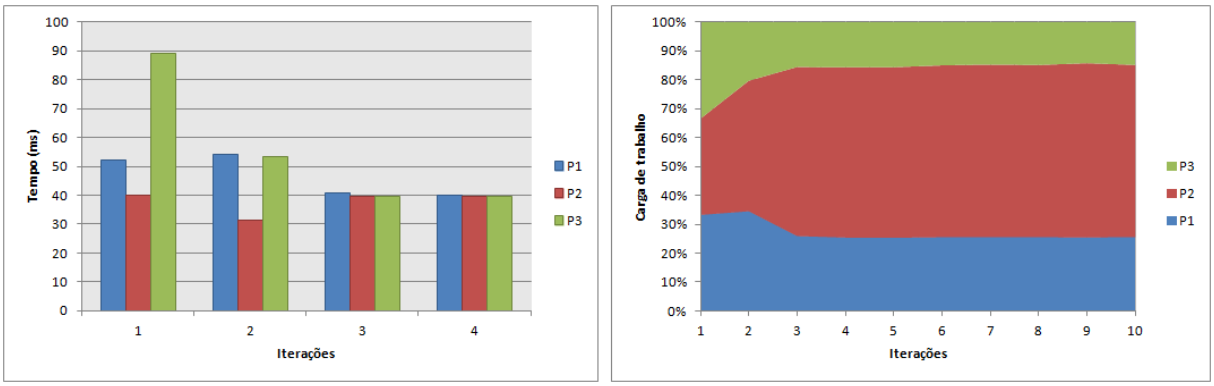


Figura 6.7: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada processo nas primeiras iterações da execução

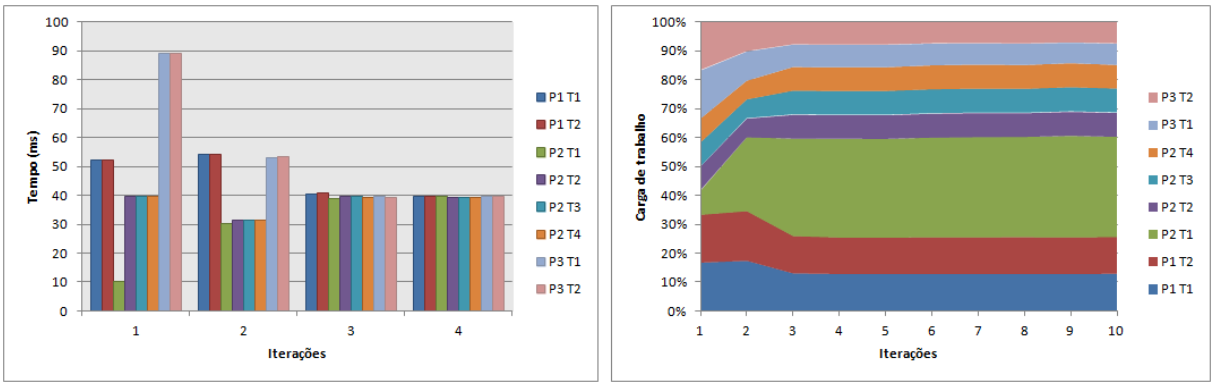


Figura 6.8: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada *thread* nas primeiras iterações da execução

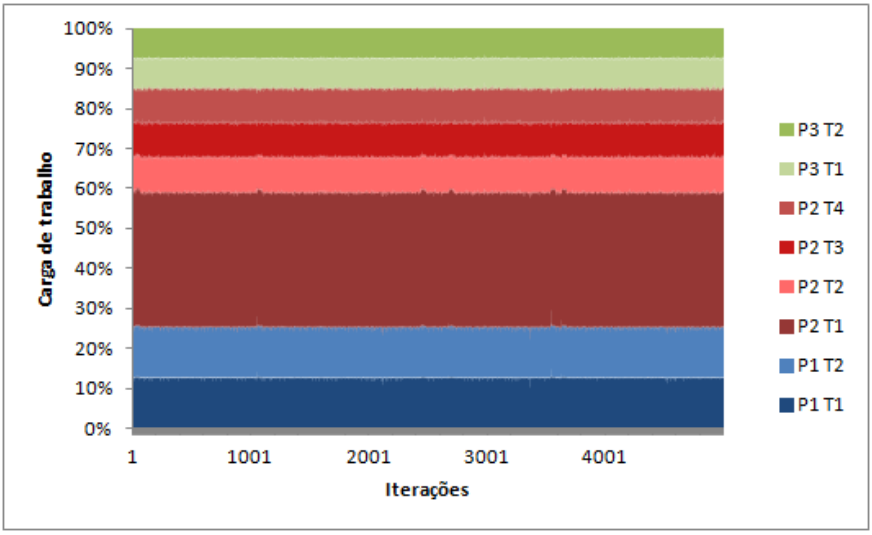


Figura 6.9: Distribuição da carga de trabalho entre as *threads* ao longo da execução

6.2 RAP

As implementações do RAP foram executadas com 5.000 tarefas e 10.000 unidades do recurso (portanto, 5.000 iterações). A Tabela 6.2 apresenta o tempo de execução de cada implementação.

Tabela 6.2: tempo de execução e *speedup* por implementação

	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
Tempo (s)	874	513	191	515	46
Speedup vs sequencial	-	1,70	4,56	1,69	19,04
Speedup vs não balanc.	-	-	2,68	-	11,18

As versões com balanceamento apresentaram *speedup* de 2,68 e 11,18 com relação às versões sem balanceamento. Novamente, a implementação balanceada com CUDA apresenta *speedup* maior que a implementação balanceada sem CUDA, pelo mesmo motivo observado nos testes do método de Jacobi. Inclusive, a implementação não balanceada com CUDA apresenta tempo médio de execução levemente maior do que a implementação sem CUDA. Isso se deve ao *overhead* de inicialização da GPU (alocação e cópia de memória) antes de iniciar as iterações.

É de se destacar o *speedup* considerável da versão balanceada com GPU, mais que 3 vezes maior que o *speedup* observado na mesma implementação no método de Jacobi. Como se pode observar nas Figuras 6.10 e 6.11, que apresentam o tempo de trabalho e a carga médios por iteração para os processos/*threads*, a GPU apresentou desempenho significativamente superior às CPUs neste problema, ficando responsável, em média, por 79% da carga da trabalho. Possivelmente a implementação do RAP se adapta melhor ao modelo de paralelismo da GPU que o método de Jacobi, permitindo um maior desempenho.

Outro ponto a se ressaltar é que, como explicado no Capítulo 5, a implementação do RAP apresenta um laço irregular, o que faz com que o custo da carga de trabalho a ser dividida não seja uniforme – de forma simplificada, o *i*-ésimo elemento da carga de trabalho tem custo *i*. Como o tamanho da carga de trabalho a ser dividida era 10.001, e a GPU ficou responsável, em média, pela carga no intervalo [1858, 9792), então, dentro desse cálculo simplificado, os 79% da carga de trabalho representam 92% do trabalho real por iteração, o que aumenta a superioridade do desempenho da GPU frente às demais *threads*.

A assimetria no custo computacional da carga de trabalho dividida pelo algoritmo de balanceamento também deve ser levada em conta ao analisar os dados. Na Figura 6.11, pode-se observar, nas versões não balanceadas, que *cores* da mesma máquina, com a mesma carga de trabalho, apresentaram tempos de trabalho diferentes em função disso. Na teoria, o algoritmo de balanceamento entende que esses *cores* possuem capacidades computacionais diferentes e altera a distribuição da carga para compensar por esse fator; na prática, os *cores* possuem a mesma capacidade, mas tal distribuição compensa a assimetria do custo da carga. Ou seja, os testes com o RAP demonstram a funcionalidade do algoritmo de balanceamento diante de problemas onde a distribuição ótima da carga é não trivial. A Figura 6.12 apresenta o tempo ocioso médio por iteração de cada *thread*, demonstrando a eficiência do balanceamento obtido.

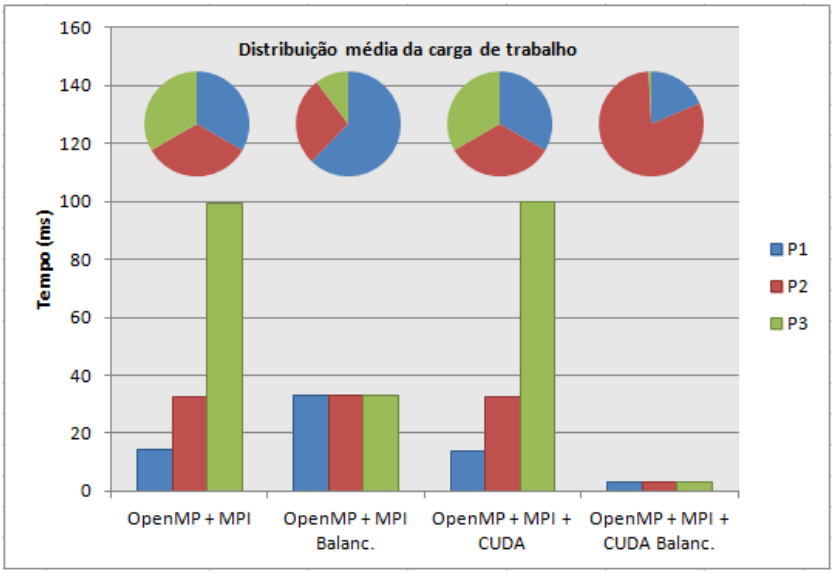


Figura 6.10: Tempo de trabalho médio por iteração por processo. Em cima, distribuição média da carga de trabalho entre os processos

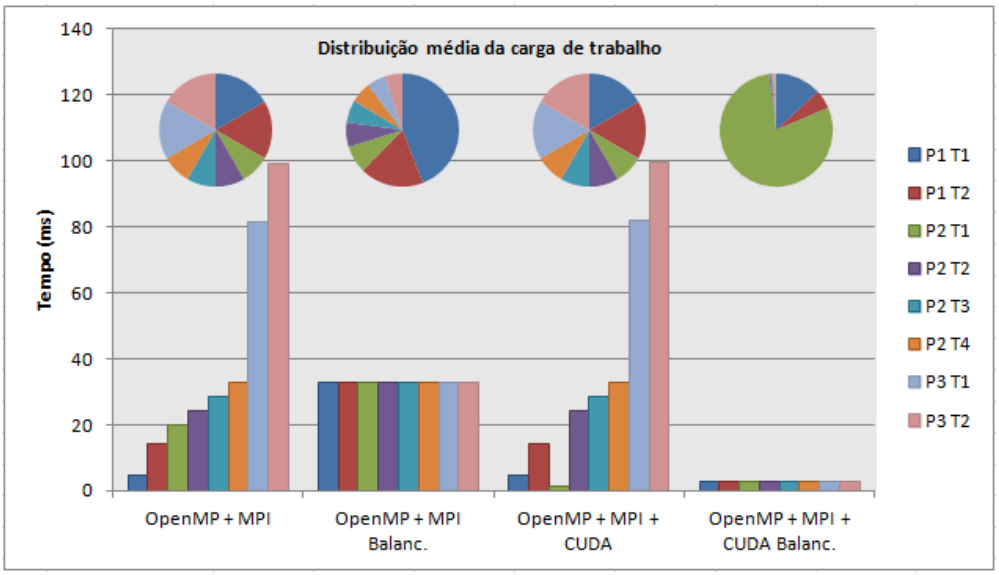
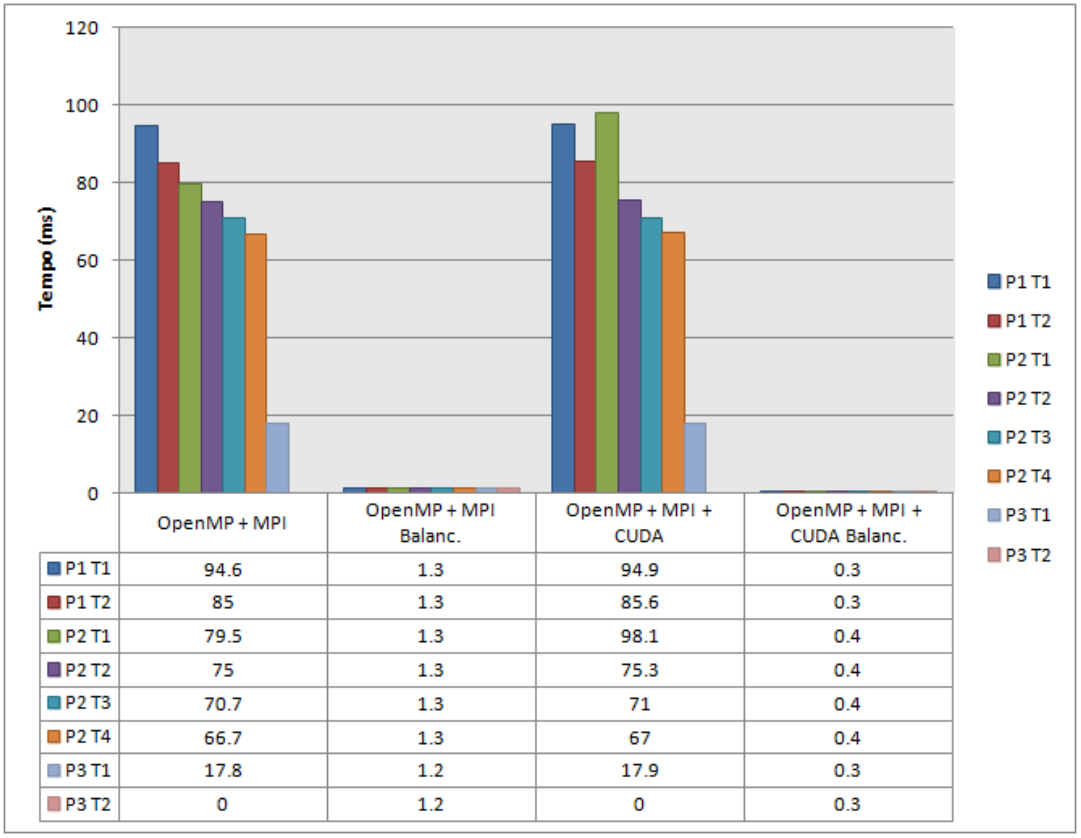


Figura 6.11: Tempo de trabalho médio por iteração por *thread*. Em cima, distribuição média da carga de trabalho entre as *threads*



	OpenMP + MPI	OpenMP + MPI Balanc.	OpenMP + MPI + CUDA	OpenMP + MPI + CUDA Balanc.
P1 T1	94.6	1.3	94.9	0.3
P1 T2	85	1.3	85.6	0.3
P2 T1	79.5	1.3	98.1	0.4
P2 T2	75	1.3	75.3	0.4
P2 T3	70.7	1.3	71	0.4
P2 T4	66.7	1.3	67	0.4
P3 T1	17.8	1.2	17.9	0.3
P3 T2	0	1.2	0	0.3

Figura 6.12: Tempo ocioso médio por iteração por *thread*

6.2.1 Tempo e Carga de Trabalho – OpenMP + MPI

As Figuras 6.13 e 6.14 apresentam o tempo de trabalho e carga dos processos e das *threads* nas primeiras iterações do teste. Novamente, a biblioteca leva poucas iterações para se aproximar da distribuição ótima de carga, embora leve mais iterações que no método de Jacobi, onde a distribuição homogênea, em arquiteturas homogêneas, seria a ótima. Essa complexidade maior na distribuição da carga pode ser observada na segunda iteração da Figura 6.14, onde ocorre o efeito inverso da primeira iteração (exceto P1 T2, as outras *threads* têm tempo de trabalho decrescente conforme sua ordem na distribuição), o que significa que o poder relativo das primeiras *threads* foi superestimado e/ou o das últimas *threads* foi subestimado. Além disso, a mesma quantidade de carga para um processo/*thread*, em iterações diferentes, pode significar tempos de trabalho diferentes, caso o início do intervalo seja diferente.

Ainda assim, logo na terceira e quarta iterações os tempos de trabalho já começam a ficar similares. Na quarta iteração, a diferença entre o maior e menor tempo de trabalho foi de 5,0 milissegundos. Na oitava iteração, a diferença foi de 1,83 milissegundos, o que a biblioteca consideraria como balanceado usando um *threshold* de 6%. A Figura 6.15 apresenta a distribuição da carga ao longo da execução.

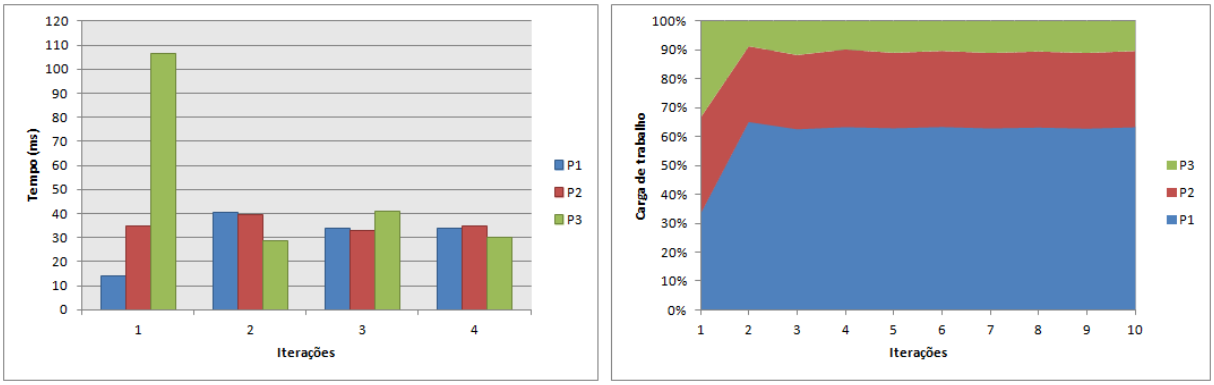


Figura 6.13: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada processo nas primeiras iterações da execução

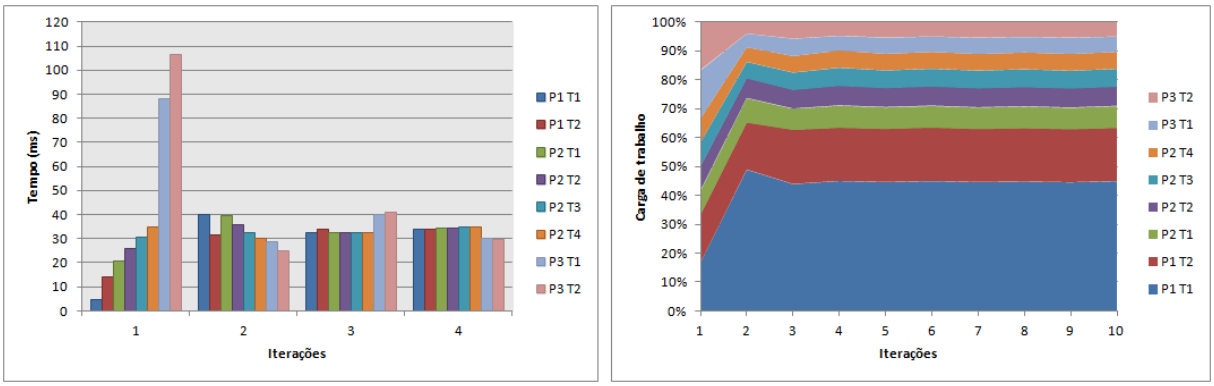


Figura 6.14: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada *thread* nas primeiras iterações da execução

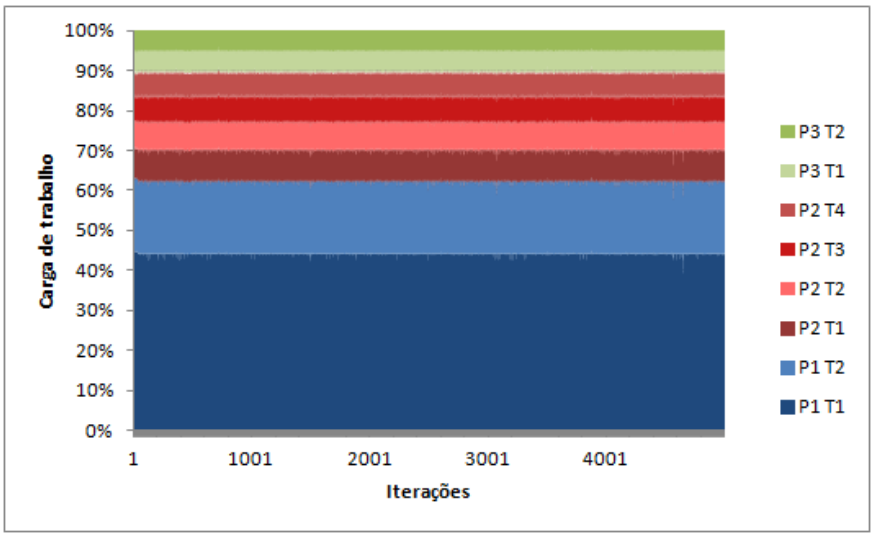


Figura 6.15: Distribuição da carga de trabalho entre as *threads* ao longo da execução

6.2.2 Tempo e Carga de Trabalho – OpenMP + MPI + CUDA

As Figuras 6.16 e 6.17 apresentam o tempo de trabalho e carga dos processos e das *threads* nas primeiras iterações do teste. Entre todas as implementações, nos três estudos de caso, esta foi a que mais demorou a estabilizar a distribuição da carga, como pode ser observado nos gráficos da direita, que mostram oscilações na divisão de carga até por volta da iteração 15, quando a divisão começa a se estabelecer, o que também pode ser visto na Figura 6.18, que ilustra o tempo de execução dos processos até a vigésima iteração. De fato, na iteração 4, que pode ser observada nos gráficos de tempo de iteração, à esquerda, a diferença entre maior e menor tempo de trabalho entre as *threads* era 6,1 milissegundos, correspondendo a 211% do menor tempo observado. Já na iteração 15, a diferença era de 0,2 milissegundos, correspondendo a 7% do menor tempo observado.

Além da complexidade relacionada ao balanceamento advinda do problema em si, o desempenho da GPU fez com que o tempo médio de trabalho de cada *thread* por iteração fosse de 2,9 milissegundos; quanto menor o tempo de trabalho, maior a influência de fatores externos à execução. Isso pode ser visto na Figura 6.18 – nas iterações 14 e 15, o processo 3 recebeu carga de 71 unidades e processou-a em 2,90 milissegundos, em ambas as iterações. Na iteração 16, a carga aumentou para 72 unidades, mas o tempo de trabalho foi de 2,40 milissegundos, por fatores de tempo de execução ao acaso. Esses 0,5 milissegundos a menos

significaram que, na iteração 17, a carga deste processo aumentou de 72 unidades para 90 – isto sim, ocasionando um pequeno desbalanceamento no sistema, uma vez que, para processar 90 unidades, o processo 3 levou 3,66 milissegundos. Na iteração seguinte, o desbalanceamento é corrigido, e a carga do processo 3 voltou a ser de 72 unidades.

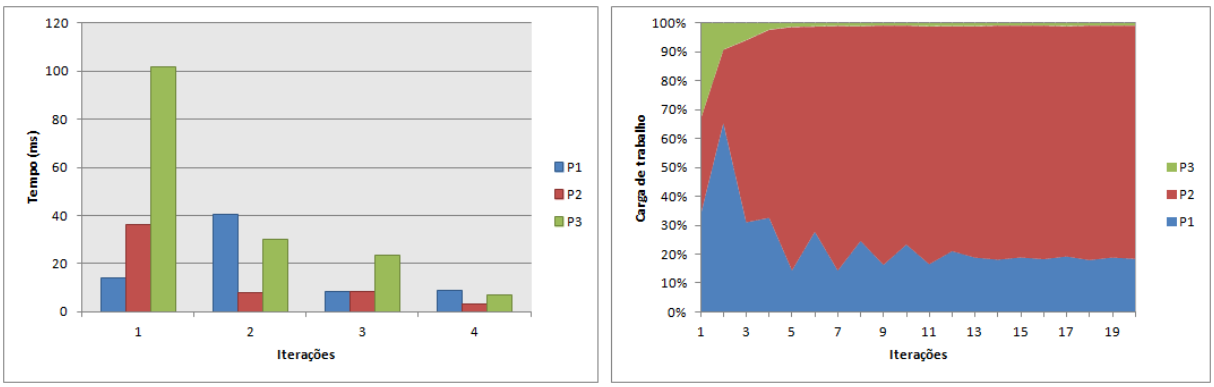


Figura 6.16: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada processo nas primeiras iterações da execução

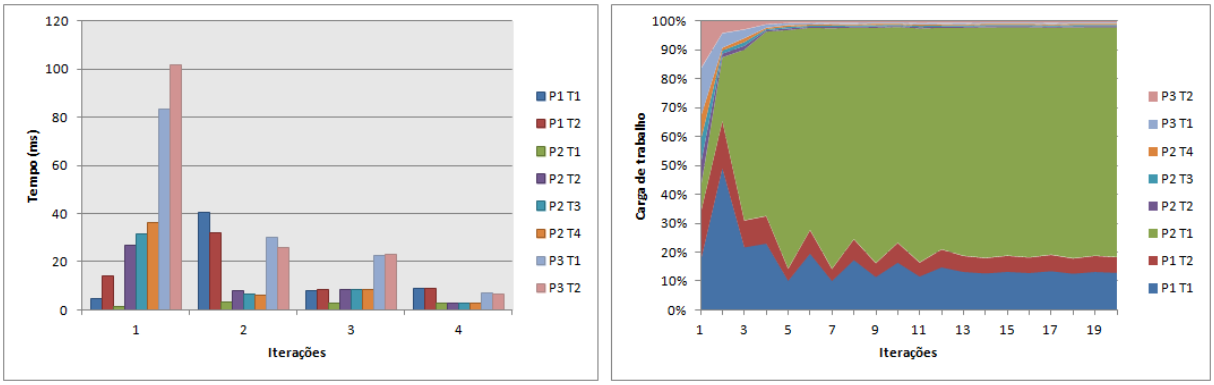


Figura 6.17: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada *thread* nas primeiras iterações da execução

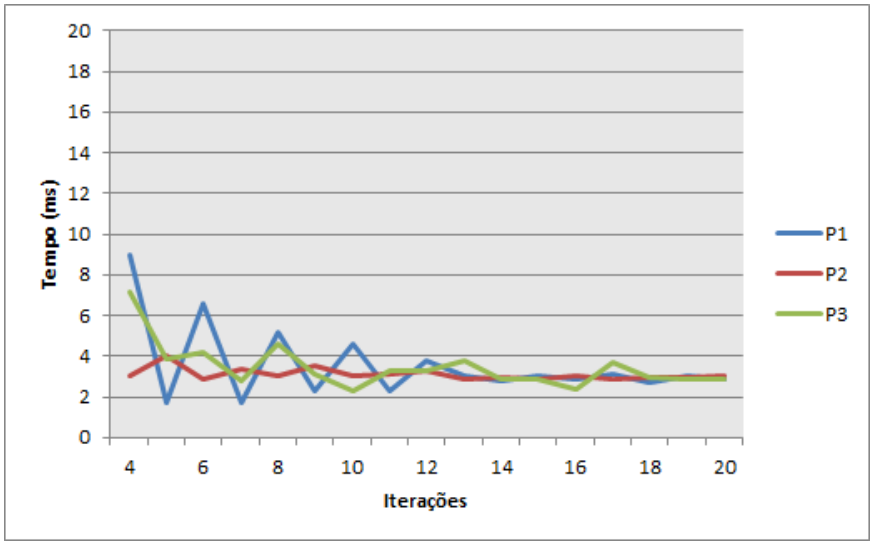


Figura 6.18: Tempo de trabalho de cada processo nas iterações 4-20 da execução

A Figura 6.19 ilustra a distribuição da carga ao longo da execução. É possível observar que, após as primeiras iterações, a distribuição da carga se estabiliza pelo resto da execução, embora ocorra uma leve variação em cada iteração, além de breves momentos com variações maiores, provavelmente em função de fatores em tempo de execução.

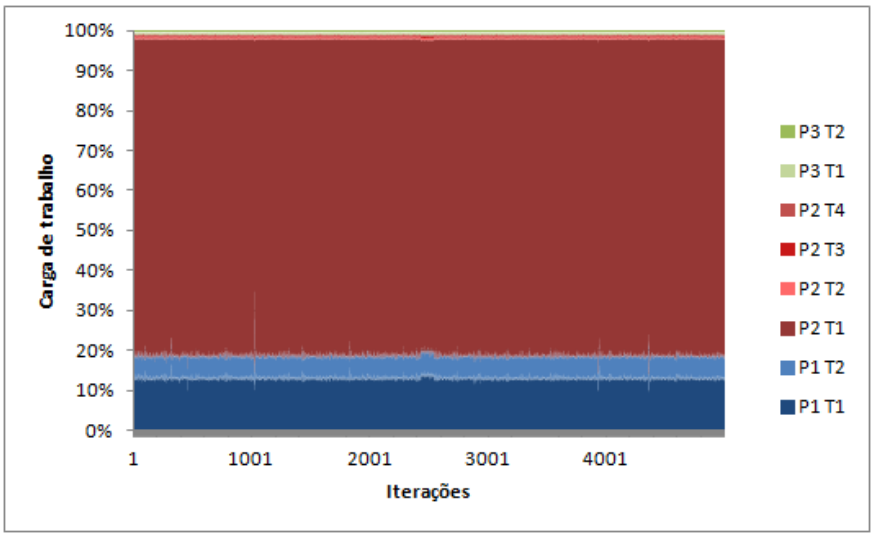


Figura 6.19: Distribuição da carga de trabalho entre as *threads* ao longo da execução

6.3 Transferência de Calor

Para este problema, foram implementadas as versões sequencial e paralela com OpenMP e MPI, com e sem balanceamento, a fim de ilustrar uma estratégia diferente de utilização da biblioteca de balanceamento em um problema que não se encaixa totalmente no modelo de aplicação especificado na Seção 4.1. As simulações do problema da transferência de calor foram executadas considerando um domínio de tamanho 10.000x10.000, sobre 10.000 iterações. Ressalta-se que a implementação paralela balanceada deste problema utilizou uma abordagem diferente com duas fases: na primeira é calculado o balanceamento, e na segunda mantém-se a distribuição de carga fixa, de acordo com o calculado na primeira fase. Além disso, os testes foram executados no mesmo ambiente computacional, porém com a tecnologia de *Hyper-Threading* ativada nas máquinas 1 e 2; por isso, estas máquinas aparecem com 4 e 8 *threads*, respectivamente.

A Tabela 6.3 apresenta o tempo de execução das implementações.

Tabela 6.3: tempo de execução e *speedup* por implementação

	Sequencial	OpenMP + MPI	OpenMP + MPI Balanc.
Tempo (s)	12137	4612	3265
Speedup vs sequencial	-	2,63	3,72
Speedup vs não balanc.	-	-	1,41

A versão balanceada apresentou *speedup* de 1,41 em relação à versão não balanceada. A Tabela 6.4 apresenta o tempo de execução médio por iteração, ilustrando a diferença entre o tempo de execução na fase 1 da implementação balanceada e na fase 2.

Observa-se que, na primeira etapa do código, cada iteração levou aproximadamente 128 vezes o tempo de uma iteração da implementação não balanceada. Isso ocorre pela quantidade de troca de dados entre os processos: na implementação sem balanceamento, cada processo deve trocar 2 linhas do *grid* com outros dois processos; já na fase 1 da etapa balanceada, os processos devem compartilhar entre si todo o *grid*. Na segunda etapa, o balanceamento permite que o tempo médio de cada iteração seja menor. Ou seja, paga-se um custo inicial para encontrar o balanceamento na primeira etapa, e a partir de então a cada iteração há um

ganho em função do balanceamento. A Figura 6.20 ilustra isso, mostrando o tempo médio até a execução atingir a *i*-ésima iteração.

Tabela 6.4: tempo de execução médio por iteração

Implementação	Tempo (s)
Sequencial	1,21
OpenMP + MPI	0,46
OpenMP + MPI Balanc., fase 1 (iterações 1-4)	58,87
OpenMP + MPI Balanc., fase 2 (iterações >4)	0,30

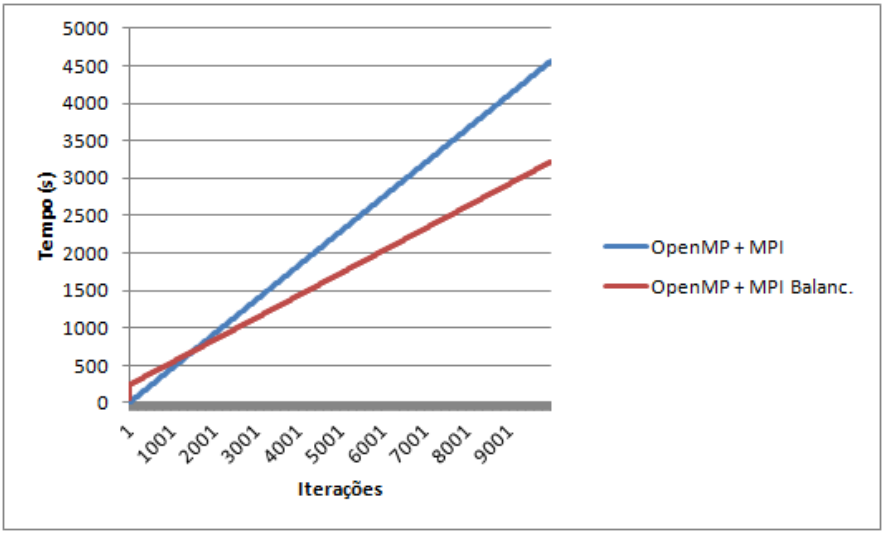


Figura 6.20: Tempo de execução médio até a *i*-ésima iteração da versão balanceada e não balanceada

6.3.1 Tempo e Carga de Trabalho – OpenMP + MPI

As Figuras 6.21, 6.22 e 6.23 apresentam o tempo de trabalho e carga dos processos e das *threads* nas primeiras iterações do teste. É possível observar que, mesmo com apenas 4 iterações na primeira etapa do código, atinge-se um nível satisfatório de balanceamento; na realidade, entre as iterações 3 e 4, não há alteração no balanceamento, pois a iteração 3 já foi considerada balanceada com o *threshold* de 10% que foi utilizado, como pode ser observado na Figura 6.24, que compara a diferença entre maior e menor tempo de trabalho entre os processos e a diferença necessária entre maior e menor tempo para ser considerado balanceado com o parâmetro de 10%.

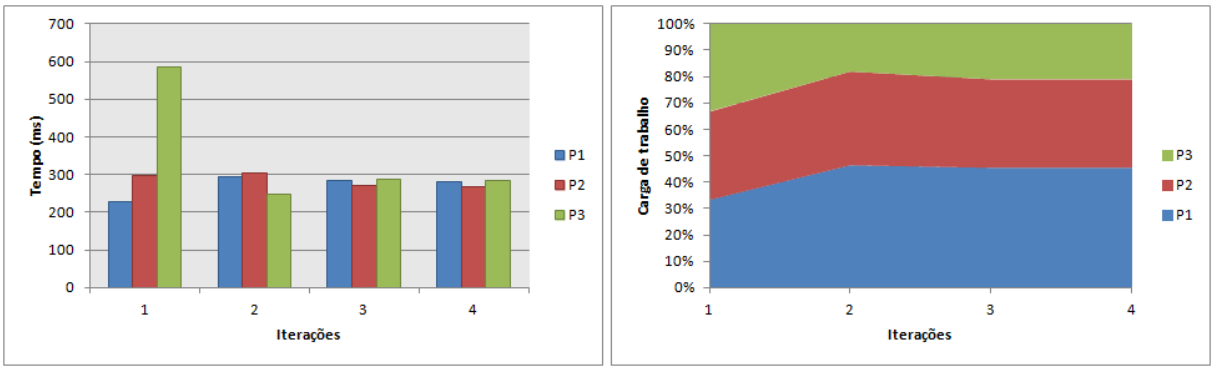


Figura 6.21: Tempo de trabalho (esquerda) e distribuição da carga (direita) de cada processo nas primeiras iterações

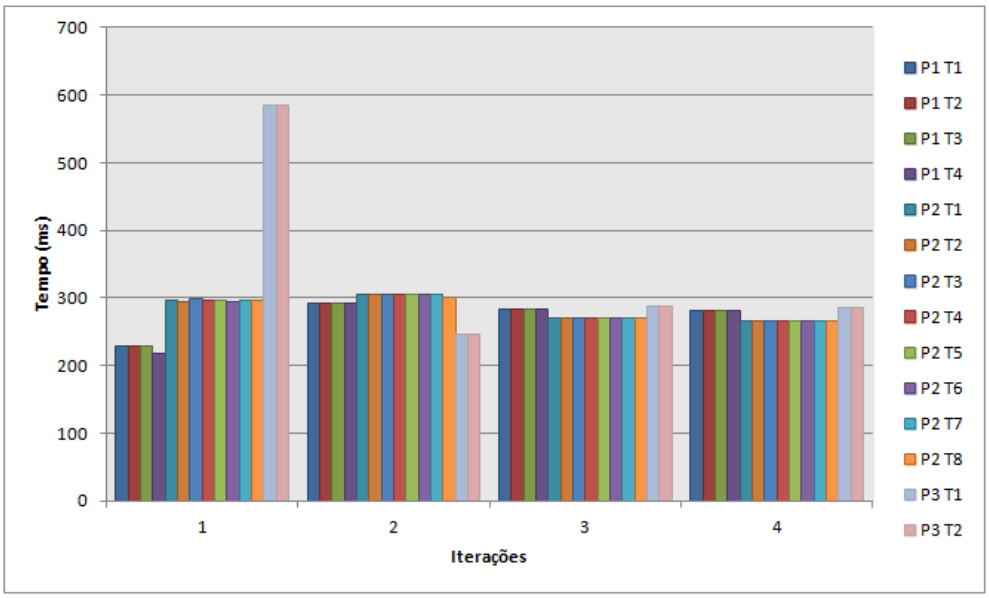


Figura 6.22: Tempo de trabalho de cada *thread* nas primeiras iterações

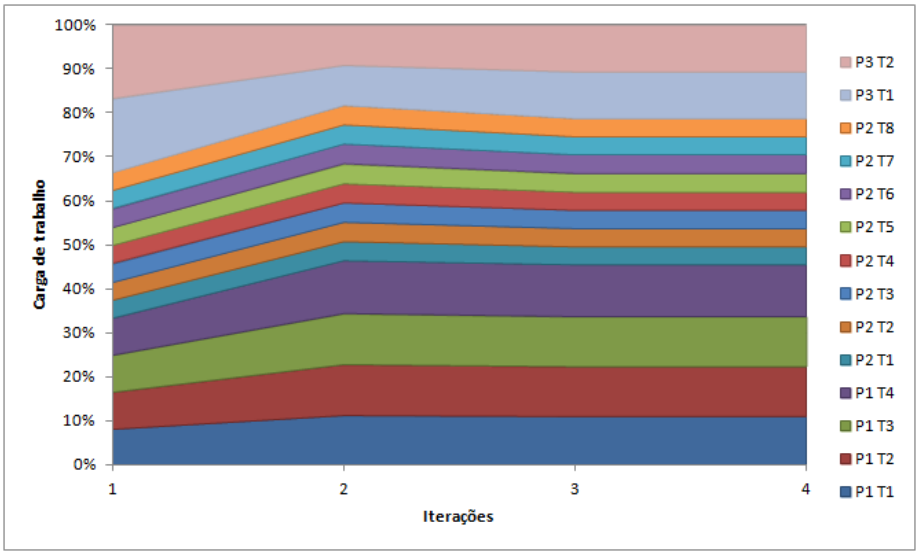


Figura 6.23: Carga de trabalho de cada *thread* nas primeiras iterações

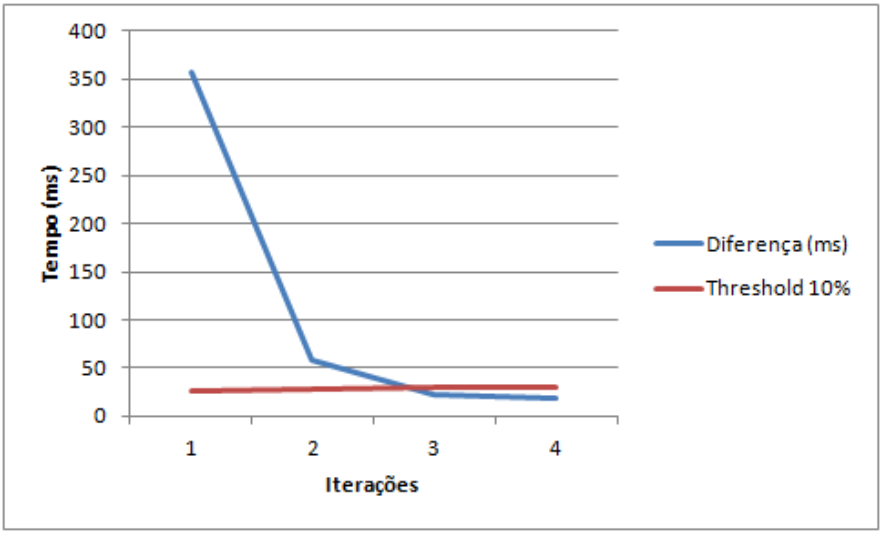


Figura 6.24: Diferença em milissegundos entre o maior e menor tempo de processamento em cada iteração entre os processos vs diferença necessária para que a biblioteca considere a carga balanceada com um *threshold* de 10%

6.4 Considerações Finais

A Tabela 6.5 apresenta um resumo dos *speedups* observados entre as implementações balanceadas e as implementações correspondentes não balanceadas.

Tabela 6.5: *speedup* entre implementações balanceadas e não balanceadas

Implementação		*Speedup* vs não balanceado
Método de Jacobi	OpenMP + MPI	1,55
	OpenMP + MPI + CUDA	1,92
RAP	OpenMP + MPI	2,68
	OpenMP + MPI + CUDA	11,18
Transferência de calor	OpenMP + MPI	1,41

Pode-se observar que, em todas as implementações, houve ganho no tempo de execução com o uso da biblioteca Multiframework Balance. O menor *speedup* observado foi de 1,41, no problema da transferência de calor, cuja estratégia de implementação difere dos demais problemas devido às dependências de dados do problema. Já o maior *speedup* foi de 11,18, no RAP implementado com OpenMP, MPI e CUDA, no qual o balanceamento permitiu explorar melhor o potencial computacional da GPU. De fato, observou-se que, nas implementações sem balanceamento, não houve benefício algum entre utilizar ou não a GPU, desperdiçando completamente tal potencial.

A melhor exploração da arquitetura paralela decorrente do balanceamento pode ser observada na Tabela 6.6, que apresenta a taxa de utilização dos processadores. A taxa de utilização foi calculada como a razão entre o tempo de trabalho de todas as *threads* e o tempo disponível de todas as *threads* por iteração, que é o tempo de trabalho mais o tempo ocioso por *thread*.

Tabela 6.6: taxa de utilização da arquitetura paralela nas implementações balanceadas e não balanceadas

Implementação		Taxa de utilização sem balanceamento	Taxa de utilização com balanceamento
Método de Jacobi	OpenMP + MPI	61,2%	98,5%
	OpenMP + MPI + CUDA	57,2%	98,0%
RAP	OpenMP + MPI	38,4%	96,3%
	OpenMP + MPI + CUDA	36,0%	89,7%

Observa-se que, utilizando a biblioteca de balanceamento, a taxa de utilização saltou, em três dos quatro casos, para mais de 96%. A menor taxa de utilização foi observada na implementação com OpenMP, MPI e CUDA do RAP. Um dos fatores que explica isso é a já discutida complexidade maior da separação de carga neste problema, que significa que pequenas alterações podem causar pequenos desbalanceamentos temporários. Outro fator é que esta foi a implementação com o menor tempo médio de trabalho por iteração, fazendo com que tempos ociosos pequenos, como os causados por oscilações da máquina em tempo de execução, representem um percentual maior do tempo disponível. De fato, esta implementação é a que apresenta o menor tempo ocioso médio absoluto entre todas as implementações, com as *threads* variando entre 0,3 e 0,4 milissegundos de ociosidade por iteração.

Outro ponto observado foi o número de iterações necessárias para que o sistema entrasse um estado considerado balanceado sob um dado valor de *threshold*, apresentado na Tabela 6.7. Nos testes com o método de Jacobi, mesmo com *threshold* de 1%, o teste que mais demorou em alcançar o balanceamento levou apenas 20 iterações, representando 0,4% das 5.000 iterações totais do teste. Na média, todas as implementações do método de Jacobi ficaram balanceadas em no máximo 6 iterações. Já no caso do RAP, alguns testes demoraram mais para entrar no *threshold* de balanceamento, com o pior teste levando 373 iterações, e 107,7 iterações em média. No caso da transferência de calor, que não está na tabela em função

da estratégia de implementação diferente, em todos os testes o *threshold* de 10% utilizado foi alcançado na 3ª das 4 iterações com balanceamento.

Tabela 6.7: número de iterações para alcançar sistema balanceado sob *thresholds* de 5% e 1%

Implementação		*Threshold* = 5%		*Threshold* = 1%	
		Média	Maior	Média	Maior
Método de Jacobi	OpenMP + MPI	1,6	5	6	20
	OpenMP + MPI + CUDA	2,0	3	5,8	12
RAP	OpenMP + MPI	7,4	16	54,2	185
	OpenMP + MPI + CUDA	24,3	45	107,7	373

O *overhead* introduzido pela biblioteca nas implementações é apresentado na Tabela 6.8. O *overhead* foi calculado comparando-se o tempo total de execução de duas implementações com OpenMP + MPI, com e sem a biblioteca, a partir da média obtida em 10 testes. Na implementação com a biblioteca, os métodos de balanceamento eram chamados normalmente, mas a distribuição de carga calculada pela biblioteca era ignorada, mantendo-se uma mesma distribuição pré-definida e inserida no código em ambas implementações.

Tabela 6.8: *overhead* de tempo de execução introduzido pela biblioteca

Implementação	*Overhead* por iteração	*Speedup* com biblioteca
Método de Jacobi	2,0 ms	0,96
RAP	1,0 ms	0,97

Observa-se que o *overhead* por iteração variou entre 1 e 2 milissegundos. O tempo necessário para executar os métodos da biblioteca é proporcional ao número de processos MPI e *threads* OpenMP utilizados. Em particular, o número de processos influencia na comunicação coletiva efetuada pela biblioteca para compartilhar os tempos de execução entre os processos. O *speedup* apresentado na tabela refere-se ao tempo da execução da implementação com a biblioteca comparada ao tempo da implementação sem a biblioteca. Este valor é relativo ao trabalho realizado por iteração no problema; quanto mais trabalho, mais negligível torna-se o *overhead* introduzido.

Por fim, um dos objetivos do trabalho era permitir que a solução desenvolvida para o balanceamento de carga pudesse ser integrada a códigos existentes sem requerer alto custo de

programação. Como foi demonstrado nos Capítulos 4 e 5, uma vez que a biblioteca utiliza estruturas comuns a aplicações paralelas, como os vetores *counts* e *displs*, presentes em métodos definidos pelo padrão MPI, é possível fazer uso da biblioteca de forma pouco intrusiva ao código. Na Seção 5.1, pode-se observar que o código paralelo sem balanceamento para o método de Jacobi foi adaptado para usar a biblioteca alterando-se apenas 2 linhas de código, além de adicionar outras 4 linhas para chamar os métodos definidos pela biblioteca.

Capítulo 7

Conclusão

Neste trabalho, abordou-se soluções para o balanceamento de carga em aplicações paralelas iterativas voltadas para execução em arquiteturas heterogêneas. Observou-se a necessidade de realizar uma boa distribuição da carga de trabalho a fim de evitar desperdício de recursos computacionais, tarefa cuja dificuldade se acentua ao se programar para arquiteturas heterogêneas em função da discrepância natural entre as capacidades computacionais de diferentes *hardwares*, e também da falta de ambientes de programação estabelecidos que tornem homogênea a exploração de cada arquitetura.

Embora se caminhe para uma simplificação da programação em arquiteturas heterogêneas, com novas ferramentas sendo desenvolvidas, deve-se considerar que há uma quantidade razoável de código já existente escrito sem tais facilidades, e reescrevê-los ou mesmo adaptá-los para utilizar tais ferramentas pode ser uma tarefa complexa e trabalhosa. Diante disso, pontuou-se a relevância de analisar soluções para o balanceamento de carga que permitam adaptação a códigos já existentes de forma pouco custosa.

Dessa forma, foi desenvolvida uma biblioteca de balanceamento utilizando duas das tecnologias mais populares para programação paralela, o MPI e o OpenMP. A biblioteca foi desenvolvida tendo em mente a exploração de arquiteturas heterogêneas escaláveis, permitindo a formação de *clusters* onde cada nó pode contar com múltiplos *cores* e aceleradores em geral. Observou-se ser possível estruturar tal biblioteca de forma a minimizar a necessidade de alterar códigos já existentes para incluí-la, bastando por vezes modificar em torno de cinco linhas do código.

O algoritmo de balanceamento utilizado, uma extensão do algoritmo apresentado por (ACOSTA; BLANCO; ALMEIDA, 2013), adequando-o para dois níveis de balanceamento, demonstrou bom desempenho, alcançando níveis satisfatórios de balanceamento em menos de

25 iterações (0,5% do total de iterações dos testes), em média, em todos os testes, e menos de 10 iterações em mais da metade dos testes, introduzindo *overhead* negligível à execução, entre 1 e 2 milissegundos por iteração. Isso permitiu elevar a taxa de utilização das arquiteturas para mais de 96% na maior parte dos testes, em contraste com a taxa de utilização observada entre 36% e 61% nos testes com distribuição homogênea da carga. Assim, obteve-se melhora no tempo de execução em todas as implementações com a biblioteca de balanceamento, com *speedup* relativo às versões não balanceadas variando entre 1,41 até 11,18, permitindo uma melhor exploração da arquitetura paralela.

Conclui-se então que, em aplicações iterativas com sincronização, realizar um balanceamento de carga adequado, embora uma tarefa não trivial em arquiteturas heterogêneas, constitui um aspecto fundamental para obter desempenho de acordo com a capacidade da arquitetura. Os testes demonstram ainda que a adição de componentes mais poderosos pode ser completamente infrutífera se não forem tomados os cuidados para garantir que tais componentes não sejam limitados por meio de gargalos gerados por outras unidades de processamento.

Uma dificuldade ocorrida durante os testes foi a presença de outros processos na máquina influenciando a medição do tempo de trabalho das *threads*. Em alguns momentos, isso gerou uma assincronia entre as *threads* de uma das máquinas, dentro de uma mesma iteração, que distorcia o balanceamento, causando perda de desempenho. Após reduzir o número de *threads* nesta máquina, o problema não ocorreu mais. Uma vez que o programador tem pouco controle sobre o escalonamento de processos realizado pelo sistema operacional, ressalta-se a importância de conferir a forma como os processadores são alocados às tarefas, particularmente neste caso, onde fatores externos à implementação podem influenciar na distribuição de carga.

Por fim, uma limitação apresentada pela biblioteca de balanceamento é a restrição de seu uso em função de uma classe particular de aplicações paralelas. Para realizar o balanceamento da simulação da transferência de calor, por exemplo, utilizou-se uma abordagem diferente dos demais estudos de caso, em função da ausência de compartilhamento do domínio completo por todos os processos em execução.

Assim, como trabalhos futuros sugere-se:

a) o desenvolvimento de técnicas para a ampliação da classe de aplicações paralelas alcançadas pela biblioteca, tal como os problemas com dependência de dados mais complexas;

b) a avaliação do algoritmo de balanceamento utilizado em problemas com carga de custo dinâmico - isto é, o custo de cada elemento pode variar ao longo da execução;

c) no contexto de aplicações que apresentem um custo por alteração da distribuição de carga, a avaliação de outros métodos de minimização de oscilações na distribuição, além da adoção do *threshold*, ou mesmo de outros algoritmos de balanceamento;

d) a avaliação de outros algoritmos de escalonamento e balanceamento baseados em outros critérios que não o tempo de execução, como o consumo de energia;

e) a extensão do funcionamento da biblioteca para outras tecnologias frequentemente empregadas em aplicações paralelas.

Referências Bibliográficas

ACOSTA, A.; BLANCO, V.; ALMEIDA, F. Dynamic load balancing on heterogeneous multi-GPU systems. *Computers & Electrical Engineering*, v. 39, n. 8, p. 2591–2602, 2013.

ACOSTA, A. *et al.* Dynamic Load Balancing on Heterogeneous Multicore/MultiGPU Systems. *2010 International Conference on High Performance Computing & Simulation*, p. 467-476, 2010.

AUGONNET, C. C. A. *et al*. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Lecture Notes in Computer Science Euro-Par 2009 Parallel Processing*, p. 863–874, 2009.

BARNEY, B. *Introduction to Parallel Computing*. Consultado na INTERNET: https://computing.llnl.gov/tutorials/parallel_comp/, 2018.

BELVIRANLI, M. E.; BHUYAN, L. N.; GUPTA, R. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization*, v. 9, n. 4, p. 1–20, Janeiro 2013.

BINOTTO, A. P. D. *A Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms*. Tese (Tese de Doutorado) – Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Setembro 2011.

BOSQUE, J. L. *et al*. A load index and load balancing algorithm for heterogeneous clusters. *The Journal of Supercomputing*, v. 65, n. 3, p. 1104-1113, Setembro 2013.

BORGES, P. A. P.; PADOIN, E. L. Exemplos de métodos computacionais aplicados a problemas na modelagem matemática. In: *ESCOLA REGIONAL DE ALTO DESEMPENHO DO ESTADO DO RIO GRANDE DO SUL*, 2006. Porto Alegre: Biblioteca do Instituto de Informática da UFRGS, 2006, p. 5-20.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. *Communications of the ACM*, New York, v. 54, n. 5, p. 67-77, Maio 2011.

CHAPMAN, B; JOST, G; PAS, R. *Using OpenMP: Portable shared memory parallel programming*. Cambridge, MA: MIT Press, 2008.

CHEN, L. *et al*. Dynamic load balancing on single- and multi-GPU systems. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

GALINDO, I.; ALMEIDA, F.; BADÍA-CONTELLES, J. M. Dynamic Load Balancing on Dedicated Heterogeneous Systems. *Recent Advances in Parallel Virtual Machine and Message Passing Interface Lecture Notes in Computer Science*, p. 64–74, 2008.

GARLAND, M.; KIRK, D. B. Understanding throughput-oriented architectures. *Communications of the ACM*, v. 53, n. 11, p. 58, Janeiro 2010.

KHRONOS. *OpenCL Overview*. Consultado na INTERNET: https://www.khronos.org/opencl/, 2017. Data de acesso: Agosto 2017.

LAMA, C. S. D. L. *et al*. Static Multi-device Load Balancing for OpenCL. *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, 2012.

LI, L. *et al*. Experience of parallelizing cryo-EM 3D reconstruction on a CPU-GPU heterogeneous system. *Proceedings of the 20th international symposium on High performance distributed computing - HPDC 11*, 2011.

LU, F. *et al*. CPU/GPU computing for long-wave radiation physics on large GPU clusters. *Computers & Geosciences*, v. 41, p. 47–55, 2012a.

LU, F. *et al*. Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters. *Computer Physics Communications*, v. 183, n. 6, p. 1172–1181, 2012b.

MAILLARD, N; SCHEPKE, C. Programação em Ambientes Computacionais com Múltiplos Níveis de Paralelismo. In: *ANAIS – 8A ESCOLA REGIONAL DE ALTO DESEMPENHO*. Porto Alegre - RS: SBC/UFPEL/UNISC/UCS, 2008. p. 141-142.

MITTAL, S.; VETTER, J. S. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, v. 47, n. 4, p. 1–35, 2015.

NVIDIA. *NVIDIA GeForce GTX 1080*. Consultado na INTERNET: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016.

PACHECO, P. Parallel Programming with MPI. 1st ed. Boston: Morgan Kaufmann Publishers Inc, 1997.

PACHECO, P. *An Introduction to Parallel Programming*. 1st ed. San Francisco, CA, Estados Unidos: Morgan Kaufmann Publishers Inc, 2011.

PINTO, V. G. *Ambientes de Programação Paralela Híbrida*. Porto Alegre, RS: Universidade Federal do Rio Grande do Sul, Dezembro 2011. Relatório Técnico 1.

SCHEPKE, C. *Ambientes de Programação Paralela*. Porto Alegre, RS: Universidade Federal do Rio Grande do Sul, Março 2009. Relatório Técnico 1.

SILVA, C. *Programação Genética Maciçamente Paralela em GPUs*. Tese (Tese de Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Setembro 2014.

SNIR, M. *et al*. *MPI: The Complete Reference*. 2nd ed. Cambridge, MA, Estados Unidos: MIT Press, 1998.

TSE, A. H. *et al*. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. *2010 International Conference on Field-Programmable Technology*, 2010.

YELICK, K. *et al*. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, v. 10, n. 11-13, p. 825–836, 1998.